NOTE TO USERS

Time Inconsistency Analysis and Correction for MSC-2000 Specifications

LiXin Wang

A Thesis

In

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montreal, Quebec, Canada

July 2005

Your file Votre référence
ISBN: 0-494-10254-3
Our file Notre référence
ISBN: 0-494-10254-3

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Mr. LiXin Wang

Entitled: Time Inconsistency Analysis and Correction for MSC-2000 Specifications

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

Master of Applied Science

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

Signed by the final examining committee:

_____ Chair

_____ Examiner

_____ Examiner

_____ Supervisor

Approved by _____
Chair of Electrical and Computer Engineering

_____20___ _____
Dean of Faculty

ABSTRACT

Time Inconsistency Analysis and Correction for MSC-2000 Specifications

LiXin Wang

Message Sequence Charts (MSC) plays an important role in the software life cycle. It is widely used in the requirements, design, and test phases for different purposes. Therefore, it is crucial to insure the correctness of MSC specifications. For that, one has to validate these MSC specifications as early as possible in the development cycle. An important aspect of MSC-2000 specifications correctness is time consistency, including absolute and relative time constraints. The time consistency of Basic MSCs (bMSCs) and High Level MSCs (HMSCs) has been investigated in the last few years. However, a very little effort and research have been devoted for the purpose of investigation and diagnosis of causes of inconsistencies, and also strategies to correct these inconsistencies.

This thesis focuses on the consistency checking of MSCs, analyzes and categorizes the causes of time inconsistencies, and provides solutions to correct time inconsistencies. For bMSCs, we classify partial order, propose solutions to identify inconsistency, and provide four correction policies for the different types of inconsistencies. For HMSCs, we decompose them into different simple paths, provide checking algorithms based on the type of the simple paths, and use different correction strategies for these inconsistent paths. Our approach helps MSC developers to insure the correctness of MSC specifications, thereby improving its quality and also that of the resulting software. A

tool, called MSCTICC, implementing our algorithms has been developed and assessed in this thesis.

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. The thesis would not have been possible without the substantial time and effort he has devoted to it.

I would like to thank Christophe Lohr who gave me many helpful ideas about time consistency checking and tracking algorithms. I also thank him for proofreading my proposal for the bMSC and HMSC inconsistency. I also thank my friends Tong Zheng, XiaoJun Zhang and Yang Liu for their advice and help when I entered the Telesoft research group.

At last, I would also like to thank my wife Hu. I owe her a great deal for her endless love and understanding in every situation. I am indebted to her for giving me comfort and taking care of our baby so that I could finish my thesis. I dedicate this thesis to her and to my son QiQi.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 MSC roles in the software life cycle

A reactive system interacts with its environment. It receives inputs that may arrive in a continuous way and in unexpected sequences from the environment [17]. Examples of reactive systems are telephony systems, air traffic control systems, traffic light control systems, etc. Most reactive systems can be seen as real-time systems and have to satisfy certain timing and concurrency requirements.

Software processes involve a set of activities that encompass the entire software life cycle [7]. The major phases are software requirements gathering and analysis, design, implementation, testing, operation and maintenance.

In the requirement phase, software engineers gather requirements from stakeholders, specify software features, and prioritize these features. The artifact of this phase is the requirement specification that is used to describe the behaviors and the properties of the system. In the design phase, the system specification is used to describe the high level architecture of the system, the interface between the components of the system, and the behavior of each component. In the implementation phase, the design is coded in a concrete programming language. Testing comes as the next phase. The software produced is tested to ensure its conformance to the functionality and performance

requirements specified in the requirements documents. Maintenance is conducted during the entire software life for modifications and updates.

Requirements specifications and system design specifications have to describe the system unambiguously. Therefore, formal languages are necessary for writing the specifications. Formal methods are mathematics based techniques [22]. Formal methods are based on formal languages. Formal specifications are analyzed to discover inconsistencies or design errors.

As a formal language, the graphical and textual notation called Message Sequence Charts (MSC) [10] was initially developed by International Telecommunication Union – Telecommunication Standardization Sector (ITU-T) as a companion to the Specification and Description Language (SDL) [11]. MSC has become one of the key notations for the requirement, design and test phases, especially in the telecommunication field. In the different phases of the software development cycle, MSC specifications serve different purposes and exhibit different levels of details as indicated in Figure 1.1. MSCs are used in the requirements phase to describe use cases or scenarios, and to describe the interactions between a system and its environment. The crucial step is then to translate the MSC model into an architectural model supporting the design of the implementation. To ensure the conformance, we can verify system design specification against the MSC requirement specification. Given target architecture, an SDL specification can be generated from the detailed MSC specification [24]. The MSC specifications can also be

used as test cases for testing purpose. Therefore, MSC specifications take a central role in the development process, and their correctness is crucial.

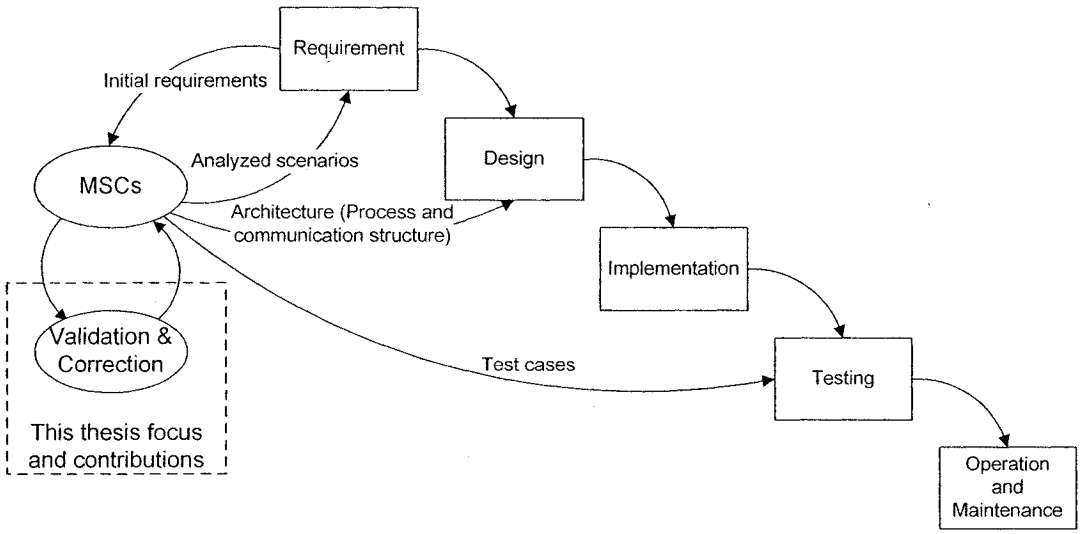


Figure 1.1 MSCs roles in the software life cycle

1.2 Motivations

To be useful in a development process, MSC specifications must not contain semantic errors. They have to be validated. An MSC specification may contain semantic errors or logical inconsistencies, so its accuracy has to be checked. Any error detected in the earlier stage will get a high pay-off since MSCs are used as the basis of the design [3]. The introduction of the time constraints in MSC-2000[10] raises the time consistency issue of the MSC specifications.

Several criteria for the correctness, such as the absence of race conditions, process divergences, confluence and inferences [2, 3], matching templates [16], etc., have been established, and techniques for checking these criteria have been developed. Some tools,

such as MESA [15], have been developed for the validation. However, in these works, the timing constraints and requirements are not considered.

Tong Zheng in his PhD research has investigated the validation of timed MSC specifications [26, 27]. He developed techniques and algorithms to validate the time consistency of MSC-2000 specifications. He treated events in MSCs as basic units and defined the semantics for bMSCs and HMSCs with compositional constructs. He proposed sufficient and necessary conditions for a bMSC and for an HMSC consistency. Following these conditions, he designed algorithms to check the consistency. In summer 2002, we implemented and integrated the algorithms into the tool MSCTCC [23].

However, there has been no effort in the works mentioned above to diagnose the reason for time inconsistency and no research has been done to provide potential solutions to correct the time-inconsistent MSCs. This lack of effort and research on the time inconsistency of MSCs initiates the research topic of this thesis.

In this thesis, we investigate further the validation of the timed MSC specifications. One of our goals is to analyze the different the causes for inconsistency. The results of the analysis form the basis for the algorithms and solutions used in the correction of inconsistent bMSCs. Another goal is to develop algorithms for diagnosing the inconsistent bMSCs to find out the causes for time inconsistency if an inconsistent bMSC is detected. Therefore, the last and the most important goal is to find methods to correct inconsistent bMSC specifications.

As HMSC is concerned, our purpose is to find appropriate approaches to transform weakly consistent and inconsistent HMSCs to strongly consistent HMSCs. If an HMSC is partially consistent (weakly consistent) or totally inconsistent, the HMSC is not valid from the semantics point of view. These errors and inconsistencies make the HMSC specifications un-implementable, or result in undesired implementations. Especially, when MSCs is worked with SDLs to transfer the MSC specifications to SDL architectures, the inconsistent time constraints in MSCs can cause the SDLs to be invalid. Our purpose is to correct these time inconsistencies in HMSCs and to provide potentially consistent ones.

1.3 Contributions of the thesis

We analyze the reasons for time inconsistency in bMSCs and HMSCs. Based on these analyses, we propose a solution by transforming a bMSC specification to a bMSC distance graph and provide inconsistent cycles tracing-back algorithms to identify these inconsistent traces. Based on the types of inconsistencies, we develop four policies for different cases to correct the inconsistent bMSCs.

For an HMSC, we decompose the HMSC graph into different paths. By analyzing these path consistencies, we figure out the reasons for inconsistency. For different types of paths, we propose algorithms and potential correction solutions for inconsistent HMSCs.

The contributions of the thesis are summarized as follows:

1. We analyze and categorize the causes for time inconsistency of MSCs.
2. We propose algorithms for checking and tracing back the inconsistencies of time constraints in bMSCs.
3. We develop strategies and policies to correct inconsistent MSCs.
4. We propose algorithms for checking and correcting the inconsistencies of time constraints in HMSC

1.4 Organization of the thesis

We introduce the MSC language in *Chapter 2*. The basic MSC constructs such as messages, events, instances, timers, inline expressions, and High-Level MSC are introduced. Most importantly, we present the time constraints of MSC-2000 and the time consistency concept, some related works and methodologies that are useful in consistency checking for bMSCs and HMSCs.

We describe time inconsistency and correction for bMSCs in *Chapter 3*. In this chapter, we first provide different definitions of temporal order time constraints; then we propose the checking methods to figure out the causes for inconsistency of bMSC specifications. We also propose four correction policies according to the causes for inconsistent bMSCs. Lastly, we develop checking and correction algorithms for these proposals.

The HMSC specifications inconsistency problem is discussed in *Chapter 4*. We analyze the causes for inconsistency of HMSCs. Then, the checking and correction methods are proposed. Finally, the algorithms for these methods are presented.

In *Chapter 5*, we introduce a tool developed for the purpose of checking time-inconsistency and the correction for bMSCs and HMSCs. The architecture and development issues are discussed. Two case studies are presented as well.

In *Chapter 6*, we summarize the contributions of this thesis and discuss some future work.

Chapter 2

Message Sequence Charts Language

2.1 Introduction

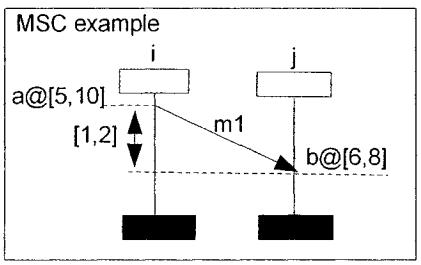
Message Sequence Charts (MSC) [10] is a graphical and textual representation of an interchange of a finite number of messages between a finite numbers of processes. MSC was initially developed by ITU-T as a companion to the Specification and Description Language (SDL) [11]. MSCs are often used in combination with SDL. MSC has gained popularity in the telecommunication field for its intuitive graphical expression and the underlining formal semantics.

The main area of application for Message Sequence Charts is as an overview specification of communication behavior of real-time systems, in particular telecommunication switching systems. Message Sequence Charts may be used for requirement specification, simulation and validation, test-case specification and documentation.

The first recommendation for MSC was approved in 1993. It contained textual syntax definitions, but a formal semantics was missing. In the second recommendation of MSC'96, the formal semantics based on process algebra was defined. Moreover, MSC'96 standard has introduced the high level MSC (HMSC) where bMSCs can be composed

using a set of operators. The latest version MSC-2000 standard adds timing constraints and data.

A simple basic MSC specification in both graphical (MSC/GR) and textual (MSC/PR) format is shown in Figure 2.1.




```
mscdocument example;
  inst i;
  inst j;
  msg m1;
  language C;
  msc bMSC;
 i: instance;
  label a out m1 to j time@[5,10];
 endinstance;
 j: instance;
  label b in m1 from i time[1,2] a, time@[6,8] ;
 endinstance;
endmsc;
```

Figure 2.1 An MSC specification in MSC/GR and MSC/PR formats

On the one hand, the MSC/GR is an intuitive description. It is easy for system analysts, designers and testers to specify behavior requirements, designs, test cases and understand them. On the other hand, the MSC/PR is a textual alternative used to describe a system, and it is an input for automatic tools for manipulation. In this thesis, we use MSC/GR for illustration; while developing the tool MSCTICC, we use MSC/PR as an input. The simplified syntax used in this present thesis is described in Appendix A.

In following sections of this chapter, we introduce the MSC language and time consistency concept.

2.2 Basic MSC

Basic Message Sequence Charts (bMSC) is the core part of Message Sequence Charts. The bMSC is used to express the behavioral requirements in terms of scenarios that the system is required to exhibit.

A bMSC is concerned with communications and local actions only. The basic concepts in bMSC are instance creations and terminations, environment, timer handling, message events, conditions, inline expressions, etc. The basic MSC concepts are described in Figure 2.2.

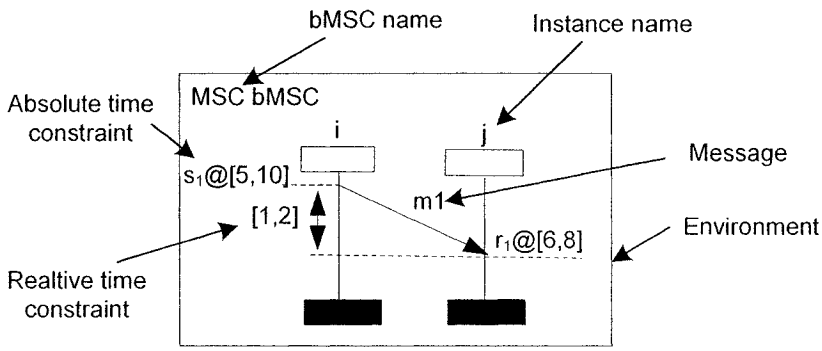


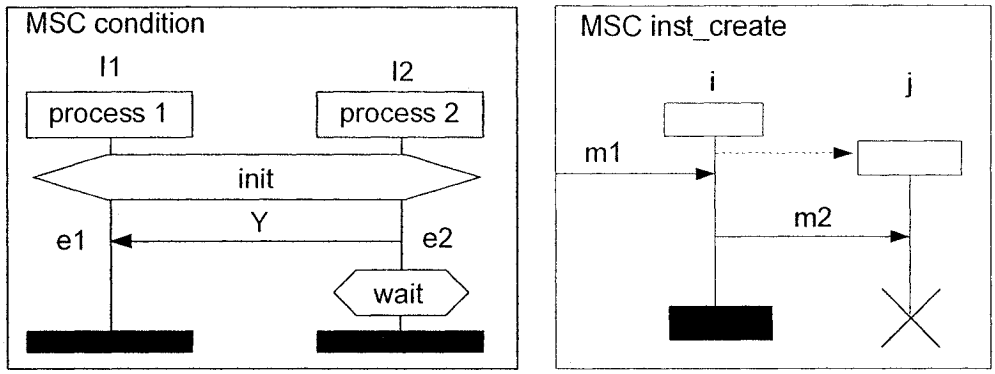
Figure 2.2 Basic MSC concepts

The basic constructs of bMSCs are instances and messages that describe the communication events. MSC instance plays an important role in the MSC specifications. An instance may be a subsystem, a process, or a thread. An instance can even be a use case [12]. An instance is represented by an axis that is delimited by a start and end symbol. The head symbol determines the start of the description of an instance within the MSC. The end symbol stands for the end of the description of the MSCs. The time is

running from top to bottom along an instance axis. Events in one instance are totally ordered according to their positions from the start to the end symbols on the instance axis. A message has a send event and a receive event on both ends. The send and the receive events of the message are called message pair events. The send and receive events provide a total order assumption between these two events. Both events of a message can happen in an instance or environment. A message is represented by an arrow from its output to input. Every event has a unique event name in the bMSCs. Every message has a unique message name attached on that arrow.

In a bMSC, the system environment is represented by a frame symbol that forms a boundary. There is no assumed ordering among the environment between the sending and receiving events. However, it is assumed that the environment behaves according to the bMSC specification.

In addition to exchanges of messages, a bMSC may contain conditions that describe the state of the instance, actions, timers, and instance instantiation and termination. Conditions are used to describe the state in the bMSC. The main purpose is for documentation. There are three types of conditions: global, partial, and local. A global condition is used to describe the global system state of all instances in the bMSC. A partial condition is used to describe the partial state of some instances in the bMSC. A local condition describes a private state of the specific instance. In Figure 2.3(a), the *init* is a global condition to indicate the current state of instance *I1* and *I2*. The *wait* is a local condition to express the state of the instance *I2* after the message *Y* is sent.



(a) (b)

Figure 2.3 bMSC conditions (a) and Instance creation (b)

In bMSCs, an instance may be created by another instance. No event can take place in a created instance before its creation. An instance stop is the counterpart of the instance creation. When it stops, an instance terminates its lifetime. Figure 2.3(b) shows that instance *i* creates instance *j*. After sending message *m2*, the instance *j* terminates its execution.

Timer events are used to define certain time related requirements. There are three types of timer operator: starttimer, stoptimer, and timeout. A starttimer is used to set with duration to initiate a timer. When the time specified by the duration expires, the timer encounters a timeout. However, before timeout happens, this timer can be stopped or reset. In Figure 2.4, timer *T1* with duration 10 is started in instance *I1* and then it is stopped. In instance *I2*, timer *T2* is started, and it is stopped after the default time duration expires.

An action is an atomic internal event in one instance. In Figure 2.4, the two internal actions are also expressed in the bMSC. One is the formal action of assigning *0* to variable *a*, the other is the informal action that express *'processing'* in a string.

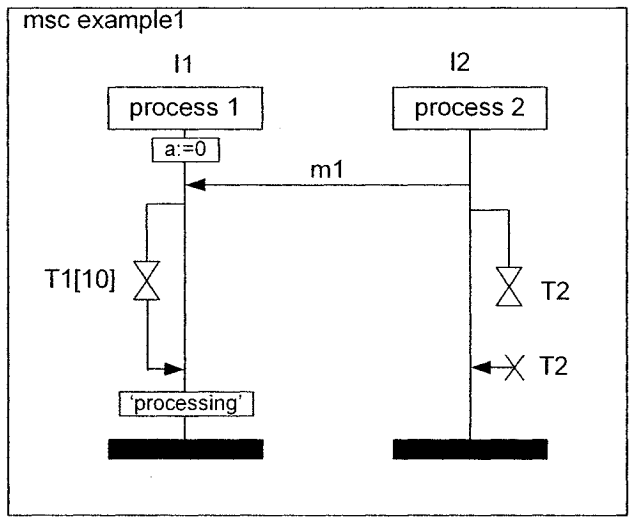


Figure 2.4 bMSC timer events and actions

An important concept in MSCs is the ordering of events. It is the basis of the temporal ordering and timing concept. The ordering of events in MSC is determined by three rules. The first rule is that events on the same instance are ordered according to their positions along the instance axis. The second rule is that a message sending event always precedes its corresponding receiving events. The third rule is that the events happening on the created instance are behind its creating event. *Coregions* are introduced to relax the order on some parts of the instance axis. Within the *coregions*, the events are not ordered.

Inline expressions define event compositions in a bMSC. There are five inline operators: parallel, alternative composition, iteration, exception, and optional region. The operator *par* defines the parallel execution of MSC sections. A parallel inline expression defines a

parallel behavior in a bMSC where no ordering is defined between events in different sections. The operator *alt* defines alternative scenarios. An alternative inline expression defines alternative execution of a behavior of a bMSC. The operator *loop* defines an iteration scenario. An iteration inline expression defines iteration execution of a section of a bMSC. Events in the iteration area will be executed many times (0 to infinity). The operator *exc* stands for the exception cases in the MSC. An exception inline expression defines exceptional behaviors in a bMSC. Either the events in the exceptional area will be executed, or the rest of the MSC specification will be executed. The operator *opt* is an optional section of an MSC specification. An optional inline expression defines an operational behavior in a bMSC. The inline expressions are described in Figure 2.5.

