Automatic Generation of SDL Specifications from Timed MSCs

Xiao Jun Zhang

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

Montréal, Québec, Canada

May 2004

Your file Votre référence
ISBN: 0-612-94720-3
Our file Notre référence
ISBN: 0-612-94720-3

Abstract

Automatic Generation of SDL Specifications from Timed MSCs

Xiao Jun Zhang

The integration of Formal Description Techniques (FDTs) in the software process enables formal validation, translation, synthesis and code generation. Message Sequence Charts (MSC) and Specification and Description Language (SDL) are two formal languages, widely used in the telecommunication industry. Generally MSC is used for the behavioral requirement specification, while SDL is used for the detailed design specification. The transition from the requirement specification to the design specification is usually performed manually; and the design has to be validated against the requirement specification.

In a previous research work, researchers from the telesoft group at Concordia University devised an approach for generating SDL specifications from MSC specifications with a given target architecture. It guarantees correctness of the design, and consistency between the SDL specification and the MSC specification. The need for validation has been eliminated.

Time concepts have been introduced in MSC-2000, which enables real-time requirements to be specified in MSC. Building on the existing framework, this thesis presents a new approach for translating MSCs with real-time requirements into SDL specifications. We analyzed and classified different types of time constraints and measurements. New algorithms for analyzing MSC specifications and generating SDL code were devised. We also built the tool and experimented with case studies to prove the feasibility of our approach.

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Mr. Xiao Jun Zhang

Entitled: "Automatic Generation of SDL Specifications from Timed MSCs"

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

Master of Applied Science

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

Signed by the final examining committee:

M.R. Soleymani Dr. M.R. Soleymani	Chair
[illegible] Dr. P. Grogono	Examiner, External to the Program
[illegible] Dr. R. Dssouli	Examiner
[illegible] Dr. A. En-Nouaary	Examiner
[illegible] Dr. F. Khendek	Supervisor

Approved by: M. O. Ahmad
Dr. M.O. Ahmad
Chair, Department of Electrical and Computer Engineering

JUN 14 2004
________ 20____

[illegible]
Dr. Nabil Esmail
Dean, Faculty of Engineering and Computer Science

Acknowledgements

I would like to express my heartfelt thanks to my supervisor, Dr Ferhat Khendek, for his guidance, encouragement and patience. He has always been there openly to discuss problems and to provide invaluable suggestions. Dr. Khendek has motivated me so much with his dedication to this work.

I wish to express my profound gratitude to my parents, Yu Chun and Sheng. Without their enormous love, understanding and supports, I would have never made it this far. I dearly thank my sisters, Xiao Li and Xiao Hua. I know I can always count on their support and encouragement. I am forever grateful to my family.

I also want to thank all teachers who have taught me in my degree program and other colleagues at Concordia University for their valuable help and cooperation. My thanks also go to all friends who have helped me along my study. Without them, today's accomplishments would not have been possible.

This research work has been supported by a postgraduate scholarship from the Natural Sciences and Engineering Research Council of Canada, the Concordia University Graduate Fellowship and a funding from France Telecom. Their support is most gratefully acknowledged.

Dedication

To my parents, Yu Chun and Sheng.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Software process is referred to as the sum of all software activities, encompassing the entire software life cycle [13]. This process involves a collection of principles, methodologies, methods and techniques. There are major phases in most software process models, namely analysis and requirement, design, implementation, testing and maintenance. Each phase has its own goal and defined products. In the analysis and requirement phase, software engineers gather requirements from the clients/users, specify software features and also prioritize them. The output of this phase is a system level document describing what the software system does, its constraints and performance requirements. Usually high level test cases may also be obtained in this phase as well. The document may be written in an informal language or in a formal language. The requirement documents are used as input of the design phase. This phase basically defines a solution to the requirements. It includes an architecture of the software system, modules in the system, and interfaces between the modules. In the implementation phase, the design is coded in a concrete programming language. Testing comes as the next phase; the produced software is tested to ensure its conformance to the functionality and performance requirements specified in the requirement documents. Maintenance is conducted during the entire software life for modifications and updates.

Because of the rapid growth of software system scale and the increase in the complexity of problems that software tries to solve, software processes need to be more formal and systematic techniques must be developed and deployed in order to render engineering processes more efficient and less error-prone. Especially, real time distributed software systems such as in the telecommunication domain, where scenarios get more complex since there are different modules running independently, perhaps on different sites. These modules interact and coordinate with message passing, and each module has its own timing. Consequently, system requirements are more difficult to capture, and design processes become more complex. To ensure the correctness and performance of such a software process, some formal methods have been introduced specially for the requirement and design phases.

Formal methods are the mathematics and modeling applicable to the specification, design, and verification of software processes. Formal methods include formal languages, techniques and tools [14]. Consequently, specifications and models that precisely describe all or part of a system's behavior at various levels of abstraction can be developed. Formal methods clearly define the semantics of the requirement and design specifications so that they can be transformed to implementation more easily. Some other advantages of the precise descriptions include automatic validation and verification of the specification against some reference models and ease of redesign and maintenance processes. Some formal languages are widely used for Telecommunication software engineering, such as SDL [4], MSC [5], and Tree and Tabular Combined Notation (TTCN) [21].

MSC is a formal language standardized by the International Telecommunication Union (ITU). It is used to specify behaviors of system components and environment by means of message exchange [1]. MSC has textual and graphical forms and it supports both incomplete and complete specifications. The latest version of MSC is MSC-2000, which bears significant extensions from its predecessors.

SDL, the most widely used formal language, is also standardized by ITU. It is based on the communicating Extended Finite State Machines (EFSM) model. SDL defines the required system behavior or the actual system behavior in a stimulus/response manner [2] with either textual or graphical forms. With supporting tools such as TAU SDL Suite [20], an SDL design specification can be converted into an implementation.

MSC and SDL are generally used together. Not only does MSC provide the requirement document for design using SDL, but also it can be used to generate test cases.

1.2 Goal of the thesis

As explained before, a software process goes through several phases. The documents produced in one phase are used as input for the next phase. Therefore, some transitions need to be performed when moving from one phase to the next. Usually this transition is done manually, that is, software engineers analyze the requirement specification document, and come up with a design to solve the problem specified in the requirement phase. Then, the design is elaborated and coded. The transition process is likely to introduce some errors. Based on formal methods, some engineering tools have been developed to automate the transition steps. For example, ObjectGeode [12] can generate high-level language code in C++ or Java from a well designed SDL design specification. Since SDL is a formal specification language, the design can be verified more easily against the requirements and translated into code without error. Therefore, in the telecommunication software processes, the work from design to implementation has been automated, which reduces the amount of effort for the software developers; and, at the same time, the produced code is less prone to error.

Similarly, we consider the transition from the requirement specification to the design specification can be automated as well. When the requirement is specified in a formal language such as MSC, then correctness and consistency can be verified. If the MSC specification is correct, we can map it into a formal design specification such as in SDL. Figure 1.1 shows the transition between phases of software process with automatic tools. Researchers have been working on developing such approaches to generate automatically the design from the requirement. The telesoft group at Concordia University devised an approach and developed a tool for generating SDL design specifications from MSC requirement specifications and a given target architecture [2] [7]. The reason for choosing MSC and SDL is that they are compatible languages and both of them are widely used in Telecommunication. The work done so far by the telesoft group was based on MSC-96. It guarantees the correctness of the design, and consistency between the SDL specification and MSC specification. The need for validation has been eliminated.

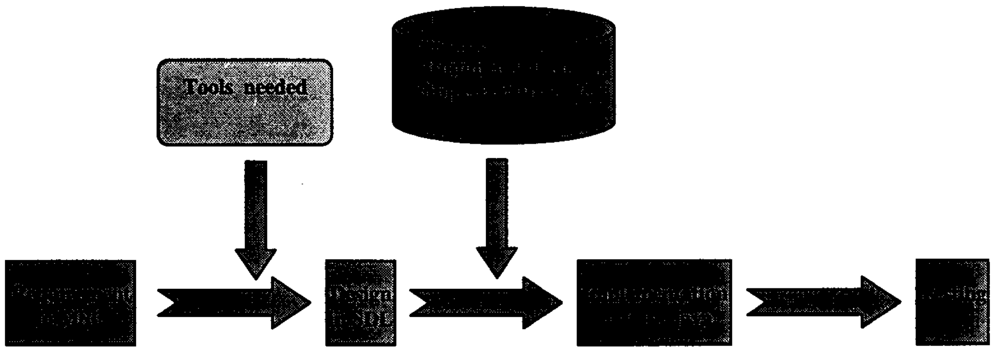


Figure 1.1: Software process with automatic tools

The MSC language has evolved and time concepts have been introduced in MSC-2000. This enables real-time requirements to be specified in MSC. The existing approach and tool do not handle time concepts. Translating timed MSC specifications into SDL specifications has become a new challenge.

Building on the existing framework, this thesis presents a new approach for handling real-time requirements in MSC, and generating SDL specification from timed MSCs with a given target design architecture. We have analyzed and classified different types of time constraints and measurements. In order to incorporate time information into the framework, the new approach has extended the existing internal structures and added new structures. We have devised new algorithms for analyzing semantics of timed MSC specifications, building necessary internal structures, detecting semantic errors, and generating process behaviors in SDL. We have built the new MSC2SDL tool with the new approach and experimented with case studies.

1.3 Contribution of the thesis

The existing approach initiated in [2] derives SDL process behaviors from a set of MSCs with a given SDL architecture. The approach ensures, by construction, consistency between the SDL specification and the basic MSC (bMSC) specification [2]. Therefore, no validation is needed for the SDL specification against the requirements specified in MSC. The approach also prevents deadlocks that may be introduced during translation. This approach has been extended in [7] to handle MSC timers, coregions, message overtaking, and high level MSC (HMSC) operators.