2.3 High Level MSC

High-Level Message Sequence Charts (HMSC) describes graphically how the basic MSCs are combined. HMSC hides low-level details and improves the readability of a system. An HMSC is a directed graph where each node is a start symbol, an end symbol, an MSC reference, a condition, a connection point, or a parallel frame [10]. There is only one start symbol; however, there could be zero to many end symbols in an HMSC. The MSC reference may refer to a bMSC or an HMSC. The condition indicates a global system state or guard and imposes restrictions on the MSCs that are referenced in the HMSC. The parallel frame contains several HMSCs that execute in parallel. We consider that the MSC references of HMSCs in the parallel frame are interleaved. The connection points are used to improve the layout of HMSCs. However, the connection points do not

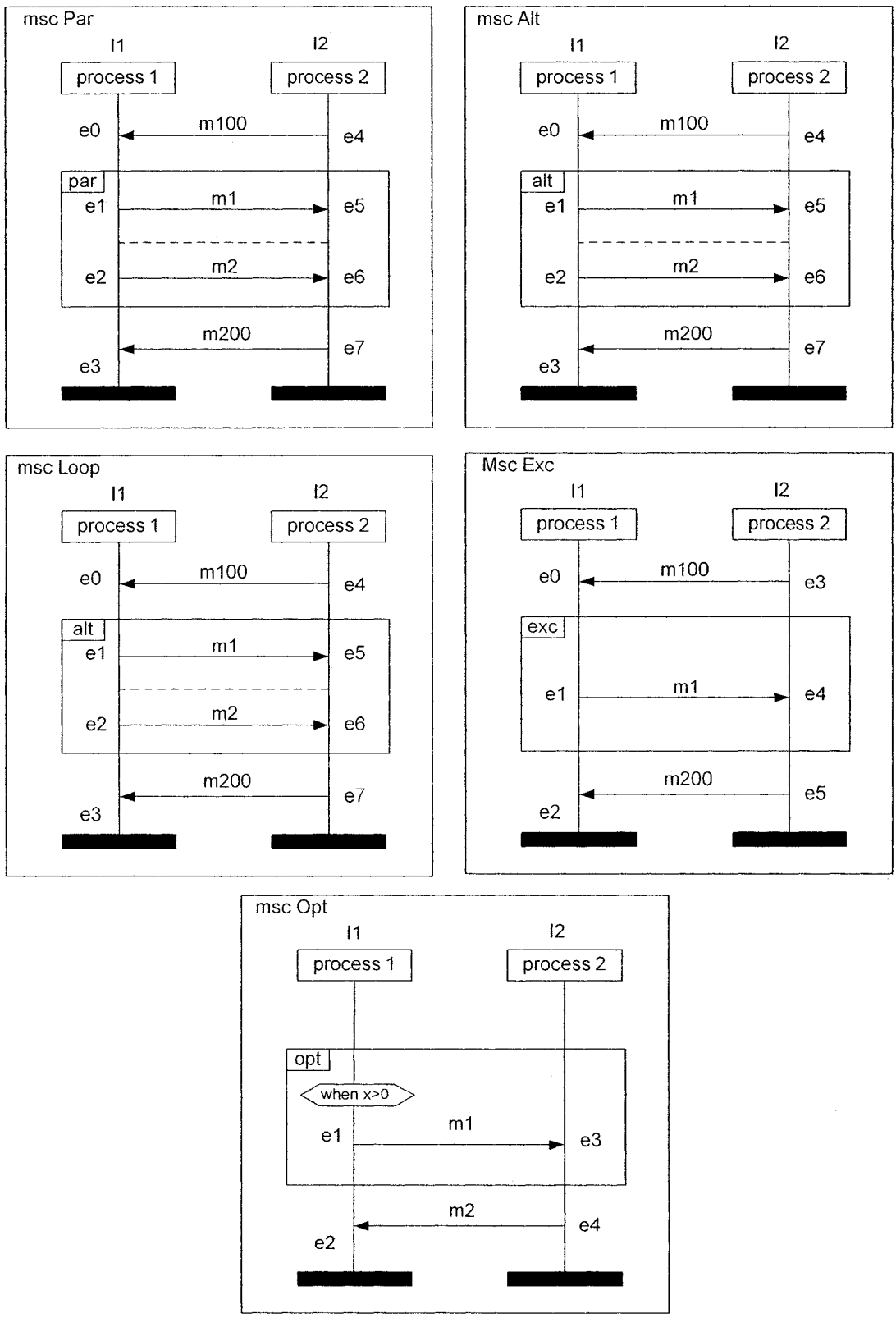


Figure 2.5 bMSC Inline expressions

affect the semantics of HMSCs. In addition, HMSCs can describe systems in hierarchical manner by combining multiple HMSCs within a single HMSC.

HMSCs are viewed as roadmaps composed by bMSCs. HMSC provides three operators that are sequential (*seq*), alternative (*alt*), and iterative (*loop*) operators to structure referenced MSCs. They are similar to the corresponding operators - inline expressions of bMSC.

A sequential HMSC operator defines the sequential execution of several bMSCs. The bMSCs will be executed one by one in the order specified by the HMSC. An alternative HMSC operator defines alternative executions of bMSCs. Only one of the alternative bMSCs will be executed for each execution trace. The iteration HMSC operator defines the iteration execution of bMSCs. The parallel HMSC operator defines the parallel execution of bMSCs or HMSCs.

In an HMSC, nodes are connected by flow lines. The flow lines indicate the possible execution sequence of the nodes. Two nodes connected by a line are concatenated process by process. This concatenation means that all the events in the first node are not guaranteed to occur before the events in the second node if the two events are not in one process. This concatenation is called weak concatenation. A node may have more than one outgoing flow line meaning that the successors of the node are alternatives. A cycle connecting several nodes expresses a repetition. In this way, the non-determinism and infinite behavior can be expressed in HMSCs.

An HMSC is shown in Figure 2.6. The symbols▽, △ and ○ represent a *start* symbol, an *end* symbol and a *connection point*, respectively. *S1*, *S2*, *S3*, and *S4* are bMSCs or HMSCs. The control flow goes to *S1* and sequentially goes to *S2*. An alternative operator following *S2* and *connection point* directs the control flow to *S3* or *S4*. There is an iterative operator, specifying a free loop from *S3* to *connection point*. The end symbol is interpreted after executing *S4*.

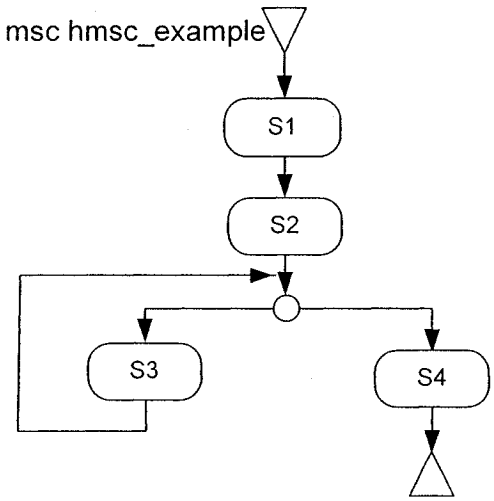


Figure 2.6 An HMSC example

2.4 Time concepts

The MSC-2000 specification introduces some new concepts that make the MSCs more powerful and flexible for specifying systems. These new concepts include data, time, control flow and object orientation in MSC documents. We are more interested in the time constraints added in the MSCs. The time constraints are used to support the notion of qualified time for the description of a real-time system with a precise meaning of the sequence of events in time.

According to MSC semantics, MSC events such as message input/output, timer events and actions are instantaneous, i.e. they do not consume time.

The MSC-2000 standard specifies time constraints: relative time constraint and absolute time constraints. A time constraint is an interval with minimum (lower) and maximum (upper) bounds in a time domain. Relative time constraint defines the minimum and maximum bound for the delay between two events. It represents the relative timing relation between two events. Absolute time constraint represents an absolute time interval at which an event occurs. The absolute time represents the global clock value of the specified system. To distinguish absolute and relative time constraints, an @ sign is used before an absolute time constraint, and an & sign is used before a relative time constraint.

Figure 2.7 shows a basic MSC with time constraints. At the time point interval between 10 and 12 time units, instance *j* sends a *VERIFY* message to instance *k*. Then instance *k* sends an *OK* message back with the minimum delay of 2 and the maximum delay of 4 time units.

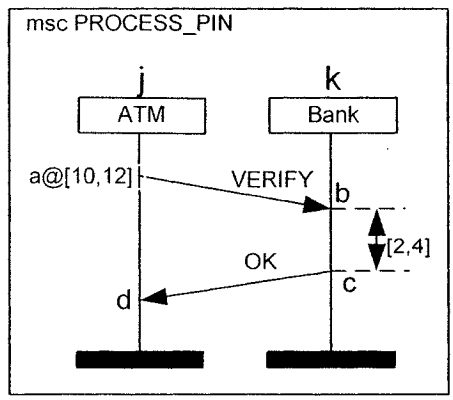


Figure 2.7 A bMSC with time constraints

Another concept is time measurement. Measurement is used to express the delay of two events that can be expressed as *&t*, or to measure the absolute time of an occurrence of an event, denoted as *@t*.

Time constraints in MSC-2000 are more general and convenient than timers because timers can only specify timing requirements in an instance while time constraints can be used to specify timing requirements in different instances. For instance, to specify a delay between two events, a timer has to be set immediately after the first event, and then a time-out event has to be placed immediately after 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.

2.5 Semantics of timed MSC

In the MSC-2000 specifications, the semantics of a timed MSC is mentioned briefly as event traces with special time events between normal events [10]. For example, a trace for the MSC in Figure 2.7 is $\{a, t_1, b, t_2, c, t_3, d\}$, where t_i represents the time intervals. The triple $\{b, t_2, c\}$ means, for example, that event c occurs after the event b in time t_2. If there is no time interval between two normal events, it means that the two events may occur simultaneously. A trace always begins with a normal event. Figure 2.7 of MSC can also be expressed as one trace $\{a, any\ time, b, 2, c, any\ time, d\}$.

The drawback of the informal semantics of the MSC-2000 specification is that the event traces do not mention how an MSC corresponds to traces, and the information about the absolute time constraints are lost. To overcome the drawback, it is proposed for a model of partial order with time constraints for MSC specification in [27].

We extend the concept in the two documents above and the time measurement concept to include the absolute and relative time constraints in the time trace definition as follows.

A timed trace of an MSC is defined as a sequence of timed events $\{(e_1@t_1), \&T_1, (e_2@t_2)...\&T_n, (e_n@t_n)\}$ *with* $\bigcup_{i=1}^{n}\{e_i\} = E$, $t_i \in Time$, $T_i = |t_i - t_j|$, *such that for all instances i and j,* $0 \le i \le n$, $0 \le j \le n$:

- *if* $e_i \le e_j$, *then* $t_i \le t_j$;
- $t_i \in D(e_i)$;
- $|t_i - t_j| \in T(e_i, e_j)$;
- t_i *must grow over all bounds in an infinite sequence of timed events.*

We define the set of traces as $Tr(l_p) = \{w \mid w \text{ is a trace of } l_p\}$.

For example, in MSC example (see Figure 2.1), the timed MSC has a set of traces, such as $\{(a@5), \&2, (b@7)\}$, $\{(a@5), \&1, (b@6)\}$ and so on.

For another example, in *MSC PROCESS_PIN* in Figure 2.7, a possible timed trace is $\{(a@10), \&t_1, b, \&2, c, \&t_2, d\}$, where t_1 stands for the time interval of message pairs *a*

and b, and t_2 stands for the time interval of message pairs c and d. Many other traces may satisfy the definition of timed trace in this MSC.

A timed labeled partially ordered set (timed lposet) is used to define the semantics for a timed MSC. An lposet defines causal orders between events. Sequential (*seq*), alternative (*alt*) and iterative (*loop*) compositions are also well defined to express the semantics of HMSC.

An HMSC is a directed graph that is composed of a set of nodes, and every node is a bMSC. The semantics of an HMSC is defined as a set of timed lposet [27].

The semantics of timed MSC provide a foundation for the time consistency of MSC-2000 specifications.

2.6 Time consistency of Basic MSCs

The problem of time consistency of MSCs has been investigated by several research groups [2][5]. In the latest research work done by Tong Zheng [26], the time consistency of absolute and relative time constraints in bMSCs is analyzed. Key points of this approach are the following: (i) the timed bMSCs specification is transformed into a simple temporal problem (STP) represented by a directed graph and is further transformed into a distance graph; (ii) then the Floyd-Warshall Algorithm is applied to

compute the shortest paths of all pairs in the distance graph. According to [4], if there are no cycles with negative costs in the graph of shortest paths, the bMSC is consistent.

A bMSC can be modeled as a directed constraint graph $G = (V, E)$. The graph G consists of a set of vertices V and a set of edges E, such that each edge E is a connection between a pair of vertices in V. The nodes are events, and there exists an edge from e_i to e_j if e_i happens before e_j. Relative time constraints can be labeled above the edges of two causally ordered events. A special event e_0 is added to the bMSC graph and is defined as the starting event that happens before any other events; therefore, it is expressed as time zero. In this way, any absolute time constraint can be translated into a relative time constraint between the event and event e_0. The number of the vertices is written as $|V|$.

Let *Time* represent time domain, e.g. non-negative real or integer numbers. Let *T(Time)* be a set of time intervals that can be open denoted as "(" or ")" , or closed denoted as "[" or "]" . We treat time domain as the domain of non-negative real numbers, then the time constraint is $[1, \infty)$ or $(0, \infty)$. For the events that have no absolute time constraints, we can express the absolute time constraints of these events with $[1, \infty)$ or $(0, \infty)$, because the e_0 happens before any events. Here, time $[1, \infty)$ is equal to time $(0, \infty)$ because algorithms mentioned in this thesis assume a discrete time domain. In a bMSC specification, if one event occurs before another event and if there is no relative time constraint specified between the two events, we can express the causal order as $[1, \infty)$. If two events have no precedent relationship with each other, we express the time constraint as $(-\infty, \infty)$ between the two events, which means the two events can happen at any time in any order.

A bMSC specifies a directed constraint graph that could be either a sparse graph (with few edges), or a dense graph (with many edges). However, in order to apply algorithms described in the following discussion, we need to transform the directed constraint graph into a complete graph [21] in which all nodes (events) are connected with each other. This is done by a classical transitive closure operation.

A complete directed constraint graph and a distance graph corresponding to the *MSC M1* are shown in Figure 2.8. For example, event s_1 and event s_2 are unordered; therefore, the timed constraint between s_1 and s_2 is (-∞,∞). It is the same case for the time constraint r_1 and r_2. Event s_2 time constraint is not specified; thus, the time constraint between e_0 and s_2 is &[1, ∞).

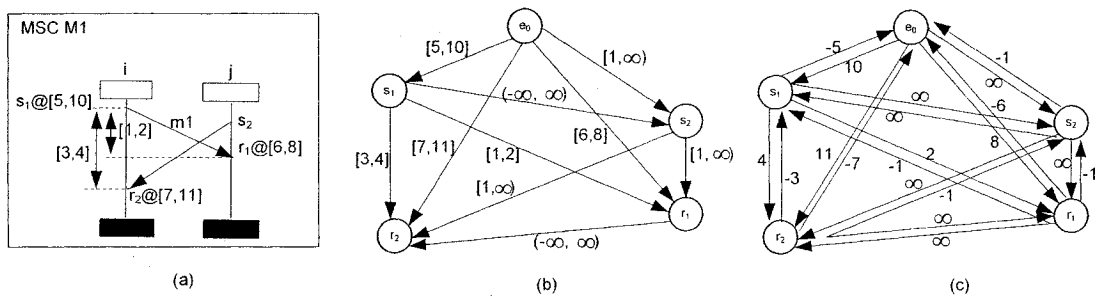


Figure 2.8 A bMSC complete directed constraint graph and its distance graph

The simple temporal problem is to decide if each node can be assigned to a value such that the time constraints between nodes are all satisfied. If such assignments exist, the associated bMSC is consistent. As discussed in [8], the consistency can be decided by the Floyd-Warshall Algorithm to compute all pairs of 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, a unique interval for each event be obtained,

in which each value can be assigned to the event so that the graph (and the corresponding bMSC) is consistent. Such an interval is defined as a reduced absolute time constraint. In the same way, a unique interval between every two events can be obtained, and the interval is defined as a reduced relative time constraint.

The Floyd-Warshall all-pairs-shortest-path algorithm has the complexity $O(n^3)$ [20] where n is the event number. The algorithm is shown in Figure 2.9. The main idea is that each successive assignment needs to be checked against previous assignments and is guaranteed to remain unaltered.

Assume that the edge, $i \rightarrow j$ is labeled by an interval, $[a_{ij}, b_{ij}]$.

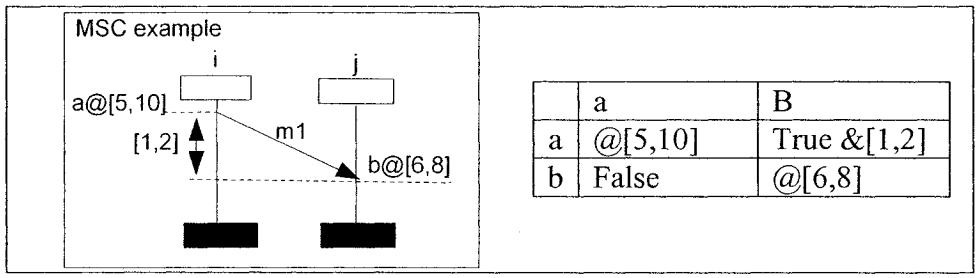
All-pairs-shortest-paths algorithm

1. *for i:=1 to n do* $d_{ii} = 0$
2. *for i,j:=1 to n do* $d_{ij}=a_{ij}$
3. *for k:=1 to n do*
4. *for i,j:=1 to n do*
5. $d_{ij} = min\{d_{ij}, d_{ik}+d_{kj}\}$;

Figure 2.9 The Floyd-Warshall Algorithm

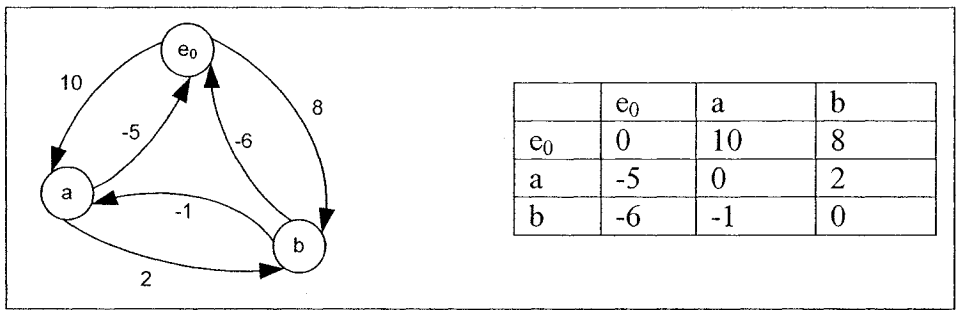
An MSC and its corresponding distance graph are shown in Figures 2.10 and 2.11 respectively. In Figure 2.10, the *MSC example* is expressed as an enhanced-event-order table with time constraints. The bMSC is transformed into a distance graph and matrix as

shown in Figure 2.11. The Floyd-Warshall Algorithm is applied on the distance graph matrix to compute the all-pair-shortest-path. The result is depicted in Figure 2.12. The reduced absolute time constraint and the reduced relative time constraint are shown in Figures 2.13. Event *a* has the absolute time constraint @[5, 7]; event *b* has the reduced absolute time constraints @[6, 8]. The reduced relative time constraints are also obtained. The reduced relative time constraint between *a* and *b* is &[1, 2].



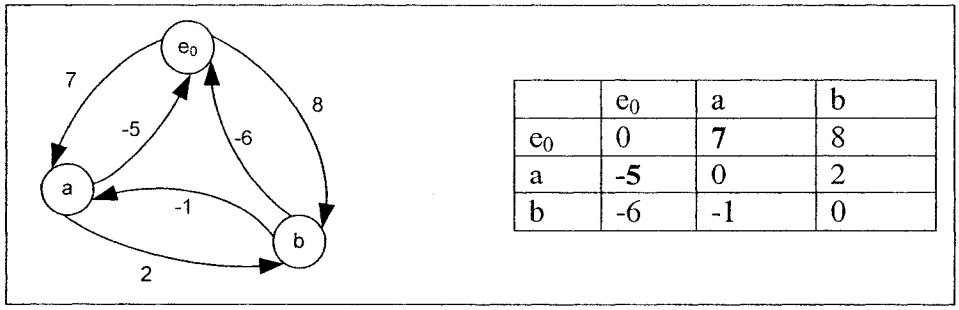
	a	B
a	@[5,10]	True &[1,2]
b	False	@[6,8]

Figure 2.10 A bMSC and its event-order table with time constraints



	e_0	a	b
e_0	0	10	8
a	-5	0	2
b	-6	-1	0

Figure 2.11 The distance graph and its matrix



	e_0	a	b
e_0	0	**7**	8
a	**-5**	0	2
b	-6	-1	0

Figure 2.12 The distance graph and its matrix after applying Floyd-Warshall algorithm

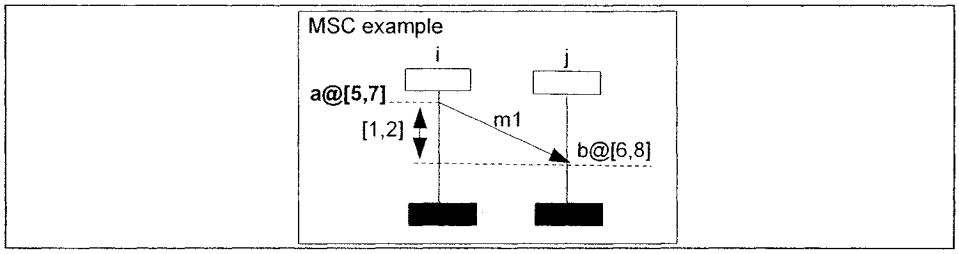


Figure 2.13 The reduced time constraints

2.7 Time consistency of High Level MSCs

An HMSC can be expressed as a directed graph $G = (S,D,L)$, where S is a finite set of nodes; $D \subseteq S \times S$ is the set of directed edges; L is a function that maps each node in S to a bMSC. An HMSC can have only one start node that has no incoming edges. An HMSC may have several or no end nodes.

An HMSC can be decomposed into finite or infinite paths. Along each path, bMSCs are composed sequentially. The consistency of an HMSC is defined in terms of consistency of its paths.

A loop in an HMSC may generate an infinite number of paths. It is not possible to check all the paths because of the infinitive number of paths. Therefore, we check the consistency of an HMSC by checking its simple paths and the type of the paths.

A **simple path** of an HMSC is defined as a finite or an infinite sequence of nodes $s_0 s_1 \ldots s_n \ldots$, in which s_0 is the start node, $s_i \neq s_j$, if $i \neq j$, and $(s_i, s_{i+1}) \in D$, $i \geq 0$. A simple path end with an end node s_n, or ends with a loop [27].