MSC-96 has evolved into MSC-2000 with time concepts. System designers can specify time constraints and measurements associated with events, which represent a crucial aspect of Telecom Software. The existing approach does not handle this aspect and our goal is to develop an approach so that the design specifications can be generated automatically from the requirement specifications for real-time software systems. We first studied and categorized MSC time constraints and measurements. In order to incorporate time concepts, we re-designed the internal structures of the existing framework. Then, we devised the mapping from MSC time concepts to SDL constructs and an algorithm for generating SDL specifications from timed MSC specifications. Moreover, a non-local choice detection technique for bMSCs and an extension for detecting non-local choice for HMSCs were introduced. We also built a compiler for a subset of MSC-2000, and implemented our new approach in the MSC2SDL tool.

The achievement of this work is not only an extension of the existing approach; it also introduces real-time requirements into the picture and handles the associated thorny issues. The contribution bridges the gap between the requirement and design of real-time software system.

1.4 Organization of the thesis

The rest of the thesis is organized into six chapters.

Chapter 2 gives an overview of the MSC language and focuses on MSC time concepts.

Chapter 3 introduces the SDL language.

Chapter 4 describes the existing approach for generating SDL specifications from MSC specifications as presented in [2] and [7]. First of all, an overview of the existing approach is presented, including the internal structures and the mechanism how bMSC and HMSC concepts such as events, coregion, HMSC operators are handled. Then, the limitations of the approach are discussed.

Chapter 5 presents the complete approach for handling time concepts. We first analyze the time constraints and measurements in MSC. Then, we present our techniques for handling them using some extended or added internal structures, and for translating them into the equivalent SDL constructs. Next, the study of HMSC is presented. Moreover, the technique for handling inline expressions is also presented and discussed. Finally we finish this chapter by discussing some time related issues and the related work.

Chapter 6 presents the implementation of our approach, the MSC2SDL tool. It includes the compiler for a subset of MSC-2000, and the SDL generator for timed MSCs. Interfaces of the tool and two case studies are also presented in this chapter.

Chapter 7 summarizes the results and contributions of the thesis, as well as the possible future investigations and expansions.

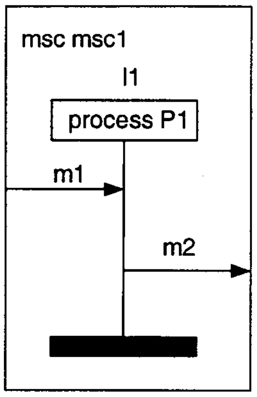
Chapter 2

Message Sequence Charts

2.1 Introduction

MSC is a powerful and expressive language to describe interactions among system components. It is standardized by the International Telecommunication Union (ITU); its latest version is MSC-2000 [5]. MSC is a formal language with two forms: textual and graphical. An MSC specification defines the order of events occurring in different instances, as well as the timing constraints on those events. MSC also supports some structural concepts. MSC has gained in popularity among telecommunication software engineers. They are not only used to capture the requirements, but also to specify test cases, and performance constraints of the systems.

In this chapter, we give a brief introduction to bMSC, MSC structural concepts (including HMSC) and Time Concepts. A complete description of MSC can be found in recommendation Z.120 [5]. Figure 2.1 shows a simple MSC specification in both Graphical (MSC/GR) and textual (MSC/PR) forms.



```
msc msc1;
I1: instance process P1;
in m1 from env;
out m2 to env;
endinstance;
endmsc;
```

Figure 2.1: MSC Graphical and Textual forms

As we can see, MSC/GR describes typical scenario(s) in an intuitive way. It is easy for designers to specify behavioral requirements and understand them. MSC/PR is an alternative way of describing system behaviors. It is easy for the automatic tools to manipulate specifications in MSC/PR form. For the illustration purpose, we use MSC/GR in this thesis.

2.2 Basic MSC

A bMSC describes behaviors of parallel processes and asynchronous message exchanges among them. We will introduce the instance, environment, message, action, timer events, instance creation, instance stop, condition and ordering concepts.