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

An HMSC's semantics are based on a set of labeled partial order sets (lposet). The time consistency of the HMSC is defined in terms of these lposets. To facilitate the development of algorithms, we define the consistency of an HMSC based on the combination of the consistencies of paths in the HMSC.

In an HMSC, bMSCs can be combined sequentially along paths, alternatively by branches, or repeatedly by cycles. We define sequential composition (*seq*), alternative composition (*alt*), and iteration (*loop*) of bMSCs in terms of their corresponding operations on labeled partially ordered sets (lposets) [26].

For two *MSCs A* and *B*, we define the following mappings:

- $M[A\ seq\ B] = M[A] \cdot M[B]$, where "$\cdot$" is a concatenation meaning *A* and *B* are sequentially connected;
- $M[A\ alt\ B] = M[A] \# M[B]$, where "#" is an alternation meaning *A* and *B* are alternative;
- $M[loop<i,j>A] = M[A^i] \# M[A^{i+1}] \# .... \# M[A^j]$, where A^i means *A* can repeated sequentially i times.

M represents a Mapping function. For $k>0$, we define $M[A^k] = M[A] \cdot M[A^{k-1}]$, $M[A^0] = \varepsilon$.

The HMSC consistency can be classified as strong consistency, weak consistency and inconsistency according to the result of the HMSC path consistency.

An HMSC is strongly consistent if and only if all its paths are consistent. If some paths are consistent and some are not, then the HMSC is weakly consistent. If all the paths are not consistent, the HMSC is called inconsistent. The formal definition and checking algorithms are proposed in [26] and implemented [23].

The main steps of checking techniques are: first, apply the Deep-first-search Algorithm [21] to traverse the HMSC to determine all the simple paths. In each simple path, all bMSCs are composed to form one bMSC. Then, the algorithms used to check time consistency of bMSC are applied.

For example, the *HMSC H1* in Figure 2.14 has two *bMSCs* L_3 and L_4 connected sequentially. The HMSC can be represented as M[L_3 *seq* L_4] = $\{L_3L_4\}$; moreover, the *HMSC H1* can be decomposed into only one path $\{L_3L_4\}$. Based on the simple path $\{L_3L_4\}$, we compose one bMSC as described in Figure 2.14(d). We apply the checking algorithm for bMSC time consistency as described in Section 2.6. We conclude that the bMSC is consistent. Because the only path is consistent, we claim the HMSC is strongly consistent.

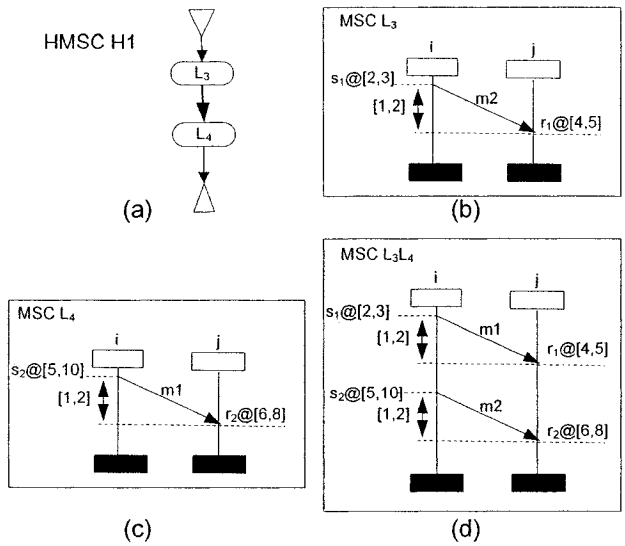


Figure 2.14 A strongly consistent HMSC

Let us consider another example. Figure 2.15 depicts an HMSC specification with alternative composition. The HMSC begins and branches alternatively into two *bMSCs* L_1 and L_2, and then finishes with an end node. The *HMSC H2* is represented as

- $M[L_1 \# L_2] = \{ L_1, L_2 \}$.

Therefore, the *HMSC H2* can be decomposed into two paths $\{L_1\}$ and $\{L_2\}$. The *bMSC* L_1 and L_2 are checked by the algorithms as described in *Section 2.6*. We conclude that *bMSC* L_1 is consistent, but *bMSC* L_2 is not consistent. Thus, the path $\{L_1\}$ is consistent, but the path $\{L_2\}$ is not. In combination of the consistencies of the two paths, the HMSC is weakly consistent.

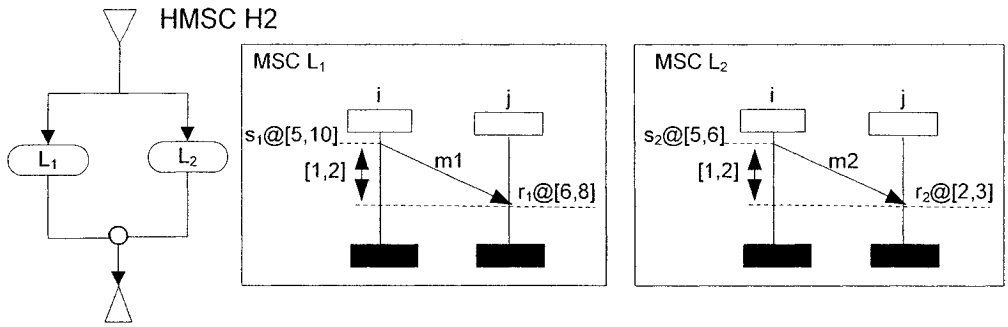


Figure 2.15 A weakly consistent HMSC with an alternative composition

In Figure 2.16, *HMSC H3* is an MSC with a path with a loop. The HMSC can be represented by:

- M [(*loop*<0,∞>*M*)] = {*M, MM, MMM, ...*}.

There is a loop in the simple path. The bMSC *M* will repeat infinitely. After MSC *M* is executed three times, the absolute time constraint of sending message *m1* will be violated. Because the upper bound of the absolute time constraints are not infinite and the path will not be consistent after it goes loop unfolding three times *MMM*, it is said to be inconsistent.

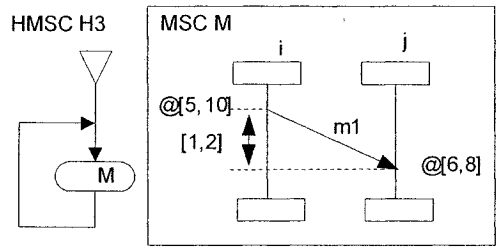


Figure 2.16 An inconsistent HMSC with a loop

2.8 Conclusion

MSC can be specified in a graphical and a textual format. MSC-2000 includes all the constructs in MSC-96, and some new constructs such as data, time, control flow, and guard conditions. Time constraints in MSC-2000 are more general and convenient than timers because time constraints can be used to specify timing requirements in different instances. Time constraints in an instance can be seen as high level requirements, while timers can be seen as low level implementations of the requirements.

We provide a definition of time trace for semantics of timed MSC. The semantics of an HMSC is based on a set of timed lposets of bMSCs.

Simple temporal problem (STP) techniques are used in transforming a bMSC into a directed constraint graph and further into a distance graph. Then All-pair-shortest-path algorithm is used to check the consistency of the bMSC.

An HMSC time consistency can be based on the consistency of paths in the HMSC. An HMSC can be decomposed into finite simple paths. A simple path can be sequentially composed by bMSCs and can form an lposet. Then the consistency checking algorithm of bMSCs can be applied on this simple path. Based on the consistency of paths in an HMSC, we categorize the time consistencies as strongly consistent if all the paths are consistent, weakly consistent if at least one path is consistent, or inconsistent if no paths are consistent.

After we introduce the MSC-2000 language and time consistency, we will explore the time inconsistency and correction methods of bMSCs in *Chapter 3*, and we will discuss the time inconsistency and correction strategies of HMSCs in *Chapter 4*.

Chapter 3

Time inconsistency analysis and correction for bMSC specifications

3.1 Introduction

Time consistency is an important requirement in real-time systems, distributed systems, and communication systems. Time consistency exists in basic-MSCs (bMSC) and High level-MSCs (HMSC).

A timed MSC is consistent if and only if a timed trace exists [26]. For example, in Figure 2.1, we can find a trace *{a@[5], &1, b@[6]}*; therefore, we claim the MSC example is consistent.

An MSC specification is inconsistent if there are conflicts among different time constraints and we cannot find a timed trace. We categorize the time inconsistencies according to the causes. The types of time inconsistency that may occur in bMSCs are as follows.

- First, absolute time constraints do not conform to the causal order of events. For example, in the *MSC N1* shown in Figure 3.1(a), event e_1 happens before event e_2 as indicated by the arrow from e_1 to e_2. However, the absolute time constraint for event e_1 and e_2 are @[6, 8] and @[1, 2] respectively. The order defined by time constraints contradicts the causal order between events e_1 and e_2. Therefore, we cannot find a timed trace for the *MSC N1*.

- Second, absolute and relative time constraints are not consistent with each other. In Figure 3.1(b), the absolute time constraint of event e_1 is @[2, 2] and the relative time constraint between e_1 and e_2 is [1, 2]. The two time constraints imply that the absolute time constraint of e_2 should be @[3, 4]. However, the calculated time constraint of e_2 @[3, 4] contradicts with the specified absolute time constraints e_2 @[6, 8]. Therefore, we cannot find a timed trace for *MSC N2*.
- Last, relative time constraints are not consistent with each other. For example, an inconsistency between relative time constraints is shown in Figure 3.1(c). Events e_1, e_2 and e_4 occur in that order. The relative time constraint between e_1 and e_2 is 1, and the relative time constraint between e_2 and e_4 is 2. The two specified relative time constraints determine that the delay between e_1 and e_4 should be 3, which contradicts the value of 2 for the relative time constraint specified in the *MSC N3*. There is no timed trace in the MSC.

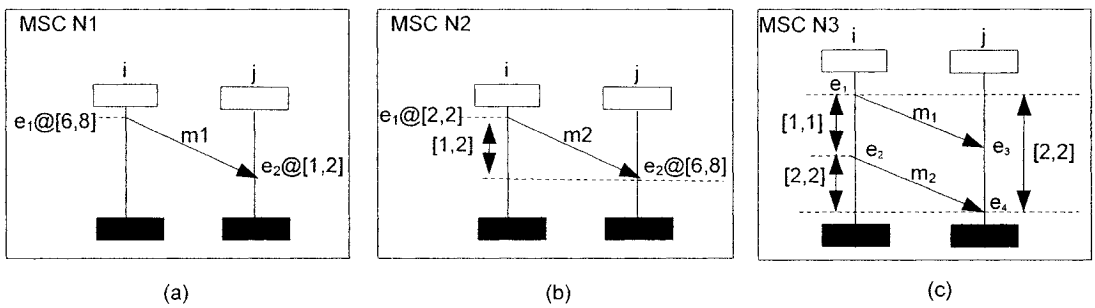


Figure 3.1 Time inconsistencies in bMSCs

The problem with checking time consistency has been investigated by different groups [2, 3, 15, 27, 25]. However, in undertaking these investigations, the research groups did not intend to explain, and their results did not show the causes of the inconsistency of the

MSC. There is no research to show which MSC traces are not consistent, nor how to correct these inconsistent traces. In this chapter, we try to propose some solutions to solve the problem of inconsistencies of bMSCs.

The rest of the chapter is organized as follows. In *Section 3.2*, we define different partial orders and time constraints to form the basis for the further discussion of time inconsistency. In *Section 3.3*, we illustrate the criteria used in checking into and tracking back to the causes for the inconsistencies of bMSCs. In *Section 3.4*, we propose some correction strategies for the inconsistent bMSCs. In *Section 3.5*, we provide several algorithms for the checking, tracing back, and correction procedures. We conclude this chapter in *Section 3.6*.

3.2 Directed orders vs. deduced orders

MSC diagrams are graphical representations of the scenarios or the executions of communication systems. The representations are formally defined in the MSC-2000 specifications [10].

In our research, we have considered bMSCs containing instances and messages, and HMSCs composed of bMSCs only, so that we can focus on the most important aspects of time consistency.

A specification designer can specify the time constraint between two events e_i and e_j by drawing it on the bMSC according to MSC-2000 rules. This is a directed order or input order time constraint $V(e_i,e_j)$. It reflects the directed causal time constraint of events in a graphical representation of the MSC.

Between e_i and e_j there could also exist a sequence of other events $\{e_k, e_l, e_m, ...\}$ defining other traces starting from e_i and leading to e_j. Collectively, these other traces of time constraints, defined by the succession of intermediate events, are called a deduced order or calculated order time constraint between e_i and e_j. The intermediate events $\{e_k, e_l, e_m, ...\}$ also have their own time constraints defined by the designer. The deduced order time constraints $I(e_i, e_j)$ between e_i and e_j are defined as the sum of time constraints related to intermediate events $\{e_k, e_l, e_m, ...\}$.

Sometimes, the MSC time requirement cannot satisfy both the directed order time constraint $V(e_i, e_j)$ and the deduced order time constraints $I(e_i, e_j)$. This is a typical case of a time inconsistency.

We extend timed lposet [26] to more accurately express different time orders. An lposet defines causal orders between events. A bMSC can be defined as a tuple with directed order events and deduced order time constraints as follows.

An MSC M is a 8-tuple (P, A, E, <, l, L, B, T), where

- *P is a set of instances or processes,*

- *A is a set of labels,*
- *E is a set of events,*
- *$\leq \subseteq E \times E$ is a partial order between events on E; $\leq$ is a union of directed order (<) and deduced order($<^*$),*
- *l: $E \rightarrow A$ is a labeling function that associates an event to a label,*
- *L: $E \rightarrow P$ is a mapping function that assigns an MSC instance to each event,*
- *B: $E \rightarrow T(Time)$ is a function that associates an event with a time interval within which the event must occur; this time interval is called the absolute time constraint,*
- *T: $E \times E \rightarrow T(Time)$ is a function that associates with a pair of events a time interval that defines the delay between the two events; $T = V \cup I$, where V stands for* ***directed order time constraint,*** *and I stands for* ***deduced order time constraint*** *.*

Definition 1: *Directed order, denoted as <, is simply the union of the orders, i.e., $< = <_c \cup \{<_{Pi} | \forall_{Pi \in P}\} \cup <<_i \cup <<_a$, where $<_c$, $<_{Pi}$, and $<_a$ are defined as:*

- *e and f are sending and receiving events of the same message. In this case, the events e and f are said to be a 'message pair'; or one says that event e directly causes event f. Let $<_c = \{(e,f) \mid e<f \wedge L(e) \neq L(f)\}$ denote the communication directed order between a sending event and a receiving event.*
- *e and f belong to the same instance with e appearing above f in the instance life-line. Then, e and f are in the local directed order. Let $E_{pi} = \{e \mid e \in E \wedge L(e)=P_i\}$. Let $<_{Pi} = < \cap (E_{pi} \times E_{pi})$ denote the local directed order of instance P_i [16].*

- *e indirectly causes f. e and f are in different instances, and e and f are not a matching pair but there is a relative time constraint specified between e and f. Let* $<<_i = \{(e,f) \mid e << f \wedge L(e) \neq L(f)\}$ *denote the indirect causal order, where* $e<<f$ *stands for that e indirectly causes f.*
- *f is an event with absolute time constraint, and e stands for the absolute time zero. Let* $<_a = \{(e,f) \mid e<f\}$ *denote f happens after e. Then, e and f are called absolute directed order.*

Directed order (denoted by <) is a direct causal relation between two events. For example, considering the *MSC M1* of Figure 3.2, we have $E=\{s_1, r_1, s_2, r_2\}$, $P=\{i, j, k\}$, $<_c=\{(s_1,r_1), (s_2,r_2)\}$, $<_i = <_k = \emptyset$, and $<_j = \{(r_1,s_2)\}$. The directed order < reflects the way the MSC is depicted. The directed order time constraints are $V(e_0,s_1)=[5,10]$, $V(e_0,r_1)=[6,8]$, $V(e_0,s_2)=[9,10]$, $V(e_0,r_2)=[11,13]$, $V(s_1,r_1)=[1,2]$, $V(r_1,s_2)=[3,4]$, $V(s_2,r_2)=[2,3]$, $V(s_1,r_2) = [5,10]$ where e_0 stands for the special event absolute time *zero*. The directed order time constraint of message pair s_1 and r_1 is $V(s_1,r_1)=[1,2]$, and the directed order time constraint of message pair s_2 and r_2 is $V(s_2,r_2)=[1,2]$. The directed order < is depicted in Figure 3.2(b). The local directed order time constraint between r_1 and s_2 is $V(r_1,s_2)=[3,4]$. The indirect causal order time constraint between s_1 and r_2 is $V(s_1,r_2) = [5,10]$. The absolute directed order time constraints are $V(e_0,s_1)=[5,10]$, $V(e_0,r_1)=[6,8]$, $V(e_0,s_2)=[9,10]$, and $V(e_0,r_2)=[11,13]$.

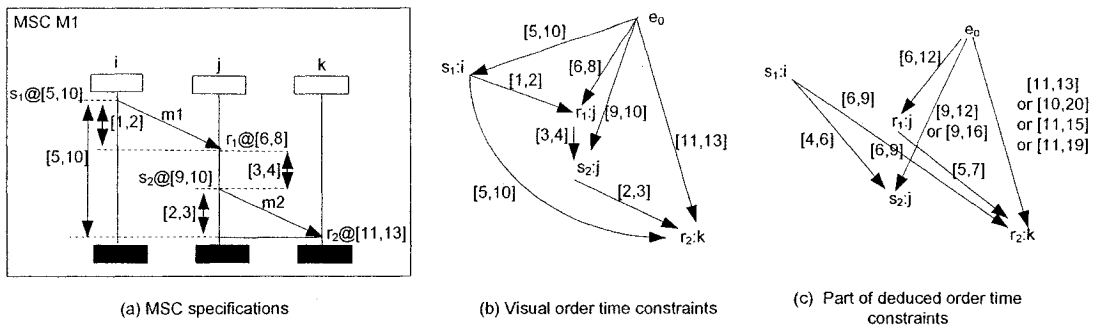


Figure 3.2 bMSC *M1*, its directed order and deduced order time constraints

Deduced Order is the cycle free relation and is a transitive closure of the directed order < without the inputted directed orders. Therefore, The deduced order is calculated from the existing directed order. The formal definition is as follows.

Definition 2 *A deduced order, denoted as* $<^*$*, is defined as a partial order existing between two events,* $e_i <^* e_k = \{(e_i, e_k) \mid (e_i < e_{i+1})$ *and* $(e_{i+1} < e_{i+2})$*...and* $(e_{i+n} < e_k)\}$ *where* $e_i, e_{i+1}, \ldots e_{i+n}, e_k \in E, e_i \neq e_{i+1} \ldots e_{i+n} \neq e_k$.

The deduced order time constraint is the time constraint between the two deduced order events, and is calculated along the transitive closure paths. Using the distance graph of a bMSC, we can calculate the deduced order time constraint by summing the directed order time constraints. Two cases are involved in calculating the deduced order time constraint from directed order time constraints. In case 1, the local directed order time constraints and the matching pair directed order time constraints are all specified. In case 2, the input indirect causal order and some local directed and matching pair directed order time

constraints are specified. The details of the calculation process are shown in *Appendix B.1*.

However, there may be many different deduced orders between two events connected and calculated by different directed orders. We only consider the shortest one by using the All-pairs-shortest-path algorithm as described in *Section 2.6*.

A part of the deduced orders of the *MSC M1* is depicted in Figure 3.2(c). The deduced order time constraints are $I(s_1,s_2)=[4,6]$, $I(s_1,r_2)=[6,9]$, and $I(r_1,r_2)=[5,7]$. Because the deduced order is a transitive closure of the directed input order without the original input directed order, there may be multiple deduced order time constraints depending on which paths the deduced order is calculated from.

For example, there are deduced order time constraints between e_0 and r_1:

- $I(e_0,r_1)=[6,12]$ by calculating from path $e_0 \rightarrow r_1$;

There are deduced order time constraints between e_0 and s_2:

- $I(e_0,s_2)=[9,12]$ by calculating from path $e_0 \rightarrow r_1 \rightarrow s_2$; or is
- $I(e_0,s_2)=[9,16]$ by calculating from path $e_0 \rightarrow s_1 \rightarrow r_1 \rightarrow s_2$.

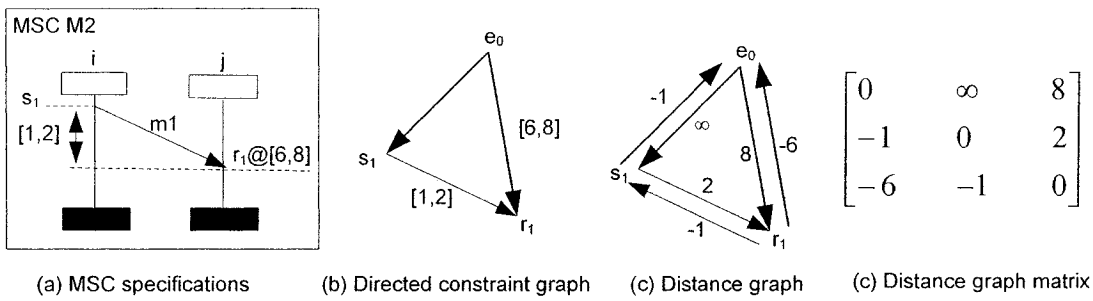
The deduced order time constraints between e_0 and r_2 are:

- $I(e_0,r_2)=[11,13]$ is calculated from path $e_0 \rightarrow s_2 \rightarrow r_2$; or
- $I(e_0,r_2)=[10,20]$ by calculated from path $e_0 \rightarrow s_1 \rightarrow r_2$, or

- $I(e_0,r_2)$=[11,15] is calculated from path $e_0 \rightarrow r_1 \rightarrow s_2 \rightarrow r_2$; or
- $I(e_0,r_2)$=[11,19] by calculated from path $e_0 \rightarrow s_1 \rightarrow r_1 \rightarrow s_2 \rightarrow r_2$.

The following section describes how to calculate the deduced order time constraint from the input directed order time constraints.

For example, in *MSC M2* in Figure 3.3(a), the directed order time constraints are $V(e_0,r_1)$=[6,8] and $V(s_1,r_1)$=[1,2] as shown in Figure 3.3(b). A special event e_0 is added to the distance graph. Using the distance graph and matrix in Figure 3.3(c) and (d), after applying the Floyd-Warshall All-pairs-shortest-path Algorithm, we can do computation based on the two input directed order time constraints to get a deduced order time constraint $I(e_0,s_1)$=[4,7]. Therefore, the s_1 absolute time constraint is @[4,7].



(a) MSC specifications (b) Directed constraint graph (c) Distance graph (c) Distance graph matrix

Figure 3.3 An example of deduced order time constraint

If a timed bMSC has a timed trace, the bMSC is time consistent [9]. The main issue is the following: if we cannot find a timed trace to fulfill the bMSC, how we can find the cause of the inconsistency and how we can propose solutions to correct the bMSC specification?

The intuitive observation leads us to the conclusion that the directed order and the deduced order are in relationship. So, in the following discussion, a correction method is proposed in order to keep all the directed order time constraints and all the deduced order time constraints consistent with each other.

3.3 Time inconsistency checking for bMSCs

A distance graph of bMSCS can be decomposed into elementary cycles. The cycles considered here consist of connected edges with directed orders and deduced orders among events, or several connected edges with directed order among events.

The process used to detect inconsistent traces is an approach of decomposition of distance graphs. The main idea is to decompose a timed distance graph into elementary cycles in order to isolate inconsistent cycles and to correct them.

Proposition 3 gives criteria for checking the inconsistency between the directed order time constraint and the deduced order time constraint.

Proposition 4 focuses on the elementary cycle consisting of a deduced order plus the directed order between two events. This proposition shows how the inconsistency of an elementary cycle can impact the bMSC global inconsistency, and vice versa.

Proposition 5 describes how to track back inconsistent cycles in the decomposition process.

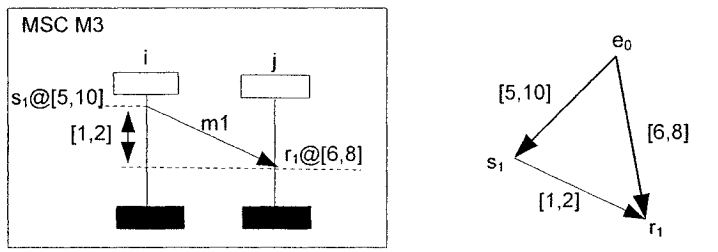
Proposition 3 *A bMSC is consistent if and only if each directed order time constraint and all of its corresponding deduced order time constraints overlap with each other, and the deduced order time constraints also overlap with each other.*

Proof. The proof is given in Appendix B2.

The deduced time constraint (absolute or relative time constraint) is the intersection of the overlapped time constraints. For example, considering the specification *MSC M3* shown as Figure 3.4(a), when a special event e_0 is added, the distance constraint graph is shown in Figure 3.4(b).

In *MSC M3*, the directed order time constraints are $V(e_0,s_1)$=[5,10], $V(e_0,r_1)$=[6,8], and $V(s_1,r_1)$=[1,2]. However, there are also three deduced order time constraints depending on how the orders are calculated. For example, the first deduced order time constraint $I(e_0,r_1)$=[6,12] is based on $V(e_0,s_1)$=[5,10] and $V(s_1,r_1)$=[1,2], and the deduced order time constraint [6,12] is overlapped with the directed order time constraint [6,8]; therefore, using proposition 3, we get the reduced time constraint $I(s_1,r_1)$=[6,8]. The second deduced order time constraint $I(e_0,s_1)$=[4,7] is based on $V(e_0,r_1)$=[6,8] and $V(s_1,r_1)$=[1,2], and the deduced order time constraint [4,7] is overlapped with directed order time constraint [5,10]; therefore, using *definition 2*, we get the deduced order time constraint

$I(e_0,r_1)$=[5,7]. The third deduced order time constraint $I(s_1,r_1)$=[1,3] is based on $V(e_0,s_1)$=[5,10] and $V(e_0,r_1)$=[6,8], and the deduced order time constraint [1,3] is overlapped with directed order time constraint [1,2]; therefore, using proposition 3, we get the reduced relative time constraint $I(s_1,r_1)$=[1,2]. All three deduced order time constraints are overlapped with the corresponding input directed order time constraints meaning that the graph is consistent.



(a) MSC specifications (b) Directed constraint graph

Figure 3.4 An example for proposition 3

Proposition 4 *For a bMSC with n events, we can decompose its complete distance graph of the bMSC into* (2^n-n-1) *cycles so that detecting the bMSC consistency is equal to detecting these cycles' consistency. The bMSC is consistent if and only if all these cycles are consistent. In other words, if there is any inconsistent cycle, the bMSC is not consistent.*

Proof. The proof is shown in Appendix B3.

For example, the *bMSC M3* of Figure 3.5(a) can be transformed into a directed constraint graph as shown in Figure 3.5(b), and then transformed into a distance graph as shown in Figure 3.5(c).

The distance graph of the bMSC can be decomposed into:

1. a cycle formed by e_0, s_1 events (see cycles in Figures 3.5 (d)),
2. a cycle formed by s_1, r_1 events (see the cycle in Figures 3.5 (e)),
3. a cycle formed by e_0, r_1 events (see the cycle in Figures 3.5 (f)), and
4. a cycle formed by e_0, s_1 , r_1 events (see the cycle in Figure 3.5(c); to make computation easy, we view the clock-wise cycle in Figures 3.5 (g) and the anti-clock-wise cycle in 3.5 Figures 3.5 (h) as one cycle).

There are $2^3 - 3 - 1 = 4$ cycles.

In summary, to make *MSC M3* consistent, all the 4 cycles cannot have a non-negative cost.

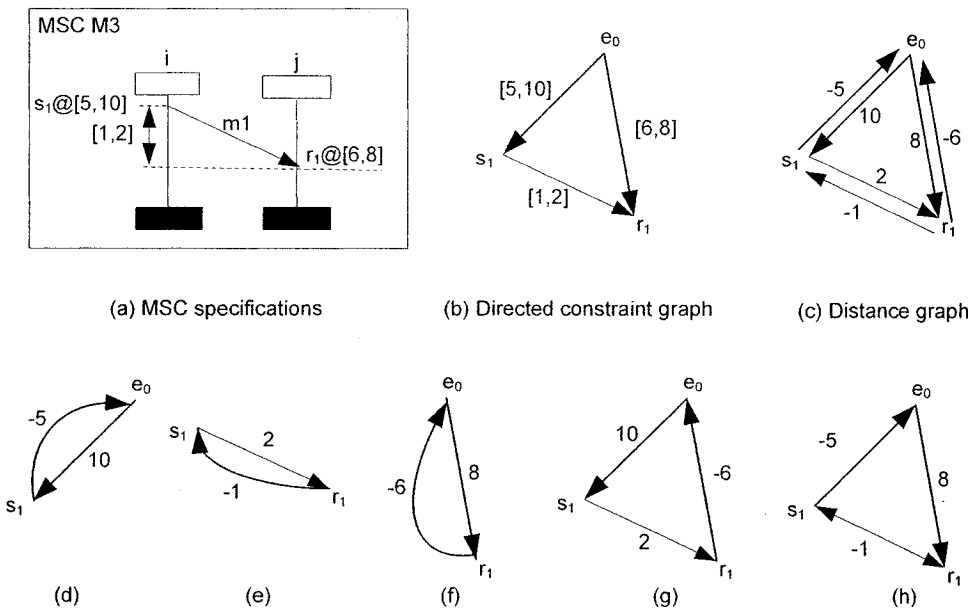


Figure 3.5 Decomposing a bMSC into multiple cycles

It would be useful to consider another example. The *MSC M1* in Figure 3.2 has five events e_0, s_1, r_1, s_2, and r_2. There are $2^5 - 5 - 1 = 26$ cycles that are required to check consistencies.

In conclusion, for a complicated bMSC specification, we can decompose the bMSC into finite number of small simple cycles and check the consistency of the cycles one by one. If there are inconsistent cycles, we have to correct these cycles until all the cycles in the MSC become consistent.

Although Proposition 4 is a possible checking method, the complexity $O(2^n)$ of the computation makes the method impractical for the detection of the consistency of all the cycles. This inspires us to find a more practical method to reduce the complexity.

It is enough to consider the time consistency of only minimum-weight elementary cycles computed by the Floyd-Warshall Algorithm for all-pairs-shortest-path. This minimum-weight reduces the complexity significantly from $O(2^n)$ to $O(n^3)$.

Theorem 3.1[8] indicates that a given STP (simple temporal problem) is consistent if and only if its distance graph has no negative cycles. The former work [25] transforms a distance graph to a distance matrix and applies the Floyd-Warshall (FW) Algorithm to compute the all pair shortest paths. If a negative value exists in the diagonal element of the matrix, the inconsistency of the bMSC is detected. The problem in the algorithm is

that it cannot, in the feed-back, identify precisely which sub-cycle(s) cause the inconsistency problem.

For example, in the *MSC N1* in Figure 3.6, the shortest path from e_0 to e_2 is 4 when we apply the Floyd-Warshall Algorithm along the path from e_0 to e_1 to e_2. However, a negative cycle exists along trace e_0 to e_1 to e_2. By applying a tracing-backing algorithm that we provide in Proposition 4, we can determine that the negative cycle comes from trace e_0 to e_1 to e_2. In this way, we can figure out the cause for the inconsistency.

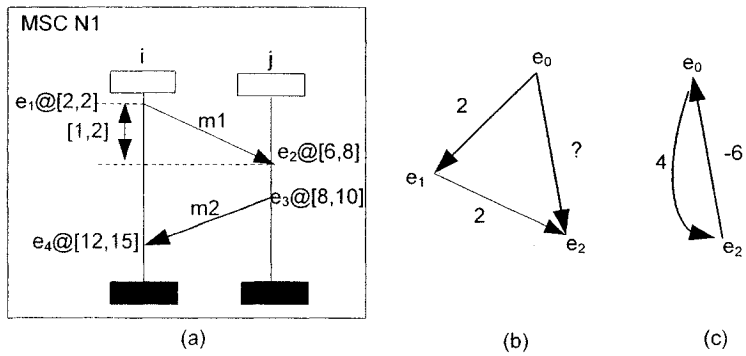


Figure 3.6 An inconsistent bMSC and its negative cycles

The following proposition gives a solution to track the inconsistent cycles in order to decrease the complexity of the checking process.

Proposition 5 *By imposing a predecessor matrix on the Floyd-Warshall Algorithm[20] and by applying a tracing-back algorithm to bMSC's complete distance graph, we can not only detect inconsistency existence in the bMSC but also identify the inconsistent cycles by expanding the sub-traces from the last negative cycles.*

Proof. The proof is shown in Appendix B4.

If we find a negative value in the distance matrix when applying the Floyd-Warshal Algorithm, we can track back from the predecessor matrix and figure out from which cycle the negative value comes. By doing so, we can identify the cause of the inconsistency.

Let adjacency matrix $W=(w_{ij})$ represent the graph of a bMSC. Also, let $d_{ij}^{(k)}$ be the weight of the shortest path from vertex i to vertex j going through intermediate vertices chosen in the set $\{1,2,..i-1,i+1,...j-1,j+1,...n\}$ by the k steps of the Floyd-Warshall algorithm. When $k=0$, the path from vertex i to vertex j has no intermediate vertex numbered higher than 0, and therefore has no intermediate vertices at all. Thus, $d_{ij}^{(0)}=w_{ij}$.

The Floyd-Warshall algorithm [20] in a recursive way is given as:

$$\textbf{Equation 1}: d_{ij}^{(k)} = \begin{cases} Min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}) \text{ if } k \geq 1 \\ w_{ij} \text{ if } k=0 \end{cases}$$

The inconsistency cycles back-track searching process can be calculated based on a predecessor matrix. The predecessor matrix P is constructed on-line just as the distance matrix is constructed. We compute a sequence of matrices $P^{(0)}, P^{(1)},...,P^{(n)}$, where $P=P^{(n)}$ and $p_{ij}^{(k)}$ is defined to be the processor of vertex j on the shortest path from vertex i with all intermediate vertices in the set $\{1,2,...,k\}$.

The recursive formulation $p_{ij}^{(k)}$ is given in the Equation 2:

Equation 2: $p_{ij}^{(k)} = \begin{cases} p_{ij}^{(k-1)} & if\ ((d_{ij}^{(k-1)} \le (d_{ik}^{(k-1)} + d_{kj}^{(k-1)})\ AND\ k \ge 1) \\ k & if\ ((d_{ij}^{(k-1)} > (d_{ik}^{(k-1)} + d_{kj}^{(k-1)})\ AND\ k \ge 1) \\ \varnothing & if\ k=0 \end{cases}$

When k=0, the shortest path from i to j has no intermediate vertices at all. For $k \ge 1$, if we take $i \rightarrow k \rightarrow j$ as the shortest path from i to j, the processor of j is k.

We provide an example to illustrate the process of tracing-back the cycle with a negative value. The *bMSC case_1* in Figure 3.7 has four events with absolute time constraints. However, there are inconsistencies between those events.

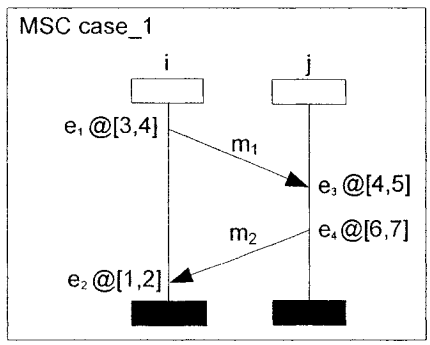


Figure 3.7 An example for tracing back inconsistent cycle

By applying the algorithms of Proposition 5, the distance matrices and the predecessor matrices are shown in Figure 3.8 and Figure 3.9.

The original matrix W

	0	1	2	3	4
0	0	4	2	5	7
1	-3	0	∞	2	4
2	-1	-1	0	-2	-1
3	-4	-1	∞	0	3
4	-6	-2	∞	-1	0

k=0

	0	1	2	3	4
0	0	4	2	5	7
1	-3	0	**-1**	2	4
2	-1	-1	0	-2	-1
3	-4	-1	**-2**	0	3
4	-6	-2	**-4**	-1	0

k= 1

	0	1	2	3	4
0	0	4	2	5	7
1	-3	0	-1	2	4
2	-4	-1	**-2**	-1	-1
3	-4	-1	-2	0	3
4	-6	-2	-4	-1	0

k= 2

	0	1	2	3	4
0	-2	1	0	1	1
1	-5	-2	-3	-2	-2
2	-6	-3	-2	-3	-3
3	-6	-3	-4	**-3**	-3
4	-8	-5	-6	-5	**-5**

Figure 3.8 Parts of distance graph matrix D for MSC *case_1*

k= 0

	0	1	2	3	4
0					
1			0		
2					
3			0		
4			0		

k= 1

	0	1	2	3	4
0					
1			0		
2			1		
3			0		
4			0		

k= 2

	0	1	2	3	4
0					
1			0		
2			1		
3			0	2	
4			0		2

Figure 3.9 Parts of predecessor matrix P for MSC *case_1*

From the distance graph matrix D and the predecessor matrix P, we get:

When $d_{22}^{(1)} = -2 < 0$,

$p_{21}^{(0)} = \varnothing$

$p_{12}^{(0)} = 0$

$p_{10}^{(0)} = \varnothing$

$p_{02}^{(0)} = \varnothing$

Therefore, L(2)={2,1,0,2}.

When $d_{33}^{(2)} = -3 < 0$

$P_{32}^{(1)} = 0$

$P_{30}^{(1)} = \varnothing$

$p_{02}^{(1)} = \varnothing$

$p_{23}^{(1)} = \varnothing$

Therefore, L(3)={3,0,2,3}.

When $d_{44}^{(2)} = -5 < 0$

$p_{42}^{(1)} = 0$

$p_{40}^{(1)} = \varnothing$

$p_{04}^{(1)} = \varnothing$

$p_{24}^{(1)} = \varnothing$

Therefore, L(4)={4,0,2,4}.

In summary, there are three inconsistent cycles, $\{e_2 \rightarrow e_1 \rightarrow e_0 \rightarrow e_2\}$, $\{e_3 \rightarrow e_0 \rightarrow e_2 \rightarrow e_3\}$, and $\{e_4 \rightarrow e_0 \rightarrow e_2 \rightarrow e_4\}$.

We consider another example of time inconsistency between different relative directed order time constraints. The *MSC case_2* is shown in Figure 3.10.

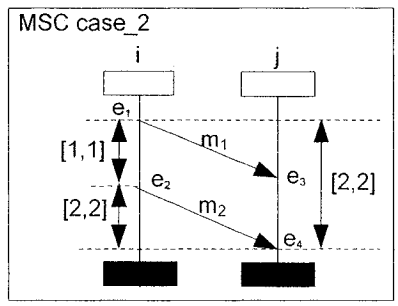


Figure 3.10 MSC *case_2* specification

The directed order relative time constraint of sending-receiving events message pair from $e_2 \rightarrow e_4$ is &[2,2]. The local directed order relative time constraint in the instance *i* from $e_1 \rightarrow e_2$ is &[1,1]. The input order pair directed order relative time constraint $e_1 \rightarrow e_4$ is &[2,2].

The algorithms in Proposition 5 are applied to the bMSC distance graph. When $k=0$, and $k=1$, there are no negative cycles. However, when $k=2$, $d(e_4e_4)^{(2)} = min(d(e_4e_4)^{(1)}, d(e_4e_2)^{(1)} + d(e_2e_4)^{(1)}) = min(0, -2+1) = -1 < 0$.

When $k=0$, all entries in the predecessor matrix are $\varnothing$s. By computing the predecessor matrix, the following matrix, when $k = 1$ in Figure 3.11, is calculated as:

k= 0

	0	1	2	3	4
0					
1					
2					
3					
4					

K= 1

	0	1	2	3	4
0					
1					
2	1				1
3	1		1		1
4	1				

Figure 3.11 Parts of the predecessor matrix P for MSC *case_2*

When $d[4,4]^{(2)} < 0$, the algorithm of Proposal 5 is applied to the predecessor matrix of $k=1$, we get:

$p_{44}^{(1)} = 2$

$p_{42}^{(1)} = \varnothing$

$p_{24}^{(1)} = 1$

$p_{21}^{(1)} = \varnothing$

$p_{14}^{(1)} = \varnothing$

Therefore, $L(4)=\{4,2,1,4\}$. We conclude that there is a negative cycle among events $4 \rightarrow 2 \rightarrow 1 \rightarrow 4$.

Moreover, an MSC specification designer can also impose some policies to further decrease the complexity. For example, in *MSC M3*, which is shown in Figure 3.5 (a), if the MSC designer decides that the events in instance *i* cannot be changed, and the relative time constraint cannot be changed either, thus, only cycles described in Figures 3.5 (f), and (c) need to be checked. Considering the Floyd-Warshall algorithm used in the procedure, this decision will dramatically decrease the computation of the time consistency checking.

3.4 Inconsistent bMSCs correction policies

Based on the causes of time inconsistency, the MSC specification designer may be asked to choose a correction procedure to eliminate the inconsistency of the MSCs. We propose four correction policies that apply to all inconsistent bMSCs, but the policy or the combination of policies applied to a specific specification are determined by the MSC designer.

Policy 1 - *Absolute Time Correction*: *Let us assume that an MSC designer decides that some absolute time constraints in one specific instance are changeable. The absolute time constraints are removed. The new relaxed time constraints are imposed on the MSC specification. We can apply the FW Algorithm to find out the absolute time constraint in the instance.*

For example, the *MSC* N_2 in Figure 3.12(a) is not consistent. We can apply Policy 1 on events e_1 in instance i as described in Figure 3.12 (b) and (c). The absolute time of event e_1 is removed. By applying the Floyd-Warshall Algorithm, we get e_1 absolute time constraint *@[4,7]*.

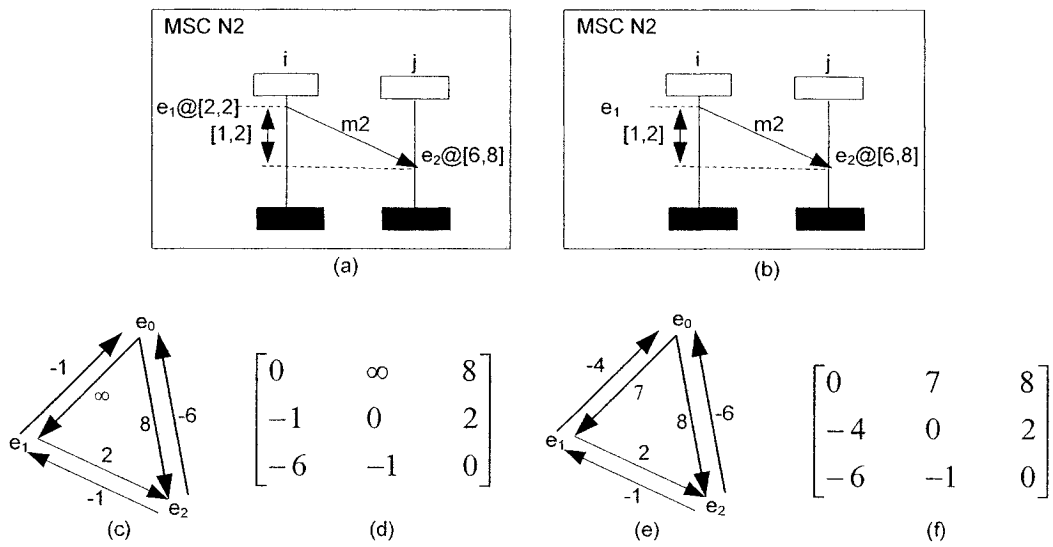
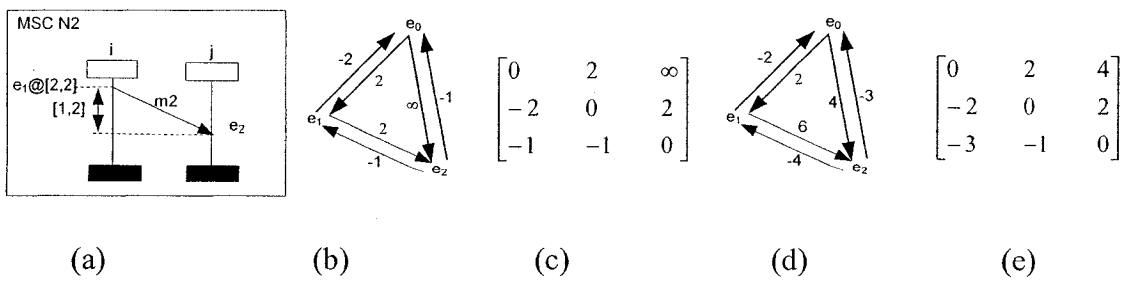


Figure 3.12 Apply Policy 1 in *instance i*

We can also apply Policy 1 on event e_2 in instance j, and we get the new MSC as described in Figure 3.13(a) and (b). By applying the Floyd-Warshall Algorithm, we get e_2 absolute time constraint @[3, 4].