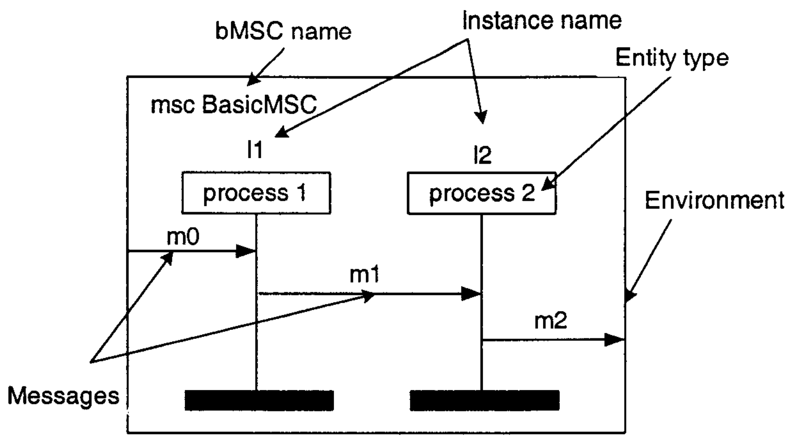


Figure 2.2: Basic MSC concepts

Instance

MSC is about behavior of instances and interactions among them as well as with their environment. An instance can be of some process type; it has its own behavior that is independent of other instances. Messages are sent from one instance to another. An MSC instance might be an SDL system, block, process, etc. MSC instances play a key role in the MSC specifications. In the graphical form, an instance has an instance head, an axis symbol, and an instance end symbol. The semantics of these elements are: the instance head symbol determines the start of the description of an instance within an MSC. It does not describe the creation of the instance. Correspondingly, the instance end symbol determines the end of the description of an instance within an MSC. It does not determine the termination of the instance. Events on one instance axis are totally ordered from top to bottom. Examples of instances, I1 and I2 are shown in Figure 2.2.

Message

Message is the means by which instances communicate with each other. A message has a sending event and a receiving event. The output and input relation provides a total order assumption between these two events. Both sending and receiving of messages can happen in instances or environment. In the graphical form, a message is represented by an arrow from its output to input (see Figure 2.2). In textual form, keywords **before** and **after** can be used to define an ordering of message events on different instances.

Environment

An MSC has an environment. Instances may interact with the environment through message exchange. There is no control on the behavior of the environment. However, an assumption that the environment cooperates properly with the MSC specification is agreed in general. There is no order among message input and output events that happen in the environment. In the MSC graphical form, the environment is represented by a frame that is the boundary of an MSC specification as shown in Figure 2.2

Action

Actions are atomic events that represent some internal manipulation within an instance. There are two types of actions. The informal actions are associated with informal text enclosed in a single quote, and the formal actions are specified with one or more data statements. These statements may be binding, etc. Figure 2.3 shows two actions, one is a formal action that binds 0 to variable a; the other is an informal action, which is in the form of a string, 'processing'.

Timer events

Timers can be used in MSC specification to define certain time related requirement. There are three kinds of timer events, namely, **starttimer**, **stoptimer** and **timeout**. When started, a timer is set to a duration (if the duration is not specified, then the default value from mscdocument is used). After the time specified by this duration elapses, a timer encounters a timeout, hence consumption of the timer signal. However, before timeout happens, this timer can be stopped. Any starttimer event must be matched by a timeout or stoptimer events, even though they can be appear in different MSCs. In Figure 2.3, timer T1 is started in instance I1 and then it is stopped. In instance I2, timer T2 is started, and after the default time duration passes, it timeouts.

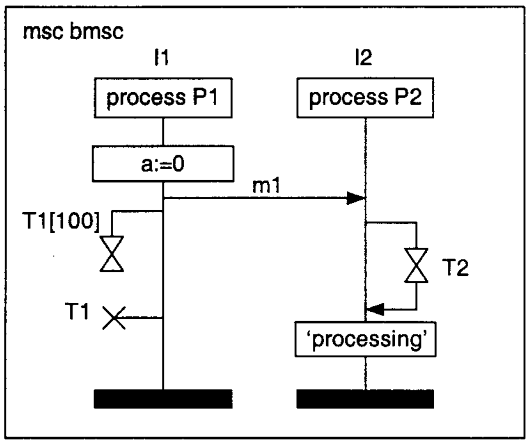


Figure 2.3: MSC timer events and actions

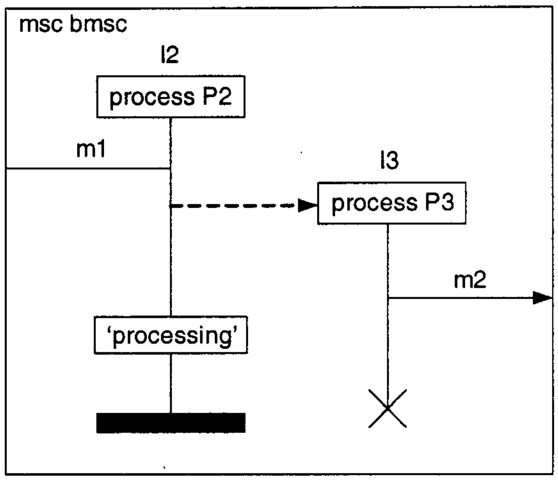


Figure 2.4: MSC instance creation and stop

Instance Creation

In MSC, an instance may be created by another instance. No events can take place in a created instance before its creation. Figure 2.4 shows an example in which instance I2 creates instance I3.

Instance Stop

The instance stop is the counterpart of the instance creation. When it stops, an instance terminates its lifetime. In Figure 2.4, after sending message m2 to the environment, instance I3 terminates its execution.

Condition

Conditions are allowed in MSC to restrict traces. There are two types of condition: setting and guarding conditions. A setting condition sets a state either global for all the instances or non-global for a subset of instances only. A guarding condition restricts the behavior of an MSC by only allowing the execution of events in a certain part of the MSC depending on the guard's value. Guarding conditions are usually used in inline expressions that we will introduce in later sections. Figure 2.5 shows setting conditions. Starting is a global condition and Processing is a non-global condition set by I2 after receiving the message m1.

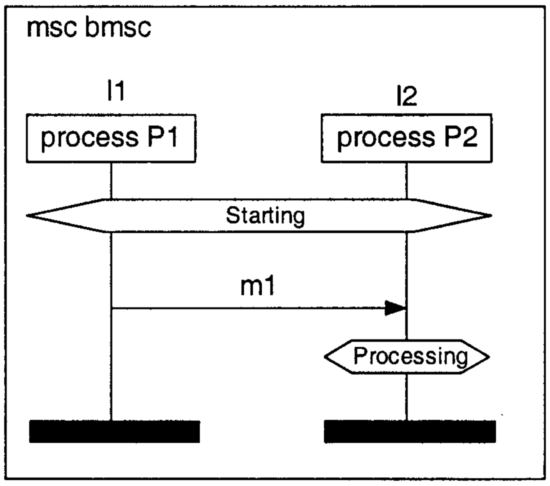


Figure 2.5: Setting condition

General Ordering

The order of events in MSC is determined by three rules, 1. Events on the same instance are ordered according to their positions along the instance axis; 2. A message-sending event always precedes its corresponding receiving event; and 3. The events happening on the created instance are behind its creating event.

However, MSC allows ordering events, which are usually used in coregion to specify some desired order among events in such an order-relaxed region. General ordering is defined by the keyword **before** and **after** in the textual form, and it also has its graphical representation. An example will be given after we introduce the coregion concept.

2.3 Structural concepts

MSC provides some structure constructs to allow for more complex specifications. We present here a brief introduction to these structures.

2.3.1 Coregion

As mentioned previously, the order among events in one instance is totally specified according to the positions of these events in the axis. However, the order among events is relaxed or partially specified in a coregion. This is useful when the order of events is not yet defined and may be refined in the future. In Figure 2.6, there is a coregion in instance I2; receiving message m2, receiving message m3 and sending message m4 are in this coregion. Then, these three events are not totally ordered by the instance axis. They can be in any order except that receiving m2 must precede sending m4 since a general order is specified between them in instance I1.

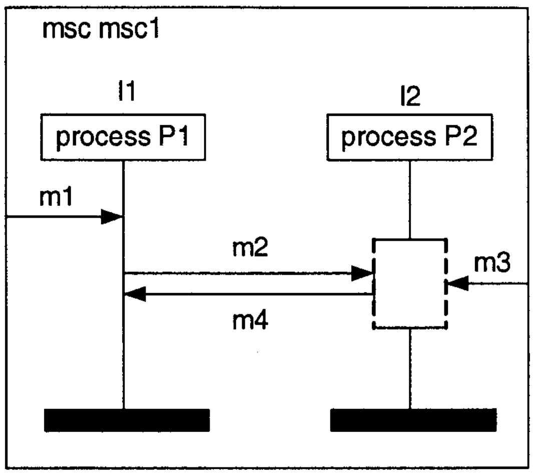


Figure 2.6: Coregion with general ordering

2.3.2 Inline expressions

There are five kinds of inline expressions, which allow designers to specify more complex behaviors.

The **seq** operator represents the weak sequencing operation for sequential composition of MSCs. A guarded **seq** operand with a false guard is dynamically illegal.

The **par** operator defines the parallel execution of MSC sections. All events in the MSC parallel section are executed. However, only the order among events within each operand is preserved. If all guards of all operands are evaluated *false*, then par-expression becomes empty.

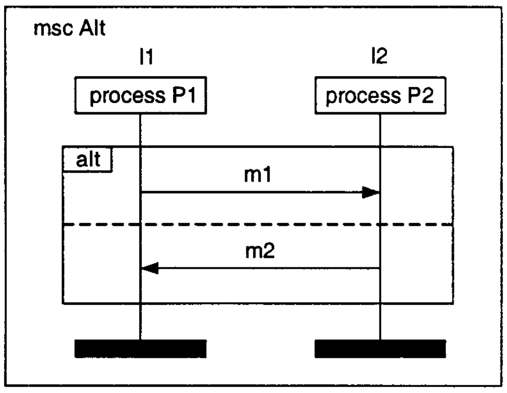
The **opt** operator is an optional section for the MSC to execute. If the guard for opt section is evaluated *false,* then this optional section is not executed. Otherwise, it is executed.

The **alt** operator defines alternative scenarios. Only one of the operands is executed. If all guards are false, then the specification is dynamically illegal. However, if there is more than one operand having true guard values, at run time, MSC randomly executes one of them. Another case is that one operand of **alt** can be guarded with the keyword **otherwise**, which is the complement of the conjunction of all guards in other operands.