(a) (b) (c) (d) (e)

Figure 3.13 Apply Policy 1 in *instance j*

In another example, the *MSC case_1* in Figure 3.14(a) is detected the presence of the negative cycles $\{e_2,e_1,e_0,e_2\}$, $\{e_3,e_0,e_2,e_3\}$, $\{e_4,e_0,e_2,e_4\}$. From these messages, the MSC designer can identify that the common events are e_0 and e_2. The MSC designer can then decide to apply Policy 1 on event e_2 of the instance i to relax on the absolute time constraints as described in Figure 3.14(b). The FW algorithms is applied to the MSC and gets the consistent MSC as described in Figure 3.14(c).

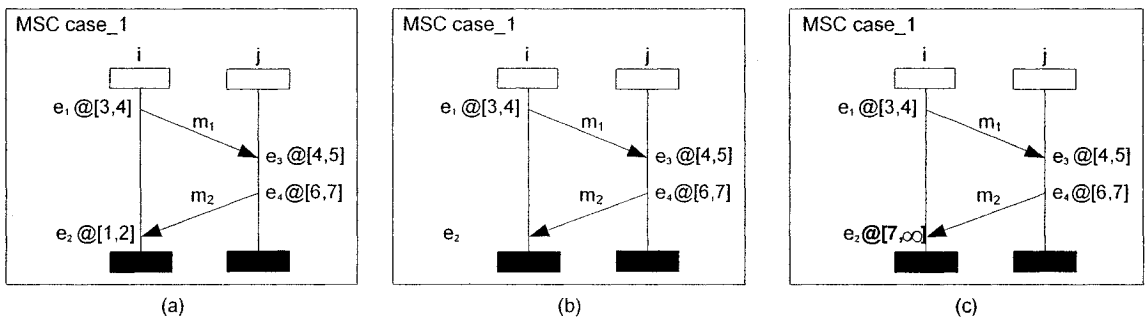
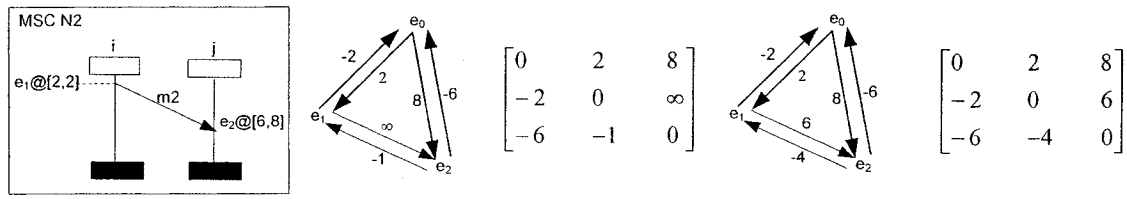


Figure 3.14 Apply Policy 1 to correct absolute time constraints

Policy 2 - Local Directed Relative Time Correction: Let us a*ssume that an MSC designer decides that it is possible to change the relative time constraint in a specific instance. We can apply the FW Algorithm to calculate the reduced relative time constraint.*

Policy 3 - Matching Pair Relative Time Correction: Let us a*ssume that an MSC designer decides that it is possible to change the relative time constraint between the matching pair of events of some messages. We can apply the FW Algorithm to calculate the matching pair relative time constraint.*

For the *MSC N2* in Figure 3.6, we can change strategies by applying Policy 3 on message pair events e_1 and e_2 as described in Figure 3.15. After applying the Floyd-Warshall algorithm, we get e_1 and e_2 relative time constraint &[4, 7].

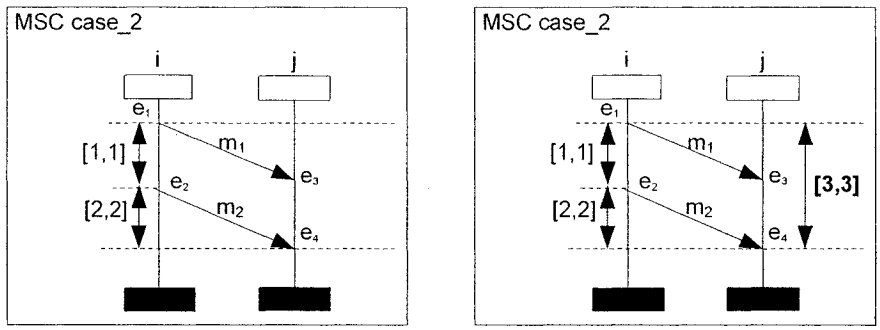


(a) (b) (c) (d) (e)

Figure 3.15 Apply Policy 3 on message *m2*

Policy 4 - Indirect Causal Order Time constraint Correction: Let us a*ssume that an MSC designer decides that the indirect causal order time constraint between two instances is changeable. We can apply the FW Algorithm to get the reduced indirect causal order time constraints.*

For example, *MSC case_2* in Figure 3.10 is not consistent. After applying the back-tracing algorithms of Proposition 5, the system designer gets the information that there is a negative cycle $\{e_4, e_2, e_1, e_4\}$. The designer may decide to apply Policy 4 on the event e_1 to e_4. The relaxed time constraints are shown in Figure 3.16 (a). After applying the FW Algorithm, the designer gets a result that the consistent time constraint between e_1 to e_4 is &[3, 3] as Figure 3.16 (b). Thus, the MSC is consistent.



(a) Before applying Policy 4 on indirectly causal order between events e_1 and e_4

(b) Apply Policy 4 on indirectly causal order between events e_1 and e_4

Figure 3.16 Apply Policy 4 on a bMSC

3.5 Algorithms to check inconsistency and correction policies

According to Proposition 4, to find the causes of time inconsistency in a bMSC, we theoretically need to check all the $(2^n\text{-}n\text{-}1)$ cycles in the distance graph of the bMSC, and then correct the inconsistent directed order time constraints or deduced order time constraints. The complexity of the checking algorithm will be $O(2^n)$.

However, we can check inconsistency in an STP network by applying the Floyd-Warshall algorithm to its distance graph to reduce the complexity. Next, we can apply the methods in Proposition 5 to trace back the cycles with negative value, and then impose the correction policies decided by the *MSC* designer to correct these inconsistent cycles until all the cycles are consistent. In this way, we can reduce the complexity to $O(n^3)$. This can be achieved in the following steps:

- Step 1: First, we transform a bMSC specification to a complete directed constraint graph G with special event e_0 as the starting event. Next, we transform the

directed constraint graph to a complete distance graph expressed in an adjacent matrix W. Then, the Floyd-Warshall algorithm is applied to compute the value of elements in the matrix. If a negative value is detected, an inconsistency cycle occurs. Then the algorithm in Step 2 is applied. We can apply the tracking back algorithm to find the inconsistent paths according to Proposition 5. Otherwise, if no negative cycle is detected, this bMSC is consistent.

- Step 2: If some paths are inconsistent, one policy or a combination of policies is used for correcting the inconsistent specification until all the paths are consistent.

The detailed algorithm is as follows.

Step 1: A bMSC graph is represented by an adjacent matrix W=(w_{ij}). Assume the input is d_{ij}=0 if i=j; or else, d_{ij}= w_{ij}, if $i \neq j$. Based on Equation 1 given in *Section 3.3*, we can construct the matrix D(n) = ($d_{ij}^{(n)}$). The bottom-up procedure can be used to compute the values $d_{ij}^{(k)}$ in an increasing order of k. Its input is an $n \times n$ matrix W.

The predecessor matrix P is constructed on-line just as the matrix D is. We compute a sequence of matrices $P^{(0)}$, $P^{(1)}$,...,$P^{(n)}$, where P=$P^{(n)}$ and $p_{ij}^{(k)}$ is defined to be the processor of vertex j on the shortest path from vertex i with all intermediate vertices in the set $\{1,2,...,k\}$. The recursive formulation $p_{ij}^{(k)}$ is given in the equation 2 in *Section 3.3*.

When k=0, the shortest path from i to j has no intermediate vertices at all. For $k \geq 1$, if we take $i \rightarrow k \rightarrow j$ as the shortest path from i to j, the processor of j is k.

Let L be a vector of size n of linked lists used to store the indexes of events involved in negative cycles. For instance L[i]={i,j,k,i} represents the negative cycle $i \rightarrow j \rightarrow k \rightarrow i$.

Algorithm: TimeConsistentChecking(in W, in-out P)

```
1   n ←rows[W]
2   Let isConsistent be true
3   D(0) ←W
4   for k←1 to n
5           do for i←1 to n
6                   do  for j←1 to n
7                           if ( (d_ij^(k-1) > (d_ik^(k-1)+d_kj^(k-1)) )
8                                   d_ij^(k)= (d_ik^(k-1)+d_kj^(k-1))
9                                   P_ij^(k)=k;
10                          end i
11                  end do
12                  if (d_ii^(k)<0) then
13                          L^(k)[i]={}
14                          isConsistent = false
15                          do Tracking_back_Path(P^(k),i, i, L^(k)[i])
16                  end if
17          end do
18  end for
19  return isConsistent
```

Algorithm: Tracking_back_Path (in P, i, j, in out L)

```
20  Let x=P_ij
21  if  x != ∅ then
22          L.add(x)
23          Tracking_back_Path (P,i,x, L)
```

24 Tracking_back_Path (P,x,j, L)

25 end if

Step 2: If some inconsistent cycles are detected in Step 1, based on the error messages that indicate which paths have negative cycles and therefore are inconsistent, the designer is asked to input a correction policy or a series of predefined correction policies to correct the inconsistent bMSC until the designer gets a consistent bMSC. By following the policy(s) chosen by the designer, we can apply the following algorithms to get the corresponding reduced relative time constraints and absolute time constraint. The algorithms for the four correction policies are as follows.

Policy 1 - Absolute Time Correction Algorithm: *In this policy, we assume that some absolute time constraints in one specific instance are changeable. We get rid of the absolute time constraint of events, and then get a new MSC. By applying the FW Algorithm, we can find out the absolute time constraint in one specific instance.*

Algorithm: AbsoluteTimeCorrection(in W, int n, in L)

26 input a instance name and event names on the instance

27 get ride of the absolute time constraint of the events and modify the graph matrix

28 check if the modified matrix is consistent after applying the Floyd-Warshall Algorithm

29 if (the matrix has no negative cycles) then

30 report the recommended reduced absolute time constraint to the designer

31 else

32 continue to apply the Policy 1,2,3, or 4

33 end if

Policy 2 - Local Directed order Relative Time Correction Algorithm: *We assume that the relative order time constraint in a specific instance is changeable. We relax the relative time constraint, and then we get a new MSC. By applying the FW Algorithm, we can determine the relative time constraint.*

Algorithms: LocalRelativeTimeCorrection(in W, int n, in L)

```
34  input the instance name and event names of the relative time constraint on the instance
35  find the cycle C that the relative time constraint fit in
36  get ride of the time constraint between these events  and modify the graph matrix
37  check if the modified matrix is consistent by applying Floyd-Warshall's algorithm along the cycle C
38  if (the cycle has no negative value) then
39    report the recommended reduced relative time constraint to the designer
40    continue to apply policy 1,2,3, or 4 on the other negative cycle
41  else
42    the policy is not applicable on the relative time constraint
43    continue to apply the Policy 1, 2, 3, or 4
44  end if
```

Policy 3 - *Matching Pair Relative Time Correction Algorithm:* *We assume that the relative time constraint between matching pairs of events of some messages is changeable. We can relax the absolute time constraint of the event, and then we get a new MSC. By applying the FW Algorithm, we determine the matching pair relative time constraint.*

Algorithm: MatchingPairEventsRelativeCorrection(in W, int n, in L)

```
45  input the message name of the message and get the matching pair event names
46  find the cycle C that the relative time constraint fit in
47  get ride of the time constraint between these events  and modify the graph matrix
```

48 check if the modified matrix is consistent by applying Floyd-Warshall's algorithm along the cycle

49 if (the cycle has no negative value) then

50 report the recommended reduced relative time constraint between matching pairs to the designer

51 continue to apply the Policy 1,2,3, or 4 on other cycles

52 else

53 the policy is not applicable to the matching pair relative time constraint

54 continue to apply the Policy 1, 2, 3, or 4

55 end if

Policy 4 - Indirect Causal Order Correction Algorithm: *We assume that the relative order time constraint between two or more instances is changeable. We can relax the absolute time constraint, and then obtain a new MSC. By applying the FW Algorithm, we can ascertain the indirect causal order time constraints.*

Algorithms: IndirecCausalOrderTimeCorrection(in W, int n, in L)

56 input event names of the relative time constraint

57 find the cycle C that the relative time constraint fit in

58 get ride of the time constraint between these events and modify the graph matrix

59 check if the modified matrix is consistent by applying Floyd-Warshall's algorithm along the cycle C

60 if (the cycle has no negative value) then

61 report the recommended reduced relative time constraint between the indirect causal pairs

62 continue to apply the Policy 1,2,3, or 4 on other cycles

63 else

64 the policy is not applicable to the matching pair relative time constraint

65 continue to apply the Policy 1, 2, 3, or 4

66 endif

In *step 1*, the running time of the *TimeConsistentChecking* algorithm is determined by the triple nested loops in line *4-6*. Each execution in line *7-9* takes O(1) time, so the algorithm of this step runs in a complexity of $O(n^3)$. The *Tracking_back_Path* algorithm will traverse n vertices of precedence matrix P in the worst case. Therefore, the complexity is O(n). The *TimeConsistentChecking* algorithm calls on the *Tracking_back_Path* algorithm at each step and in the worst case, the overall complexity is $O(n^4)$.

In *step 2*, the complexity of all the correction policies is determined by Floyd-Warshall's algorithm. Therefore, the complexity of *step 2* is $O(n^3)$.

In summary, the complexity of the overall running time is $O(n^4 + n^3)=O(n^4)$ for n events in the worst case.

3.6 Discussion

There are many causes for time inconsistency in bMSCs, such as absolute time causal order conflicts, relative time constraint conflicts along some paths, relative and absolute time conflict, etc. Directed order time constraint and deduced order time constraint are defined to find the causes for inconsistency. We can calculate the deduced order time constraints based on the directed order time constraints. The visual and the deduced order time constraint have to be overlapped to meet the requirements of the sufficient and necessary condition for the time consistency.

In order to trace the inconsistent part of a bMSC, we have to decompose a bMSC into finite sub-graphs and check each cycle. A more efficient method is to use feed-back messages to display inconsistency information and to provide correction decisions according to designer policies.

When inconsistencies in a bMSC are found and the causes of the negative cycles are detected, we define four correction policies. If an MSC designer determines that it is necessary to change the absolute time constraint of one or some events, Policy 1 is applied. When the designer decides to change the local directed order time constraint in one or more specific instances, Policy 2 is applied. When the designer wants to change relative time constraints of a message pair, Policy 3 is applied. If the designer needs to change the deduced order of time constraints, Policy 4 is applied.

We have built a tool named MSCTICC (MSC Time Inconsistency Checking and Correction) that implements these solutions. In this tool, we implement the time inconsistency checking, inconsistent cycles tracing back, and the correction policies algorithms.

Chapter 4

Time inconsistency analysis and corrections for HMSC specifications

4.1 Introduction

A real-time distributed system can be specified and designed using HMSC. HMSC expresses high level functional requirements and constraints. A timed HMSC is composed of a set of bMSCs together. Time constraints are specified within bMSCs and may cause time conflicts within bMSCs and also between bMSCs in an HMSC. This is the problem of time inconsistency of HMSCs.

If there are some errors in an HMSC concerning time constraints, the errors can cause partial inconsistency (the HMSC is called weakly consistent) or total inconsistency (the HMSC is called inconsistent). The HMSC is not valid from the semantics point of view. These time inconsistencies make the HMSC specifications un-implementable, or result in undesired implementations. Especially, when HMSCs are working with SDLs to transfer the MSC specifications to SDL architectures, the inconsistent time constraints in HMSCs can cause the SDLs to be invalid. The goal of this thesis is not only to validate the time consistency in HMSC but also to correct the inconsistency until a consistent one is achieved.

The rest of the chapter is organized as follows. In *Section 4.2*, we introduce the time inconsistency and analyze the reasons that may cause the time inconsistency in HMSCs.

Based on the analysis, we propose correction solutions according to the types of the paths of the HMSC. In *Section 4.3*, we describe algorithms for checking inconsistencies in HMSCs and for correcting these inconsistencies. We discuss the related work in *Section 4.4*.

4.2 Time inconsistency for HMSCs

The consistency of an HMSC is defined in terms of the consistency of its paths. A path is consistent if and only if the corresponding lposet, obtained by composing sequentially all the bMSCs in the path, has a timed trace.

An HMSC can be strongly or weakly consistent (depending on whether all parts or only some parts of the specification are consistent respectively), or inconsistent. The general and special case algorithms are proposed [26].

An HMSC is like a roadmap for a system. An HMSC specifies a set of scenarios (paths) to be implemented in the requirement and design specification. If all the scenarios are mandatory and have to be implemented, and if every scenario (path) is time consistent, we claim the HMSC is strongly consistent.

Sometimes, an MSC specifies that at least one path is time consistent but not all paths are time consistent. Under this circumstance, some scenarios conform to time consistency and other scenarios do not. In this case, we claim that the HMSC is weakly consistent.

Otherwise, if we cannot find any path that satisfies the time constraints, we claim that the MSC is inconsistent.

If an HMSC is inconsistent, we have to refine some time constraints of bMSCs in some paths until the HMSC becomes consistent for all the time constraints of all the paths; otherwise, if the HMSC is weakly consistent, we have to identify which paths are inconsistent and correct the time constraints until time consistency is achieved.

Several causes of HMSC inconsistency are identified. One such cause may be that there are inconsistent bMSCs existing in the HMSC specification. The loop paths may also cause inconsistency if the absolute time constraints of some events in a loop restrict the occurrence of the loop. Another potential cause may be that the conflicts of time constraint exist among bMSCs in a path of a HMSC.

Intuitively, we analyze the causes of the inconsistency of HMSCs and propose some solutions to correct the inconsistent HMSC step by step until consistent ones are achieved.

As defined in *Section 2.7*, a **simple path** of an HMSC is defined as a finite or infinite sequence of nodes. The simple path begins from a start node and end with an end node or a loop. It is a prefix of a path. There are no identical nodes in a simple path.

If a simple path is consistent, all the composed bMSCs has to be consistent. Otherwise, if there is any bMSC that is not consistent, it implies that the HMSC path cannot be consistent.

According to *Definition 4* in [26], a simple path is consistent if and only if the lposet, obtained by composing sequentially all the bMSCs lposets, has a timed trace. If a bMSC in a path of HMSC is not consistent, there is no timed trace in the bMSC. Thus, when it is combined with other bMSCs to form a simple path, the composing lposet will not have a consistent timed trace.

For example, in Figure 4.1, to make *HMSC H4* consistent, both *MSC* N_5 and N_6 have to be consistent. Because of the inconsistency of *MSC* N_6, there is no timed trace in *MSC* N_6 that conforms to the time constraint requirements. Therefore, there is no timed trace in *HMSC H4* when the two bMSCs are sequentially composed together. The HMSC is not consistent in this case.

To facilitate further discussion in differentiating the type of inconsistencies of simple paths, we extend the concept of simple path into simple path with loop (called flow path), loop path, and simple path without loop (called combinational-flow-loop path).

Definition 6 Let H = (S,D,L) be an HMSC. *A **flow path** of an HMSC is defined as a finite sequential composition of node* $s_0 s_1 ... s_n$ *where* s_0 *is the start node;* s_n *is the end node;* $(s_j,s_i) \notin D$, *in which* $s_i \neq s_j$, *if* $i \neq j$, $s_i \in S$, $s_j \in S$, $0 \leq i \leq j \leq n$; *and* $(s_i,s_{i+1}) \in D$.

A flow path begins with a start node and end with an end node. Further more, there are no node(s) in loop(s). Therefore, a flow path contains only a finite node of paths.

HMSC H4 in Figure 4.1 is an example of an HMSC with a flow path $\{N_3N_4\}$. The HMSC is sequentially composed of *MSC* N_3 and N_4 without loops.

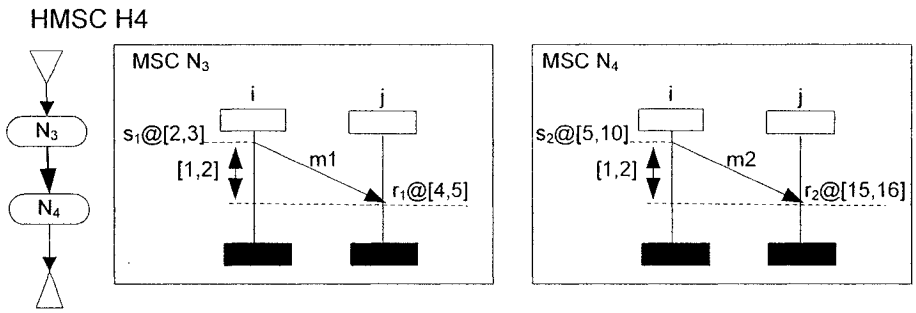


Figure 4.1 An inconsistent HMSC with an inconsistent bMSC

An alternative composition of an HMSC can be decomposed into two flow paths. For example, the *HMSC H2* in Figure 2.15 can be decomposed into two flow paths $\{L_1\}$ and $\{L_2\}$.

Definition 7 *A **loop path** is defined as a sequence of nodes* $s_1 \ldots s_n$, *in which* s_1 *is the start event,* $(s_i, s_{i+1}) \in D$, $0 \leq i \leq n$; *the path can end a loop such that* $(s_n, s_1) \in D$.

A loop path is a simple path ending with a loop. The HMSC in Figure 2.16 is an example of a loop path. The HMSC has only one path where *MSC M* repeats infinitely.

Definition 8 ***A combinational-flow-loop path*** *is a sequence of nodes* $s_1 \ldots s_n$, *in which* s_1 *is the start event,* s_n *is the end node, and there are nodes* s_j *and* s_i *such that* $(s_j, s_i) \in D$, $s_i \neq s_j$, *if* $i \neq j$, *and* $s_i \in S$, $s_j \in S$, $0 \leq i \leq j \leq n$.

An example of a combinational-flow-loop path with a node in a loop is shown in Figure 4.2. The node bMSC N_5 *is* in a loop followed by bMSC N_6 *and an end node*. Another example of a combinational-flow-loop is shown in Figure 2.6. The *connection point* is a node in a loop but also in the simple path {*s1, s2, connection point, s4, end node*}.

We can check the time inconsistency of a path in an HMSC according to the type of the path. Furthermore, we can apply different correction policies and strategies according to the type of inconsistent paths.

A loop path and a combinational-flow-loop path are most often used to express scenarios where the bMSCs in the loop will repeat in an infinite number of times in HMSCs. We discuss this type of time inconsistency in the following section first, and then we discuss the inconsistency of normal path.

Loops in an HMSC may cause inconsistency if events in a loop are constrained by absolute time requirements. In an HMSC, a loop path generates an infinite number of paths. When a loop is repeated, an HMSC is composed sequentially of bMSCs. All the occurrences of events in a bMSC are constrained by their time constraints. The absolute

time constraint of an event and the relative constraint between two events are not changed in the iteration of the loop.

Intuitively, if the upper bounds of all the absolute time constraints are infinite in a consistent bMSC, events in the bMSC can be repeated infinitely without violating their absolute and relative time constraints in the discrete time domain.

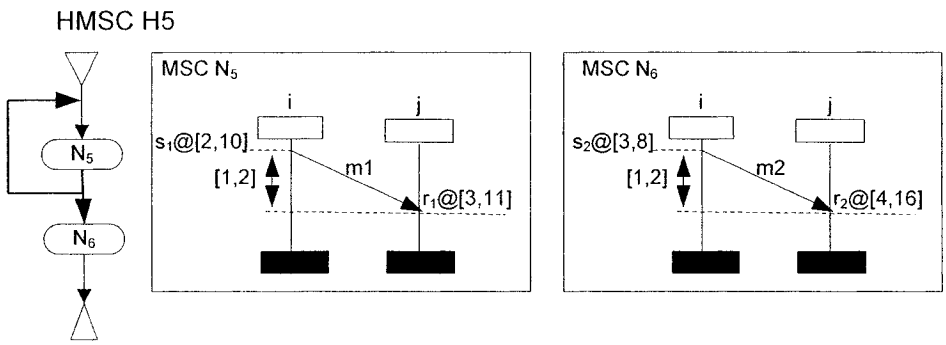


Figure 4.2 An inconsistency HMSC path with a node in a loop

In the combinational-flow-loop path H5 as described in Figure 4.2, the *HMSC H5* can be decomposed into a simple path $\{N_5^* \cdot N_6\}$. The N_5^* stands for the *MSC* N_5 in a loop and will repeat an infinite number of times. If the *MSC* N_5 does not contain absolute time constraints, the instance i can send a message m_1 an infinite number of times and the HMSC describes an infinite number of paths N_5N_6, $N_5N_5N_6$... etc. However, the absolute time constraint @[2,10] restricts instance i to sending message $m1$ only 9 times because our time domain is fixed to non-negative integers. Then all the paths $N_5^iN_6$ (i>9) are not consistent.

Moreover, the consistency of such a combinational-flow-loop path may also be affected by the absolute time constraints of the bMSCs following the loop. Due to the absolute time constraint in *MSC* N_6, the message *m2* must be sent within time [3,8]. Since sending message *m1* has to occur before sending *m2*, *m1* can only be sent within time [2,7]. That is 6 times at most. So, all the paths $N_5^i N_6$ (i>6) are not consistent.

Based on the intuitive analysis of the examples above, we describe a basic requirement of HMSC consistency of loop path as follows.

If an HMSC is strongly consistent, all events of the bMSC in every loop path must have infinite upper bounds in their absolute time constraints so that there will always be time points for the events when the loop is repeated. Moreover, if an event *e* is in a bMSC that follows a loop and the event *e* occurs causally after an event *f* in the loop, the occurrence time of *e* has to be later than the occurrence of *f*. The upper bound of the absolute time constraint for *e* has to be infinite also.

For example, in the *HMSC H5* of Figure 4.2, the upper bounds of event s_1 and r_1 in bMSC N_5 have to be infinite to keep the loop consistent; moreover, the events s_2 and r_2 in bMSC N_6 are causally after the event s_1 and r_1 in the loop. The upper bounds of s_2 and r_2 have to be infinite also to keep the HMSC time consistency.

In a simple path of an HMSC, the consistency of each bMSC component does not guarantee the overall consistency of the HMSC. For example, the *HMSC H6* in Figure

4.3 consists of a sequential composition of *MSC* N_7 and *MSC* N_8. The time constraints in *MSC* N_7 and N_8 are consistent respectively. However, when *MSC* N_7 and N_8 are composed sequentially, the absolute time constraints of events s_1 and s_2 contradict their local directed order along instance *i* lifeline, and the absolute time constraints of events r_1 and r_2 contradict the local directed order along instance *j* lifeline. Therefore, the HMSC is not consistent even though the bMSC components are consistent. Additional work should be done to ensure the consistency of the HMSC.

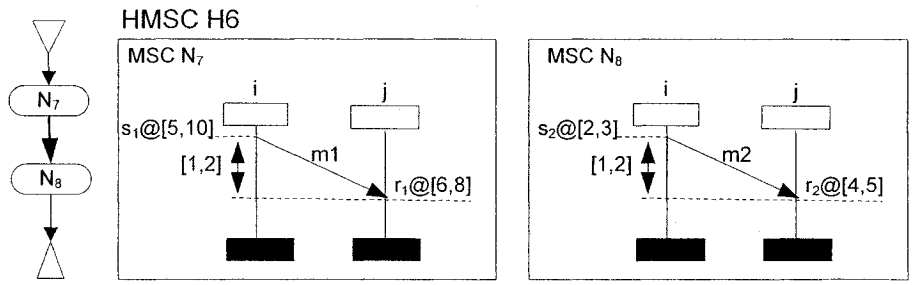


Figure 4.3 An inconsistent HMSC with time conflict between bMSCs

To determine the cause for inconsistency in a simple path of an HMSC, we need to check every sub-simple path also. A sub-simple path is a sequential part of the simple path.

Definition 9 *Let* $H = (S,D,L)$ *be an HMSC, and let a simple path be* $s_0 s_1 \ldots s_{i-1} s_i s_{i+1} \ldots s_{j-1} s_j s_{j+1} \ldots s_n$. *A sub-simple path is a sequential of node* $s_i s_{i+1} \ldots s_{j-1} s_j$, *in which* $(s_i, s_{i+1}) \in D$, $s_i \neq s_j$, *if* $i \neq j$, $s_i \in S$, $s_j \in S$, *and* $0 \leq i \leq j \leq n$.

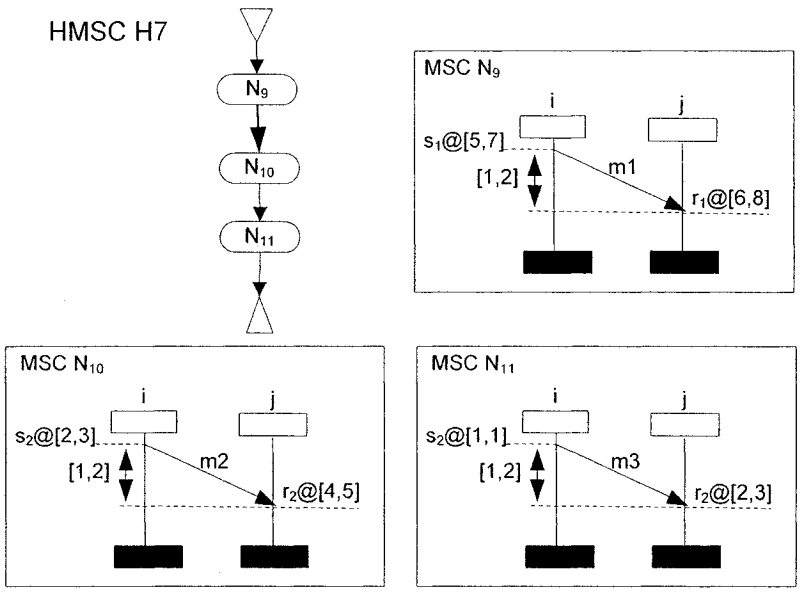


Figure 4.4 An inconsistent HMSC with a simple path

Proposition 10 *A simple path of HMSC is consistent if and only if all of its sub-simple paths are consistent.*

Proof. The proof is shown in Appendix B.5.

For example, in the *HMSC H7* of Figure 4.4, the consistency of HMSC needs all the simple sequential sub-simple paths $\{N_9N_{10}\}$, $\{N_{10}N_{11}\}$, $\{N_9N_{10}N_{11}\}$ to be consistent.

Proposition 11 *To find out the causes of the inconsistency of a simple path composed of n consistent bMSCs sequentially, we have to check the sub-simple paths n × (n-1)/2 times.*

Proof. The proof is shown in Appendix B6.

For example, in Figure 4.4, b*MSCs* N_9, N_{10}, and N_{11} are consistent, and their reduced absolute time constraints can be calculated by applying the FW Algorithm to those bMSCs. However, when they are composed sequentially to form one HMSC, we check sub-simple paths $\{N_9N_{10}\}$, $\{N_{10}N_{11}\}$, and $\{N_9N_{10}N_{11}\}$. We get the result that sub-simple path $\{N_9N_{10}\}$ is inconsistent, sub-simple path $\{N_{10}N_{11}\}$ is inconsistent, and sub-simple path $\{N_9N_{10}N_{11}\}$ is inconsistent. Therefore, there are 3*2/2 = 3 checking processes.

The strategies used to correct the inconsistent HMSCs are applied according to the decision of the MSC specification designer. Based on the consistency result report of sub-simple paths, the MSC designer can judge which bMSCs cause the inconsistency. Then, all the absolute time constraints of the bMSCs are removed, and a new timed lposet is composed based on the new specification with the changed timing requirement. And then the FW Algorithm is applied to the new bMSC. The reduced absolute time constraints can be achieved if the new lposet is consistent. Otherwise, a new correction policy needs to be further applied until we get a consistent one.

4.3 Algorithms to correct the inconsistency for HMSC

If an HMSC is inconsistent or weakly consistent, to ascertain the cause of the inconsistency and then to correct the inconsistency, we first check every bMSC and correct any inconsistent bMSC. Next we decompose the HMSC into paths; we check the time consistency of every simple path. Based on the type of the path, different correction

policies are applied to any inconsistent ones until a consistent path is obtained. This procedure can be achieved in the following steps:

Step 1: *We find out all bMSCs by using Deep-first-search (DFS) algorithms to traverse the directed graph of the HMSC with the bMSCs as nodes. Then, we check the consistency of every bMSC and correct the inconsistent bMSCs using algorithms as described in Chapter 3.*

Algorithm: Check&Correct_bMSCs()

```
For (each bMSC ) do
  If ( the bMSC is not consistent) then
        Travse graph to find all bMSCs;
        Apply bMSC_Checking&Correction Algorithms;
        Apply Floyd_Warshall Algorithms to get reduced absolute time constraints of the bMSC;
    CorrectInconsistentHMSCPath()
  End If
End For
End Algorithm
```

Step 2: *We traverse the HMSC to determine all simple paths and check the consistency of each path and correct the inconsistent paths.*

If a simple path ends with a loop or it is combinational-flow-loop path, we determine whether all the absolute time constraints of events of bMSCs in the loop has infinite upper bounds and we also determine whether the upper bounds of events reached from

the loop are also infinite. If they are not, we have to correct their upper bounds so that they are infinite.

If a simple path does not have a loop, we compose all bMSCs of the path into a bMSC and check its consistency. If it is not consistent, we apply error detection algorithms in the following way: we combine every two sequential bMSCs into one bMSC and check the bMSC consistency; we combine every three sequential bMSCs into one bMSC. until all n sequential bMSCs of the HMSC form a bMSC and ascertain the bMSC consistency. After reading the error messages of inconsistency, a designer makes a judgment about which bMSCs cause the inconsistency, and inputs the names of the bMSCs that are changeable. The absolute time constraints of events in the changeable bMSCs are removed and a new HMSC path is formed. The new path is applied the checking and correction algorithms until it becomes consistent.

Assume an HMSC is inconsistent or weakly consistent, the detailed algorithm for correction is as follows.

Let Simple_Path[n] be the path sequentially composing of bMSCs.

Loop__Path[n] be the path with a loop.

S be the stack to hold node bMSC.

Algorithm: CorrectInconsistentHMSCPath()

Travse_Graph_To_Find_simple_Path();

For (Each simple_ path in Loop_Path[n]) do

For (each node s such that s∈ Loop_Path) Do

For (each event e_i in s) Do

If $ubr_i \neq \infty$ then

Assign $ubr_i = \infty$

End IF

End For

End For

End For

For (Each path in Simple_Path[n]) do

For (Each path in Loop_Path[n]) do

Calculate the Afetr_loop_Path

For (each node s such that $s \in$ After_loop_Path) Do

For (each event e_i in s) Do

If $ubr_i \neq \infty$ then

Assign $ubr_i = \infty$

End IF

End For

End For

End For

End For

For (Each path in Simple_Path[n]) do

For (each node x in the path from begin to the second node to the end) do

For (next node to x in the path to the end) do

1. Check the simple path consistency

2. Input which bMSCs are changeable, and apply Step-by-step correction algorithms to correct the bMSC

End For

```
    End For
End For
```

Algorithm: Travse_Graph_To_Find_Path()

```
S.Push(beginning node);
For (each neighbor w to vertex v) do
    If (there is an element x == w and x∈ S) then
            Put elements from x to the top element in the stack S into Loop_Path
    End If
        S.Push(w)
End For
IF (top element x == end element of the graph)
    Put elements in the stack to Simple_Path
End IF
S.Pop()
End Algorithms
```

In the aforementioned algorithms, *Check&Correct_bMSCs* checks the time constraints of each node. For each node, in the worst case, the complexity of checking and of correction of the bMSC inconsistency is $O(n^4)$. Therefore, if there are m bMSCs, the complexity is $O(m \times n^4)$, where m is the number of bMSCs, and n is the event number in a bMSC.

In Step 2, in the algorithm *Travse_Graph_To_Find_Path*, the deep-first-search complexity is $\Theta(l + n)$, where l is the edge number, n is the vertex number [7]. Because n>l in HMSC graph, the overall complexity is $\Theta(n)$.

The algorithm *CorrectInconsistentHMSCPath* checks the simple path in a m×(m-1)/2, where m is the number of bMSCs. The complexity of checking and of correction of the bMSC inconsistency is $O(n^4)$; therefore, the complexity is $O(m^2 \times n^4)$. If there are x loop paths; each path has y bMSCs; each bMSC has n events. The complexity is $O(x \times y \times n) = O(|z|^3)$, where |z| is the maximum number of x, y, and n. Therefore, the overall complexity of algorithm *CoorectInconsistentHMSCPath* is $O(m^2 \times n^4)$.

In summary, the complexity in the two steps of the correction of the inconsistency is $O(m^2 \times n^4)$, where m is the number of bMSCs, and n is the event number in a bMSC.

We take *HMSC H8* in the Figure 4.5 as an example to illustrate the algorithms. The *HMSC H8* is composed sequentially with *bMSCs* N_{12}, N_{13}, N_{14}, and N_{15}. We apply the checking and correction algorithms as follows.

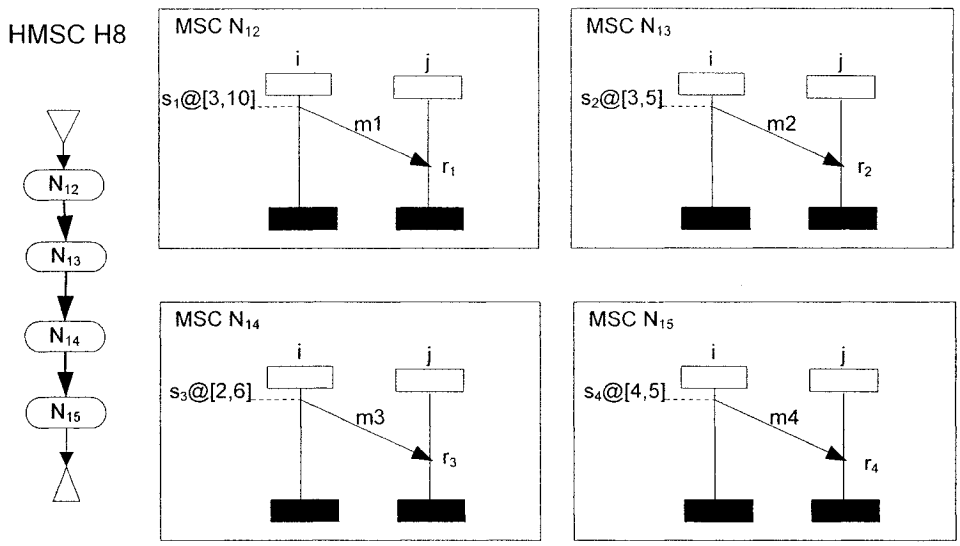


Figure 4.5 An HMSC with a flow path

Step 1 checks the consistency of all the bMSCs based on the algorithm in *Chapter 3*.

Step 1: Check&Correct_bMSCs

bMSC N_{12} is consistent.

bMSC N_{13} is consistent.

bMSC N_{14} is consistent.

bMSC N_{15} is consistent.

In Step 2, the HMSC is traversed to find all paths. The path consistency is checked based on the type of the path.

Step 2: Traverse_Graph_To_Find_Path

Sequential_Simple_Path[1] = {$N_{12}N_{13}N_{14}N_{15}$}.

CorrectInconsistentHMSCPath.

Sequential_Simpe_Path[1]= {$N_{12}N_{13}N_{14}N_{15}$} is not consistent.

{$N_{12}N_{13}$} is consistent.

{$N_{13}N_{14}$} is consistent.

{$N_{14}N_{15}$} is consistent.

{$N_{12}N_{13}N_{14}$} is consistent.

{$N_{13}N_{14}N_{15}$} is consistent.

{$N_{12}N_{13}N_{14}N_{15}$} is not consistent.

After doing an analysis of the results, the designer decides to correct the *MSC N12* or *MSC N15* and gets the following results:

Solution 1: If the MSC designer decides that bMSC N_{12} is changeable, the algorithms will calculate that bMSC N_{12} for event s_1@[1, 2], r_1@[2, ∞).

Solution 2: If the MSC designer decides that bMSC N_{15} is changeable, the algorithms will calculate that bMSC N_{15} for event s_4@[6, ∞), r_4@[7, ∞).

4.4 Discussion

There are several causes for time inconsistency in HMSCs. It is necessary to ensure an HMSC strongly consistent before it is used in system design and implementation. In order to change weakly consistent and inconsistent HMSCs so that they become strongly consistent, we have to find all the inconsistent paths or sub-paths.

An HMSC can be divided into loop paths, combinational-flow-loop path, and flow paths. In loop paths and combinational-flow-loop paths, the upper bounds of all the events in the loop(s) have to be infinite in order to keep the loop consistent. In the flow paths, the sub-simple paths have to be checked to detect all the potential bMSC nodes that cause the inconsistency. The correction strategies are then applied to the bMSCs on the paths.

Our approach can give a recommendation to make the HMSC strongly consistent. The tool MSCTICC implements HMSC correction algorithms based on the approaches proposed in this chapter.

Chapter 5

The tool MSCTICC and case studies

5.1 The tool MSCTICC overview

The algorithms described in the previous sections have been implemented as a tool named MSCTICC in C++. We reused the work of MSC2SDL of the Telesoft Research Lab at Concordia University. The parts of the works that have been reused are the bMSC and HMSC parser, the event builder, and the event-order-table builder [24].

We use the MSC-2000 standard textual formats to represent the MSC specifications with time constraints. The input specifications are either bMSCs or HMSCs. The main modules and sub-modules are shown in Figure 5.1.

The functionalities of the main module and sub-module are as follows:

(1) Dispatcher: distributes input of bMSCs or HMSCs into the proper processing path according to the type of the MSC.

(2) bMSC Processor: analyzes bMSCs.

- bMSC Parser: checks the syntax of the input bMSCs.
- Event Builder: builds the internal presentation of the bMSC events.
- Enhanced-event-order-table Builder: builds an enhanced event-order-table with time constraints.