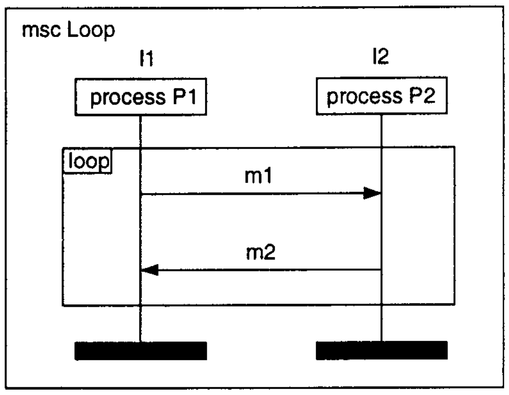
The **loop** operator defines an iteration scenario. Event sequences inside of a **loop** are executed repeatedly to the specified times. If no loop boundary is specified, MSC uses some default values. If the boundary is specified as a range, then the loop is executed a number of times within the range.

The **exc** provides a facility for specifying exceptional cases in MSC. If the execution enters **exc** section, MSC terminates after execute all events inside this section. Otherwise, MSC executes the rest of the section. An **exc** section must be shared by all instances in the MSC.

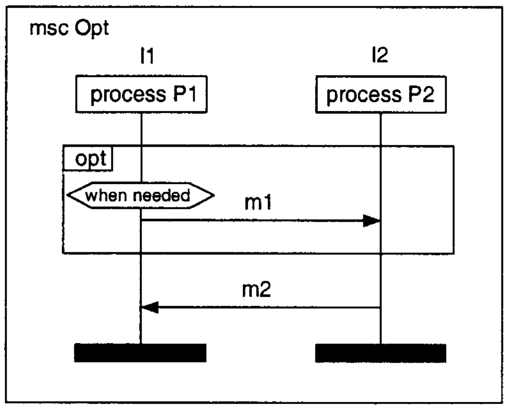
Examples of inline expressions are shown in figure 2.7. In figure 2.7(a), the **alt** inline expression has two operands. One is instance I1 sends message m1 to instance I2; the other is instance I2 sends message m2 to instance I1. Only one operand can be executed. In figure 2.7(b), a **loop** inline expression is specified. The scenario that instance I1 and I2 exchange message m1 and m2 is executed repeatedly. In figure 2.7(c), the **opt** inline expression has a guarding condition, if this condition is evaluated true, the opt inline expression is executed. In figure 2.7(d), an **exc** inline expression is specified. If an exception occurs after instance I1 receives message m0 from instance I2, I1 sends message m1 to I2. Otherwise, I1 sends m2. In figure 2.7(e), the **par** inline expression describes two parallel execution traces. One is instance I1 send message m1 to instance I2, the other is instance I2 send message m2 to instance I1 followed by I1 sending message m3 to I2.



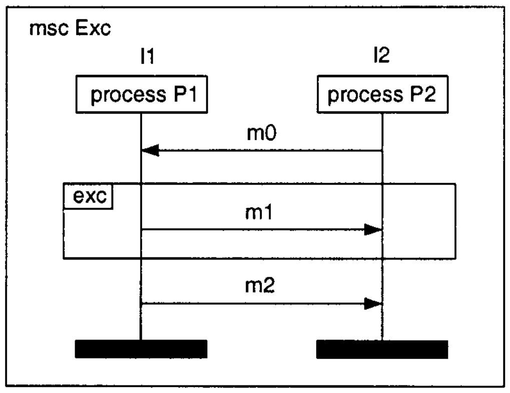
(a)



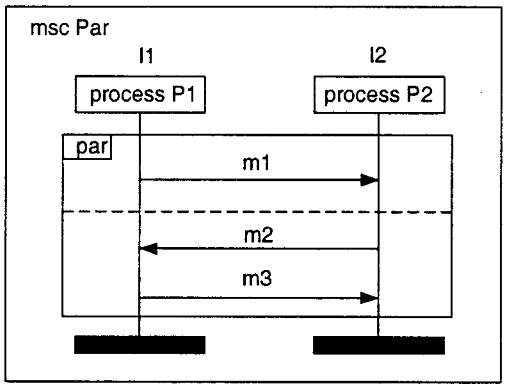
(b)



(c)



(d)



(e)

Figure 2.7: Inline expressions

2.3.3 High-level MSC

HMSC provides a means to graphically combine a set of MSCs. An HMSC is a directed graph. Its nodes can be a start symbol, an end symbol, an MSC reference, a condition, a connection point, or a parallel frame [5]. In addition, HMSC provides four operators to structure referenced MSCs in sequential, parallel, alterative, or iterative manner. They are similar to the corresponding operators in Inline expressions. HMSC describe a system hierarchy that can contain both bMSCs and HMSCs. Figure 2.8 shows an HMSC example, where m1, m2, m3, and m4 are bMSCs or HMSCs. The control flow goes to m1 after interpreting the start symbol. An alterative operator following m1 directs the control flow to m2 or m3. There is also an iterative operator, specifying a free loop from m2 to m1. The end symbol is interpreted after executing m4.

The end symbol is not compulsory; an HMSC does not need to have it. Conditions can also be used in HMSC to indicate global system states or guards and impose restrictions on the MSCs that are referenced in an HMSC specification [5].

In an HMSC, the components of MSC are connected process by process according to the weak sequencing semantic. For instance, if both m3 and m4 are bMSCs and contain a common process P; an event e1 in P in m3 and another event e2 in P in m4, then e1 precedes e2. However, the order of events on different processes is not specified, unless it can be implied through weak sequencing semantic and temporal order rules.

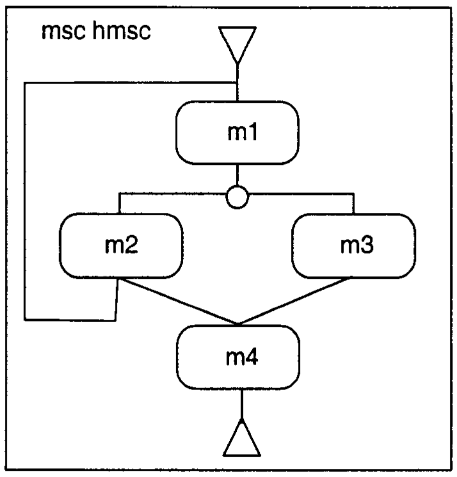


Figure 2.8: An HMSC

2.4 Time concepts

Time concepts are introduced in MSC-2000 to support the notion of quantified time for the description of real-time systems with a precise meaning of the sequence of events in time. Time constraints can be specified in order to define the time at which events may occur. The progress of time is represented explicitly in a quantified manner, i.e. the traces of events are enhanced with a special event that represents the passage of time with quantitative time values. Classical MSC, disregarding time, can be interpreted as a set of traces of events. When time concepts are introduced, the event traces are a subset of those of untimed MSC. The time progress (i.e. Clocking) is equal for all instances in an MSC. Moreover, all the clock values are equal, i.e. a global clock is assumed [5]. According to MSC semantics, MSC events such as message input/output, timer events and actions are instantaneous, i.e. they do not consume time. Time concepts can be used in MSCs in two manners: absolutely or relatively. The absolute time represents the global clock value of the specified system. The relative time represents the time distance between pairs of events. Furthermore, time can be measured and also be used as constraints to pairs of events. The time concepts permit more refined MSC specifications.

2.4.1 Relative constraints and measurements

A relative time constraint is a requirement defined between a pair of events, i.e. the time distance from the preceding event to the subsequent event. A relative time measurement measures time distance from the preceding event to the subsequent event in Figure 2.9; a relative time constraint is specified between event e0 and e1. A relative time measurement is specified between event e0 and e2; the measurement result can be used in the following specification to constraint other events.

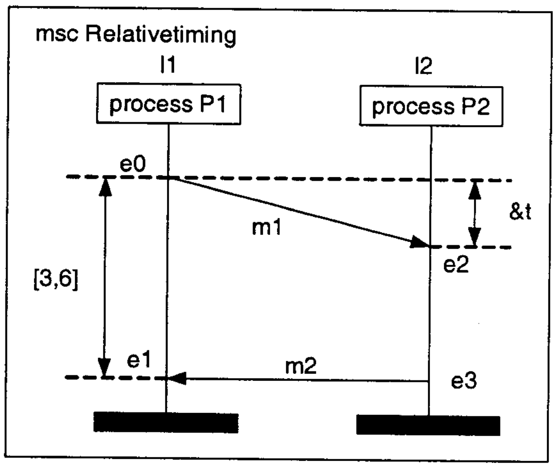


Figure 2.9: Relative time constraint and measurement

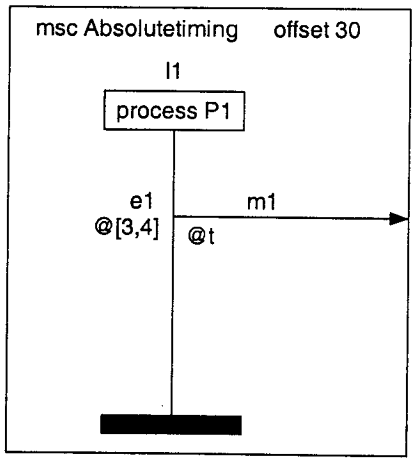


Figure 2.10: Absolute time constraint and measurement

2.4.2 Absolute constraints and measurements

An absolute time constraint is defined for one event as the global clock value at which this event happens. An absolute time measurement measures the occurrence time of an event. For example, an absolute timing is specified for event e1 in Figure 2.10, which requires event e1 to occur at absolute time in the range [3, 4]. Moreover, an absolute time measurement is also specified for e1, which measures its occurrence time.

2.4.3 Time offset

A time offset can be assigned to an MSC, which is used to offset all absolute time values within that MSC. In Figure 2.10, MSC Absolute timing has an offset 30, and then the absolute constraint for e1 actually becomes [33, 34].