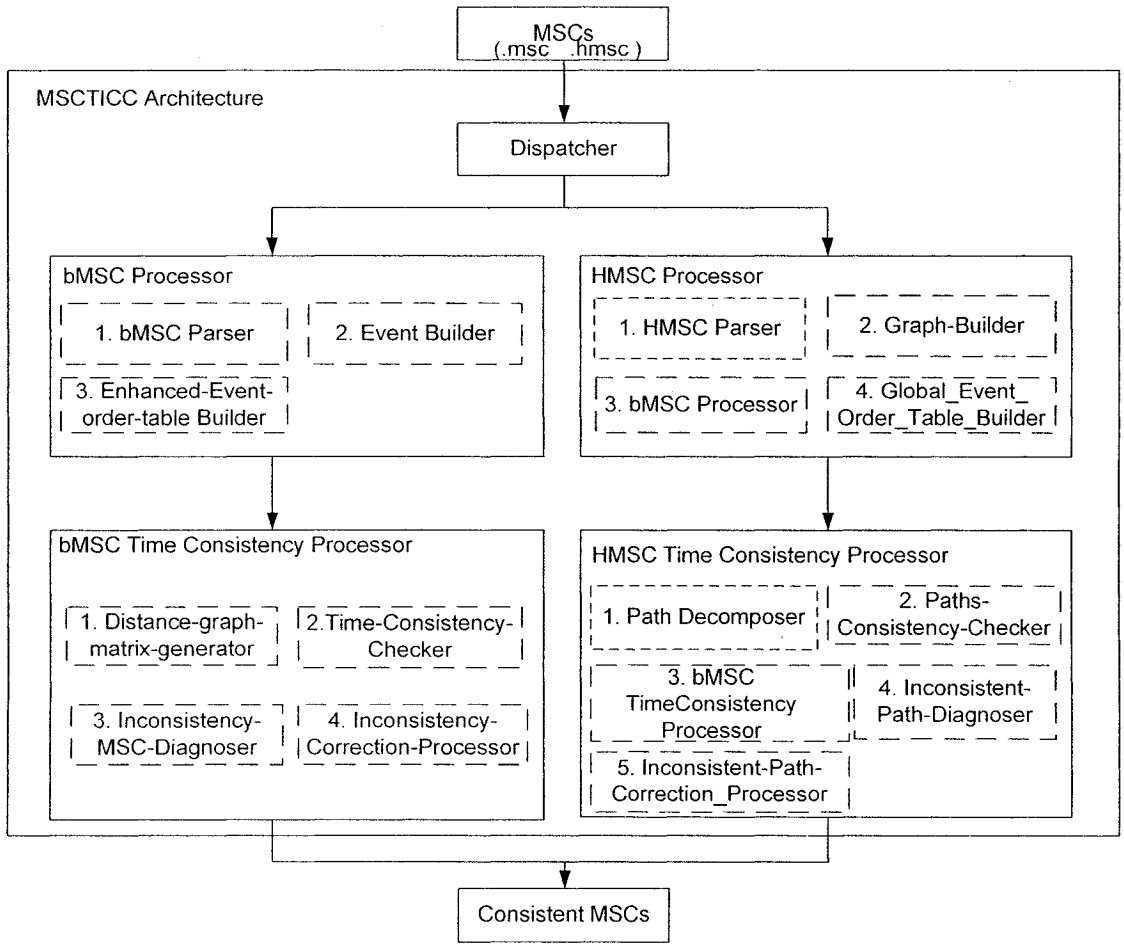


Figure 5.1 The MSCTICC tool architecture

(3) bMSC Time Consistency Processor

- Distance-graph-matrix-generator: generates a distance graph matrix based on the enhanced event-order-table.
- Time-consistency-checker: applies the Floyd-Warshall algorithm on the Distance-graph-matrix.
- Inconsistency-bMSC-diagnoser: lists all the inconsistent cycles in the bMSC for a system designer to decide correction policies.
- Inconsistency-correction-handler: applies different policies to the bMSC until it finds a consistent bMSC.

(4) HMSC Processor:

- HMSC Parser: checks the syntax of the input HMSCs.
- Graph Builder: builds the internal presentation of HMSCs.
- Global Event Order Table: merges individual event order tables into a global one.

(5) HMSC Time Consistency Processor:

- Path Decomposer: decomposes an HMSC into different paths.
- Path-consistency-checker: checks the consistency of each path according to the type of the path, such as a loop path or a sequential path.
- Inconsistency-path-diagnoser: Lists the inconsistent loop paths and the sub-paths of each sequential path for the designer to apply the correction policies.
- Inconsistency-correction-handler: applies the change policy to the bMSCs of different paths until the HMSC becomes a consistent HMSC.

The execution flows are described as follows.

First of all, the Dispatcher analyzes the input type files. Then,

(1) If the input is a basic MSC file, the control flow goes to the bMSC Processor. The bMSC Processor parses the inputs, checks the syntax of bMSCs, and then builds the internal presentation of MSC events and the Enhanced-event-order-table. Next, the system transforms the table into a distance graph matrix and applies Floyd-Warshall algorithms to check if the bMSC is consistent. If it is not, the system lists all the negative cycles of the system to provide a decision choice for the system designer. The designer

then applies different correction policies to the distance graph matrix and checks the consistency on the new bMSC recursively until the bMSC becomes consistent.

(2) If the inputs are a high level MSC file and a set of basic files, the Dispatcher module sends them to the HMSC parser, which checks the syntax of the HMSC. The HMSC Processor builds the internal presentation of the HMSC as a graph and then calls the bMSC Processor to process each referenced bMSCs as described in bMSC processing. Next, the HMSC Processor combines individual enhanced-event-order-tables into a global one. The system decomposes the specification into different paths and checks the consistency of each path. If there are some paths that are not consistent and if the path has a set of bMSCs within a loop, the corrections are applied to the absolute time constraints of all events within the loop; if the path is a sequential path, the designer checks the errors message and applies the correction policy on some bMSCs until the path becomes consistent.

5.2 Case studies

We present two applications in this section.

5.2.1 Test case 1 – The automatic call back service

In this case, we study one feature of interaction IN service. Automatic Callback allows a user to easily call another user who is often on the phone. When a caller dials another internal extension and finds it busy, the caller dials some digits on his phone or presses a

special *Automatic Callback* button. The caller's phone will try to connect automatically to the other phone in a loop way. When the called person hangs up, the phone will give a message to the caller's phone, and the caller gets a special ring. This feature saves a lot of time by automatically retrying the call until the extension is free.

The time constraints are specified in the HMSC. In the *MSC try1*, the sending event *e1* should happen in the absolute time constraint [3, 10] and the receiving event should happen in [4,12]. The MSC busyresponse1 specifies that the first send event *e1* should be in the time range [3, 5]. In the *MSC Aonhook*, in instance *A*, the second sending event *e2* should be in the time constraint [2, 6], the receiving event is in [3,8]. The *MSC Ring* specifies that the receiving event *e3* in instance *A* should be at [4, 5], the sending event *e4* is in [6,10].

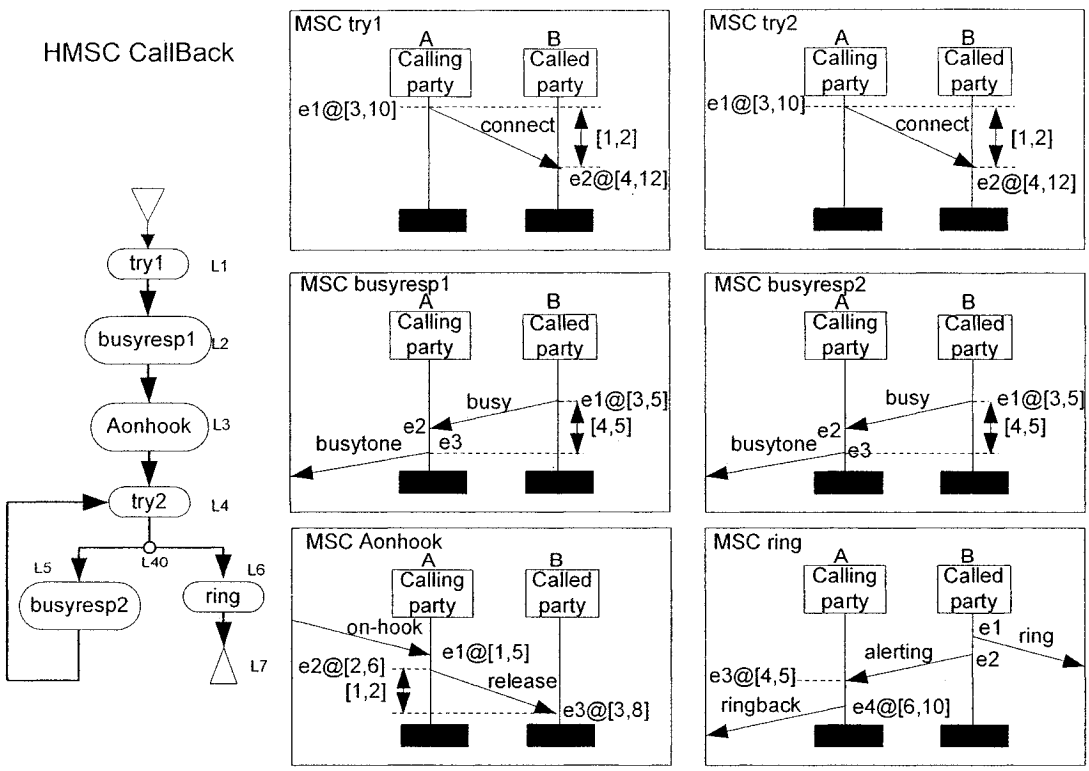


Figure 5.2 A test case for HMSC *CallBack*

The running result is shown in Table 5.1.

Table 5.1 The running result of HMSC *CallBack*

```
(1)   Checking HMSC time consisyency:
(2)   Path 0 is: try1 busyresp1 Aonhook try2 connect ring  is not consistent.
(3)   The current path is inconsistent.
(4)   Under the current path,Before&InLoopPath 0 is: try1 busyresp1 Aonhook try2 busyresp2  is not
        consistent.

(5)   ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
(6)   Conclusion: The HMSC: callback.hmsc is inconsistent.
(7)   ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

(8)   The HMSC is not stronly consistent.
(9)   Please follow the following message to correct the specification.

(10)  In a loop, the upper bound of event e1 in bMSC try2 is not infinite.
(11)  Then, set the upper bound of event e1 in bMSC try2 to infinity.
(12)  In a loop, the upper bound of event e2 in bMSC try2 is not infinite.
(13)  Then, set the upper bound of event e2 in bMSC try2 to infinity.

(14)  In a loop, the upper bound of event e1 in bMSC busyresp2 is not infinite.
(15)  Then, set the upper bound of event e1 in bMSC busyresp2 to infinity.

(16)  In a after-loop, the upper bound of event e3 in bMSC ring is not infinite
(17)  Then, set the upper bound of event e3 in bMSC ring to infinity.
(18)  In a after-loop, the upper bound of event e4 in bMSC ring is not infinite
(19)  Then, set the upper bound of event e4 in bMSC ring to infinity.

(20)  HMSC simple path: try1 busyresp1 Aonhook  is not consistent!!!!!!!
(21)  HMSC simple path: busyresp1 Aonhook  is not consistent!!!!!!!

(22)  If press 'q' to quit, or Input the bMSC name that can be changed: Aonhook
(23)  If press 'q' to quit, or Input the bMSC name that can be changed: q

(24)  The recommended bMSC Aonhook event e1 is @[12,inf]
(25)  The recommended bMSC Aonhook event e2 is @[13,inf]
(26)  The recommended bMSC Aonhook event e3 is @[14,inf]
(27)  The recommended bMSC Aonhook event  in env is @[11,inf]

(28)  The HMSC is strongly consistent now.
```

The system decomposes the HMSC into different simple paths. After checking all simple paths, it finds that some of the paths are inconsistent. We get a simple path {*try1 busyresp1 Aonhook try2 connect ring*} is not consistent in line *2*, and a combinational-

flow-loop path {*try1 busyresp1 Aonhook try2 busyresp2*} is not consistent in line *4*. Therefore, we get the conclusion that the HMSC is inconsistent as indicated in line *6*.

The system traverses the HMSC to find all the bMSCs, and then checks all the bMSCs, and does not find any that are inconsistent. Therefore, the system moves to check and correct the paths' inconsistency.

The system finds a loop {*try2 busyresponse2*}. It finds that the upper bounds of events *e1 and e2* in *MSC try2,* and *e2* in *bMSC busyresponse2* are not infinite. Therefore, it sets these upper bounds to infinity. It also finds that the upper bound of event *e3* and *e4* in *bMSC ring* is not infinite. Because the *bMSC ring* follows the loop path {*try2 busyresponse2*}, the upper bounds have to be set to infinity. These messages are clearly indicated from line *10* to *19*.

Then, the simple path {*try1 busyresponse1 Aonhook try2 ring*} is checked. From the diagnostic messages of line *20* and *21*, the MSC designer can easily decide that the *bMSC Aonhook* causes the inconsistency in this path. The designer inputs the bMSC name into the correction procedure in line *22* and *23*. The system obtains the time constraints of the events of bMSCs in line *24* to *27*. Then, the system concludes the HMSC is consistent as indicated in line *28*.

5.2.2 Test case 2 – A communication setup protocol

This case describes a communication protocol involving the connection request, confirm and communication process. When a system is in *Disconnected* state, a connection request is initiated. As described in *bMSC Connection_request,* instance *i* sends request message m_1 to instance *j*, and the *instance j* sends back an acknowledge message m_2 to instance *i*. The absolute time constraints are specified for the four events. The absolute time of send event E_1 for message m_1 is @[3,4], and the absolute time of receive event E_1 for message m_1 is @[4,5]. The absolute time of send event E_4 for message m_2 is @[6,7], and the absolute time of receive event E_2 for message *m1* is @[1,2].

In the *Wait_for_Response* state, the connections is confirmed. In the MSC *Connection_Confirm*, the instance *i* sends two confirm messages m_1 and m_2 in a relative time &[1,1]. For the second message m_2, the send and receive time constraint is &[2,2]. The relative time for send event E_1 of the first message and the receive message of the second message m_2 is &[2,2].

Then the system is in the *Connected* state, and communications begin. As *MSC Communication* describes, the *instance i* sends packages m_1 and m_3 to *instance j*, and receives messages m_2 and m_4 from *instance j*. The relative time constraints are specified in Figure 5.3.

After applying the MSCTCC tool, we get the following result.

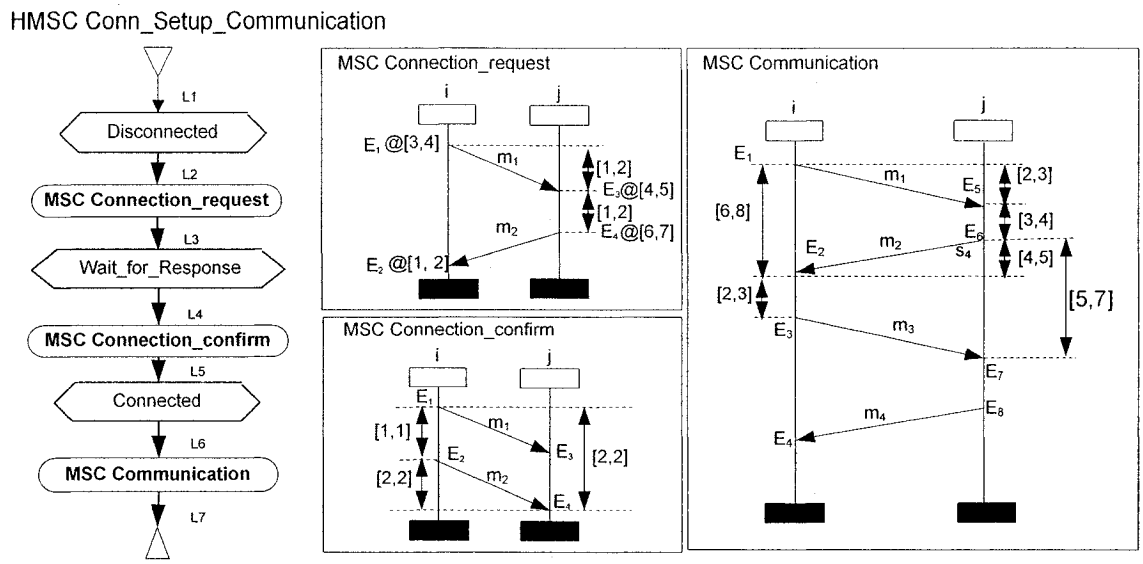


Figure 5.3 A test case for HMSC *Connnection_Setup&Communication*

Table 5.2 The running result of HMSC *Connnection_Setup&Communication*

(1)	Checking HMSC time consisyency:
(2)	Path 0 is: Connection_request Connection_confirm Communication is not consistent.
(3)	The current path is inconsistent.
(4)	++
(5)	Conclusion: The HMSC: Conn_Setup_Communication.hmsc is inconsistent.
(6)	++
(7)	The HMSC is not stronly consistent.
(8)	Please follow the following message to correct the specification.
(9)	Cycle E2 E1 E0 E2 is inconsistent
(10)	Cycle E3 E0 E2 E3 is inconsistent
(11)	Cycle E4 E0 E2 E4 is inconsistent
(12)	Basic MSC Connection_request is not consistent. Please correct it.
(13)	Please input 'q' to quit, or input a policy number(0,1,2,3,4): 1
(14)	Input the instance name to apply Policy 1: i
(15)	The events in this instance are E1 E2
(16)	Please input the event name with absolute time constraint in this instance: E2
(17)	The recommendation for the correction is:
(18)	The absolute time constraint of event E2 is @[7,inf)
(19)	Cycle E4 E2 E1 E4 is not consistent.
(20)	Basic MSC Connection_confirm is not consistent. Please correct it.
(21)	Please input 'q' to quit, or input a policy number(0,1,2,3,4): 4
(22)	Input the first event name: E1
(23)	Input the second event name: E4
(24)	The relative time constraint between events E1 and E4 &T[E1,E4] is [3,3].
(25)	Cycle E6 E5 E1 E2 E6 is not consistent.
(26)	Cycle E7 E3 E2 E6 E7 is not consistent.
(27)	Basic MSC Communication is not consistent. Please correct it.

```
(28)  Please input 'q' to quit, or input a policy number(0,1,2,3,4): 2
(29)  Input the instance name to apply Policy 2: i
(30)  The events in this instance are E1 E2 E3 E4
(31)  Input the first event name: E1
(32)  Input the second event name: E2
(33)  The recommended relative time between events E1 and E2 &T[E1,E2] is [9,12]
(34)  Cycle E7 E3 E2 E6 E7 is not consistent.
(35)  Still inconsistent. Continue to apply Policy 1, 2, 3, or 4.
(36)  Please input 'q' to quit, or input a policy number(0,1,2,3,4): 3
(37)  Input the message name to apply Policy 3: m2
(38)  The send event name   : E6
(39)  The receive event name: E2
(40)  The last recommendation to apply Policy 2 correction is as following:
(41)  The relative time constraint between events E6 and E2 &T[E6,E2] is [2,3].

(42)  The HMSC is strongly consistent now.
```

The HMSC is decomposed into only one flow path. The time consistency checking algorithms obtain an error message, which indicates that the flow path {*Connection_request Connection_confirm Comminication*} is not consistent as indicated in line *7*.

The system traverses the HMSC to find all the bMSCs, and then checks all the bMSCs, and finds some inconsistent bMSCs and begins to correct them.

The system, then, checks all bMSC nodes one by one. The causes for the inconsistency are listed in line *9*, *10* and *11* for bMSC *Connection_request.* The MSC designer decides to apply bMSC correction Policy 1 to the bMSC on instance *i* and on the events E_2. Then the system applies correction algorithms and obtains the events absolute constraints in line *18*. Then, the *MSC Connection_request* is consistent.

The system continues to check the next bMSC and ascertains that the *MSC Connection_request* is not consistent. The cause of the inconsistency is given in line *19*. The MSC designer decides to apply bMSC correction Policy 4 on events E_1 and E_4. Therefore, the system gives the correction time constraints on the relative time constraint between E_1 and E_4 in line *24*.

The system detects that the bMSC *Communication* is not consistient. The cause is listed in lines *25* and *26*. The MSC designer decides to apply bMSC correction Policy 2 to events on instance *i* between *E1* and *E2*. Therefore, the recommended relative time constraints between *E1* and *E2* are listed in line *33*. However, the system gives another inconsistent cycle as indicated in line *34*. The designer decides to apply Policy 3 to the message *m2*. The designer gets the recommendation of the relative time constraint on sending and receiving events as indicated in line *41*.

The system checks the simple of HMSC and finally gets a consistent specification as indicated in line *42*.

5.3 Strengths and limitations of the tool

In the above two cases, we apply the tool MSCTICC in the following way.

First, we check the consistencies of all the bMSCs. If a bMSC is not consistent, we can track back the negative cycles existing in the bMSC. According to the causes in the

bMSC, we apply four policies or the combination of the policies to correct the bMSC until we get consistent bMSC.

Then, we check the HMSC path consistency. If there is a loop path or a combinational-flow-loop path, the tool will set all the upper bounds of the events in the loop to infinity. A simple path is then checked with an algorithm to ascertain if the simple path is consistent. All the sub-paths are checked and the possible inconsistent bMSC nodes are listed for the designer to give the correction possibilities. The designer then relaxes the absolute time constraints of the bMSCs, and a path with new timing requirements is applied by the checking algorithm until the path becomes consistent.

In this way, all the bMSCs and all the loops and sequential paths are made consistent. The designer gets a recommendation about how to change the MSC specification to meet the timing requirement.

There are strengths and limitations:

Strengths:

1. The tool handles the bMSC relative and absolute time constraints. It handles inline expressions including *seq*, *alt*, *loop*, and *opt*.
2. The tool handles the HMSC with *seq*, *alt*, and *loop* operators.
3. The tool can check the consistency of bMSCs and HMSCs,
4. The tool gives the causes for inconsistency and initiates policies for correction,

5. The consistent MSCs output would be an input for the MSC to SDL translation.

Limitations:

1. The tool inherits the limitation of the previous works from the TeleSoft Group. That is, this tool does not allow declarations in bMSCs. Thus it requires an *msc* document to be included in the input MSC files;
2. Time constraints associated with orderable events other than input/output are not implemented in this version.
3. In bMSCs, no nested inline expression is allowed.
4. Multi-instances for one process are not handled in this tool.
5. Only time constraints inside bMSC are considered, but the time requirement between the bMSCs is not considered.
6. Timer in bMSCs is not considered in the current work.

Chapter 6

Conclusions

6.1 Contributions

In this thesis we have developed a tool for time consistency checking and inconsistent MSCs diagnostics and correction.

A bMSC specification is first described as an enhanced-event-order-table with time constraints, and then transformed into a distance graph matrix. The simple temporal order theory is applied to the checking process for consistency analysis based on the distance graph matrix.

The different temporal orders and time constraints have been defined and various criteria are analyzed to find the causes for time inconsistencies for basic MSCs. By applying different diagnostic methods to the matrix of the distance graph, we can trace back all the negative cycles in the distance matrix. Therefore, we can exhibit which part of the bMSCs cause the time inconsistency.

Four correction policies have been proposed for the MSC designer. If the MSC designer decides to change an absolute time constraint on an instance, Policy 1 is applied. If the relative time constraint on an instance is changeable, Policy 2 can be applied. If the relative time constraints on the message pair are changeable, Policy 3 is applied. If any

other two events' relative time constraints can be changed, we can apply Policy 4. The algorithms for the correction processes are illustrated in the thesis.

As for the HMSCs, they are decomposed into different simple paths. We propose an approach to differentiae different simple and corresponding algorithms to check the time consistency path by path. According to the type of the simple path, different correction policies can be applied to correct the paths until consistency is reached.

We provide a very innovative and promising tool based on approaches discussed in the thesis. The tool can help the MSC designer to obtain a consistent MSC specification. This way, system development can be made much easier, and errors can be detected and fixed at every stage of the software development.

6.2 Future work

Our approaches so far cover only a subset of the MSC-2000 specifications. There is still further research to be done. This may cover:

- Allowing MSC references in bMSCs, HMSC references in HMSC, etc. *Parallel* operators of inline expressions and HMSC would also be interesting extensions. Further, the *exc* inline expression may be further developed.
- External date type: Current work allows only for limited date types and data manipulation, such as integer, time and simple operations. Complex systems need

more advanced data types and operations that may be defined in other languages, such as C and ASN.1. Handling imported data definitions and functions in MSC will make our approach more powerful.

- HMSC specification with time constraints in a finite loop is not handled by our approach. Further investigations need to be carried out.
- The consistency of time constraints in MSCs raises new implementability issues. Further work should be done to investigate whether other kinds of time constraints can be guaranteed under a given architecture.
- The time constraints between bMSCs in an HMSC should be explored in future work.
- More complicated cases with time requirements need to be tested.

Bibliography

[1] R. Alur and D. L. Dill: A theory of timed automata, Theoretical Computer Science, Vol. 126, Number 2, pp. 183-235, 1994.

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

[3] R. Alur, K. Etessami and M. Yannakakis: Inference of Message Sequence Charts, 22nd International Conference on Software Engineering, pp. 304-313, 2000.

[4] J. Cameron, N. D. Griffeth, Y. J. Lin, M. E. Nilson, W. K. Schnure, H. Velthuijsen: A Feature Interaction Benchmark for IN and Beyond. Feature Interactions in Telecommunications Systems, ISO press, pp. 1-23, 1994.

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

[6] T. H. Cormen. C. E. Leiserson, R. L. Rivest: Introduction to Algorithms, second edition, MIT Press, pp. 120-134, 2001.

[7] G. Cugola, C. Ghezzi: Software Processes: A retrospective and a path to the future, Software Process: Improvement and Practice, Vol.4, No.3, 1998.

[8] R. Dechter, I. Lmeiri, J. Pearl: Temporal Constraint networks, Artificial Intelligence 49, pp. 61-95, 1991.

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

[10] ITU-T, Message Sequence Chart – MSC-2000, ITU-T Recommendation Z.120, November 1999.

[11] ITU-T, Specification and Description Langage – SDL-2000, ITU-T Recommendation Z100, November 1999.

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

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

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

[15] S. Leue and H. Ben-Abdallah, MESA: Support for scenario-based design of concurrent systems. In Proc. 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1384, pages 118-135, 1998

[16] V. Levin, D. Peled: Verification of Message Sequence Charts via Template Matching, FORTE, pp. 139-154, 2000.

[17] Z. Manna and A. Pnueli : The Temporal Logic of Reactive and Concurrent Systems, Specifications, Springer-Verlag, pp. 150-180, 1992.

[18] Z. Manna and A. Pnueli: Modes for Reactivity, Acta Informatica, 30, pp. 609-678, 1993.

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

[20] K. H. Rosen: Discrete Mathematics and its Applications, McGraw-Hill, 5-th edition, ISBN 0-07-242434-6, 2003.

[21] C. A. Shaffer: A Practical Introduction to Data Structures and Algorithm Analysis, Second Edition, Prentice Hall, p360-361, 2000.

[22] J. M. Wing: A Specifier's Introduction to Formal Methods, IEEE Computer, 23(9), pp. 8-24, 1990.

[23] L. X. Wang: Implementation of Time Consistency of MSC-2000 Specifications, Internal Report for NSERC Undergraduate Research, Department of Electrical and Computer Engineering (ECE), Concordia University, Summer 2003.

[24] X. J. Zhang: The MSC2SDL2004 Tool User Guide, Department of Electrical and Computer Engineering, ECE, Concordia University, Summer 2004.

[25] X. J. Zhang: Automatic Generation of SDL Specifications from Timed MSCs, M.A.Sc. Thesis, Department of Electrical and Computer Engineering, Concordia University, 2004.

[26] T. Zheng, F. Khendek: Time Consistency of MSC-2000 Specifications, Computer Networks, Volume 42, Issue 3, pages 303-322, 2003.

[27] T. Zheng: Validation and Refinement of Timed MSC Specifications. PhD thesis, Department of Electrical and Computer Engineering, Concordia University, 2004.

Appendix A

Textual Syntax of a Simplified 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>|

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>]{ ']' | ')'}

<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

B.1 Calculate deduced order time constraints from directed order time constraints

Case 1: The local directed order time constraints and the matching pair directed order time constraints are all specified. For a constraint graph in Figure B.1, assume that there are (n+1) events connected by directed order s. Directed order time constraints are $V[e_1,e_2]=[l_1,u_1]$, $V(e_2,e_3)=[l_2,u_2]$, ..., $V(e_n,e_{n+1})=[l_n,u_n]$. Let $I(e_1, e_{n+1})=[x,y]$ be the deduced order time constraint(see Figure B.1(a)).

The lower bound of the deduced order time constraint can be calculated as the sum of all the lower bounds of all the directed order time constraints, whereas, the upper bound of the deduced order time constraint can be calculated as the sum of all the upper bounds of all the directed order time constraints.

According to *Theorem 3.1*[8] we get following inequations:

$u_1+u_2+....+u_n + (-x) \geq 0$ *(1)*

$y+(-l_1)+(-l_2)+...+(-l_n) \geq 0$ *(2)*

(1): $x \leq (u_1+u_2+....+u_n)$ *(see Figure B.1(b))*

(2): $y \geq (l_1+l_2+....+l_n)$ *(see Figure B.1(c))*

Inequations (1) and (2) stand for: $I(e_1, e_{n+1})=[(l_1+l_2+....+l_n), (u_1+u_2+....+u_n)]$.

Case 2: The input indirect causal order and some local directed and matching pair directed order time constraints are specified. Let $V(e_k,e_{k+1})=[l_{k+1},u_{k+1}]$, with $k\in[1,n-1]$, be the directed order time constraints (see Figure B.1(a)). Assume that these directed order time constraints are given by the designer, except $V(e_{i-1},e_i)=[l_i,u_i]$, which remains an unknown time constraint. Moreover, assume that the directed order time constraint $I(e_1,e_{n+1}) = [x,y]$ is also given.

According to *Theorem 3.1* in the paper [8]:

$$u_1+u_2+\ldots u_i\ldots+u_n+(-x)\geq 0 \qquad (1)$$

$$y+(-l_1)+(-l_2)+\ldots(-l_i)+\ldots+(-l_n)\geq 0 \qquad (2)$$

So: (3): $x-(u_1+u_2+\ldots u_{i-1}+u_{i+1}.+u_n)\leq u_i$ *(see Figure B.1 (b))*

(4): $l_i\leq(l_1+l_2+\ldots.+l_n)-y$ *(see Figure B.1(c))*

Theses two inequations stand for the $I(e_i, e_{i+1})=[x-(u_1+u_2+\ldots u_{i-1}+u_{i+1}.+u_n),(l_1+l_2+\ldots.+l_n)-y]$.

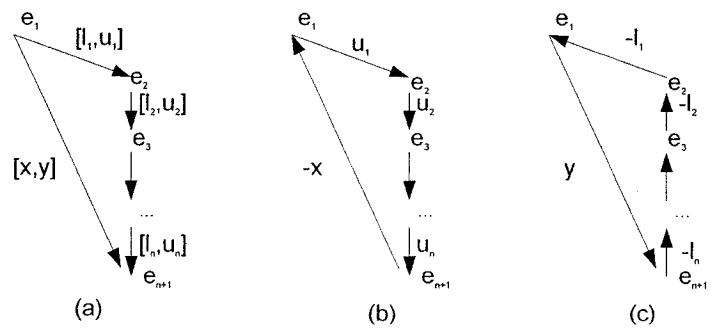


Figure B.1 Connected directed order time constraints and the deduced order

We can apply the All-pairs-shortest-path algorithm along the specified paths formed by these events $\{e_1,e_2,e_3,...e_{n+1}\}$. For example, let $I(e_1, e_{n+1})=[x,y]$ be the deduced order time constraint(see Figure B.1(a)). When we apply the All-pairs-shortest-path algorithm along path $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \quad ... \rightarrow e_n \rightarrow e_{n+1}$, we achieve the shortest path between any events in the distance graph; therefore, we get the reduced relative time constraint between e_1 and e_{n+1}. This procedure is also applicable to Case 2. We can get any deduced order time constraint based on the other path's time constraint by applying the All-pairs-shortest-path algorithm. [End of Cases]

B.2 Proof for Proposition 3

Proof of "overlapping between visual and causal order implies bMSC consistency".

Let x and y be two events, and let x causally happens before y (see Figure B.2 (a)). Let $V(x,y)=[l_1,u_1]$ be the directed order time constraint between x and y. Let $I(x,y) = [l_2,u_2]$ be one of the corresponding deduced order time constraints . If there is an overlapping between $[l_1,u_1]$ and $[l_2,u_2]$ (i.e. $[l_1,u_1]\cap[l_2,u_2]\neq\varnothing$), then $u_1 \geq l_2$ and $u_2 \geq l_1$. So, $u_1+(-l_2) \geq 0$ and the corresponding cycle (see Figure B.2 (b)) is a positive cycle. Moreover, $u_2 + (-l_1) \geq 0$ and the corresponding cycle (see Figure B.2 (c)) is also a positive cycle. According to the Theorem 3.1 [8], a given simple temporal problem (STP) T is consistent if and only if its distance graph G_d, has no negative cycles. In the same way, if $[l_1,u_1]$ is one of the deduced order time constraints, the above inequalities still hold. Therefore, if all the

directed order time constraints and the corresponding deduced order time constraints are overlapping, we can conclude that the bMSC is consistent, and vice-versa.

Proof of "bMSC consistency implies overlapping between visual and causal order, and between deduced orders".

If the bMSC is consistent, then there is no negative cycle according to Theorem 3.1[8]. Considering the directed order and the corresponding deduced order time constraint, it means $u_1 + (-l_2) \geq 0$ and $u_2 + (-l_2) \geq 0$. This condition means that the directed order time constraint and the corresponding deduced order time constraint are overlapping.

Let us assume that $[l_1,u_1]$ is the directed order time constraint, $[l_2,u_2]$ is the deduced order time constraint. Then the overlapping time constraint is $[l_2,u_1]$ as shown in Figure B.2 (d) is the reduced time constraint. Because $l_2<u_1$ as the assumption, $u_1+(-l_2)>0$. There is no negative cycle exist in the distance graph. Therefore, the bMSC is consistent. [End of Proof]

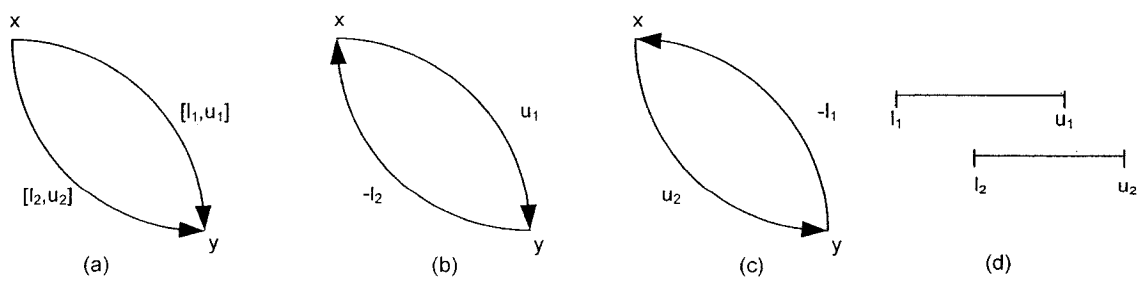


Figure B.2 Directed order , deduced order time constrain and their distance graph

B.3 Proof of Proposition 4

Proof: By *Theorem 3.3* (Decomposability) [8], any consistent Simple Temporal Problem is decomposable into substructure based on the constraints in its distance graph. We conclude that, a bMSC as a STP distance graph can be decomposed into sub-graph cycles and further consider theses cycles' consistency. According to Theorem 3.1[8], a given STP is consistent if and only if its distance graph has no negative cycles. In other word, the bMSC is consistent if and only if all these cycles are consistent.

Let a bMSC have n events, every two events can form a cycle, so there are C(n,2) cycles; every three events can form two cycles, so there are C(n,3) cycles; every four events can form two cycles, so there are C(n,4) cycles ... every n events can form two cycles, so there are C(n,n) cycles. Therefore, we may have $(\sum_{k=0}^{n} C(n,k)$–n-1) cycles. By theorem $\sum_{k=0}^{n} C(n,k) = 2^n$ [10], we can get: $(2^n - n - 1)$. [End of Proof]

B.4 Proof of Proposition 5

Proof: Let's consider a bMSC and its corresponding complete distance graph G=(V,E), where:

$V=\{e_0, e_1, e_2, e_3, \ldots, e_n\}$,

$E=\{(e_0,e_1), (e_0,e_1), \ldots (e_0,e_n), (e_1,e_2),(e_1,e_3),(e_1,e_4)\ldots(e_1,e_n), (e_2,e_1)(e_2,e_3)\ldots(e_2,e_n),\ldots.(e_n,e_1), (e_n,e_2)\ldots(e_n,e_{n+1})\}$

Relative time contraints are:

$T(e_1,e_2)=w_{12}=u_{12}^{(0)}$, $T(e_2,e_1)=w_{21}=-l_{21}^{(0)}$,

$T(e_1,e_3)=w_{13}=u_{13}^{(0)}$, $T(e_3,e_1)=w_{31}=-l_{31}^{(0)}$,

…. ,

$T(e_1,e_n)= w_{1n}=u_{1n}^{(0)}$, $T(e_n,e_1)= w_{n1}=-l_{n1}^{(0)}$,

….,

$T(e_{n-1},e_n)=w_{(n-1)n}= u_{(n-1)n}^{(0)}$, $T(e_n,e_{n-1})=w_{n(n-1)} = -l_{n(n-1)}^{(0)}$.

Let adjacency matrix $W=(w_{ij})$ represents the graph.

Also, let $d_{ij}^{(k)}$ be the weight of the shortest path from vertex i to vertex j going through intermediate vertices chosen in the set $\{1,2,..i-1,i+1,...j-1,j+2,...n\}$ by the k steps of the Floyd-Warshall algorithm. When k=0, the path from vertex i to vertex j has no intermediate vertex numbered higher than 0, and therefore has no intermediate vertices at all. Thus, $d_{ij}^{(0)}=w_{ij}$.

Let's define the Floyd-Warshall's algorithm [20] in a recursive way by :

Equation 1: $$d_{ij}^{(k)}= \begin{cases} min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}) \text{ if } k \geq 1 \\ w_{ij} \text{ if } k=0 \end{cases}$$

Further, let's assume condition: $d_{ii}^{(k)} <0$, and $d_{ii}^{(k-1)} \geq 0$, $\ldots d_{ii}^{(0)} \geq 0$ where $k>1$. For the pair (i, i) where $i \leq n$, let's consider all paths from i to i whose intermediate vertices are taken from V = $\{1,2,..k-1,k+1,...n\}$, and let p be a minimum-weight path from V;

furthermore, path p is simple, because we assume that weight of p is $d_{ii}^{(k-1)} \geq 0$ (Figure B.3 (a)).

We break path p down into $i \rightarrow k$ and $k \rightarrow i$ (see Figure B.3 (b)). By Lemma 25.1[8], sub-paths of shortest paths are also shortest paths. According to the lemma, p_1 is a shortest path from i to k. Similarly, p_2 is a shortest path from k to i. We can conclude from Equation 1 that $dii^{(k)} = d_{ik}^{(k-1)} + d_{ki}^{(k-1)} < 0$.

In the same way, we continue to break down p_1 and p_2 into further shortest paths p_{11}, p_{12} and p_{21}, p_{22} until that:

in path p_{11}, $d_{ix}=w_{ix}= d_{ix}^{(0)}$,

in path p_{21}, $d_{xk}=w_{xk}= d_{xk}^{(0)}$, and $d_{ky}=w_{ky}= d_{ky}^{(0)}$,

in path p_{22}, $d_{yj}=w_{yj}= d_{yj}^{(0)}$,

in path p_{23}, $d_{ji}=w_{ji}= d_{ji}^{(0)}$ as shown in Figure 10(c).

In conclusion, we can get $d_{ii}^{(k)}= d_{ix}^{(0)} + d_{xk}^{(0)} + d_{ky}^{(0)} + d_{yj}^{(0)} + d_{ji}^{(0)} < 0$, which means that there is cycle with negative cost. This inequation indicates that the path $i \rightarrow x \rightarrow k \rightarrow y \rightarrow j \rightarrow i$ is inconsistent.

The inconsistency sub-traces back-track searching process can be calculated based on the predecessor matrix. The predecessor matrix P is constructed on-line just as the D matrix is constructed. We compute a sequence of matrices $P^{(0)}$, $P^{(1)}$,…,$P^{(n)}$, where $P=P^{(n)}$ and

$p_{ij}^{(k)}$ is defined to be the processor of vertex j on the shortest path from vertex i with all intermediate vertices in the set $\{1,2,...,k\}$.

The recursive formulation $p_{ij}^{(k)}$ is given in Equation 2:

Equation 2: $p_{ij}^{(k)} = \begin{cases} p_{ij}^{(k-1)} & if\ ((d_{ij}^{(k-1)} \le (d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \ AND\ k \ge 1) \\ k & if\ ((d_{ij}^{(k-1)} > (d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \ AND\ k \ge 1) \\ \varnothing & if\ k=0 \end{cases}$

When k=0, a shortest path from i to j has no intermediate vertices at all. For $k \ge 1$, if we take $i \rightarrow k \rightarrow j$ as a shortest path from i to j, the processor of j is k. [End of Proof]

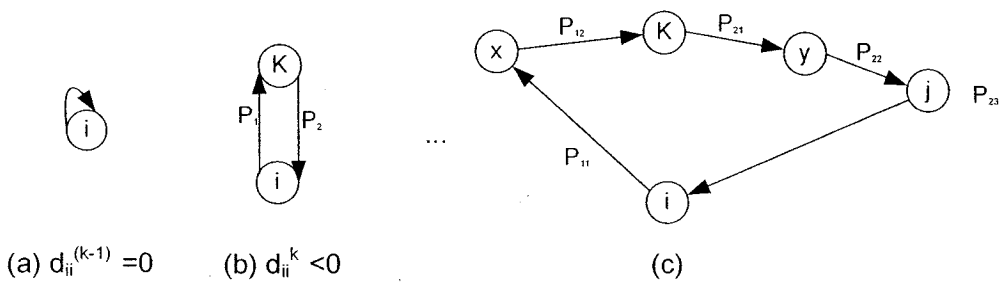


Figure B.3 The shortest path and its break down

B.5 Proof of Proposition 10

Proof: Let H=(S,D,L) be an HMSC, a simple path is sequential of nodes $s_0 s_1 ... s_n$ (n>0).

If part: If the HMSC simple path $s_0 s_1 ... s_n$ is not consistent, it means at least one sub-path $s_0 s_1 ... s_n$ is not consistent.

Only if part: If some sub-paths are inconsistent, because a sub-simple path is a prefix of a simple path, then the simple path would not be consistent.

If we find an HMSC simple path to be not consistent, we have to check all the sub-simple paths time consistency and figure out the inconsistent sub-simple paths. [End of Proof]

B.6 Proof of Proposition 11

Proof: Let there be n consistent *bMSCs* A_1, A_2, A_3, ..., A_n as Figure B.4. When these bMSCs are composed sequentially to form a simple path, the simple path is not consistent. From the Proposition 10, there must be some sub-simple paths that are not consistent. To find the causes for the inconsistency between any two bMSCs, we need to form sub-path between A_1 and A_2, A_2 and A_3, ..., A_{n-1} and A_n. Therefore, we have (n-1) sub-paths. Similarly, to form three bMSCs path, we have A_1 , A_2, and A_3 , A_2 , A_3, and A_4 ,..., A_{n-2}, A_{n-1}, and A_n, therefore, we have (n-2), and so on, until A_1, A_2, ... A_n form a simple path. Totally, we have (n-1) + (n-2) + (n-3) + (n-4) + ...+2 +1 = n(n-1)/2. [End of Proof]

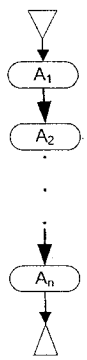


Figure B.4 A simple path with n bMSCs