2.4.4 Time points, measurement and intervals

These two concepts are used to define semantics for time constraints and measurements. In terms of absolute time constraints or measurements, they are always referred as time points, which are the absolute global clock values. As far as relative time constraints are concerned, a time interval is referred, which represents the time distance between two time points. Time points and time intervals can also be measured as described in the previous sections. In Figure 2.10, the absolute time constraint for event e1 is specified as a time point, which should fall into the absolute time range [3, 4]. While in Figure 2.9, the relative time constraint specified between e0 and e1 is refereed as a time interval, which should be in the range [3, 6].

2.5 Data part in MSC

MSC-2000 also introduced data concepts [5]. Basically data can be used statically or dynamically [5]. Designers can declare messages, timers with parameters, as well as dynamic variables in the MSC document. Data can be manipulated in binding through message passing or actions. Data can also appear in a condition expression, a time constraint or as timer duration. The usage of data in MSC is very important in our study of translating time requirements into SDL and non-local choice detection. Figure 2.11 shows an example of MSC data usage. A variable called "a" first is used in a guarding condition to restrict the execution of the MSC specification. Then, its value is bound to a variable called "b" in another instance through a message passing. Finally, the MSC increments the value of variable "b" by one with an action box.

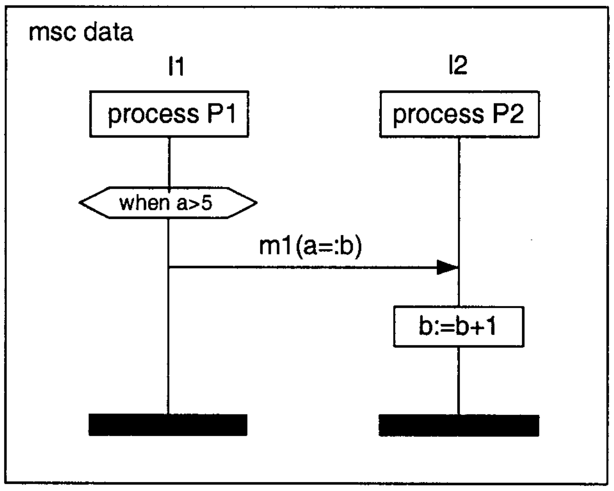


Figure 2.11: Data in MSC

Chapter 3

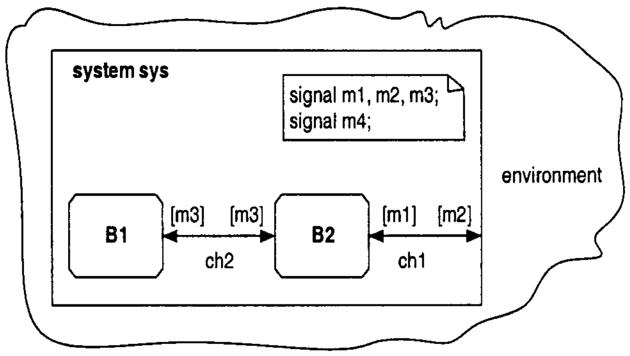
Specification and Description Language

3.1 Introduction

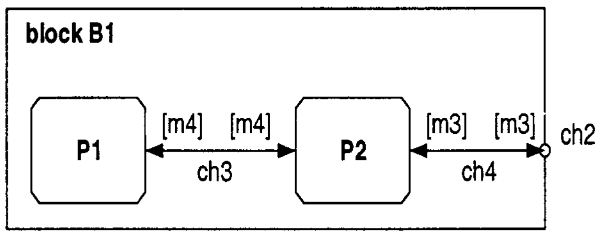
SDL has been standardized by ITU. Its latest version is SDL-2000 [4]. This language has been defined for unambiguous specification and description of reactive systems. SDL specifications consist of behaviors, data description and structure [4]. The basic behavior description is based on the Extended Finite State Machine (EFSM) model. Data description is based on abstract data types. Abstract Syntax Notation One (ASN.1) language [16] has been added into SDL standard for data description. The structure consists of a set of hierarchical blocks and processes, communicating through signal channels. In this chapter, we review SDL concepts important to our study.

3.2 Environment

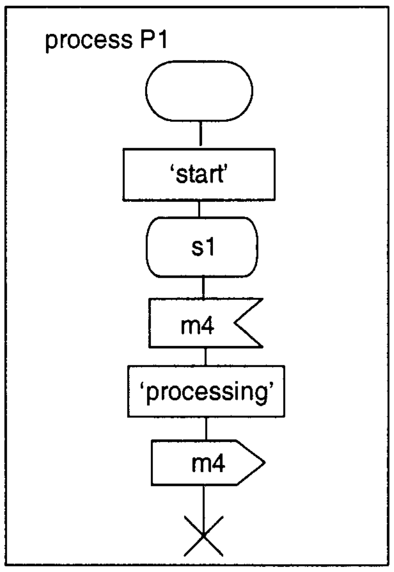
SDL Systems interact with their environment. The behavior of the environment is non-deterministic, but by assumption, it cooperates with SDL systems. Figure 3.1 (a) shows an example of the environment, which is indicated by the boundary of the system.



(a) System Level



(b) Block level



(c) Process level

Figure 3.1: A simple SDL example

3.3 Agents

In SDL-2000, an agent is defined as one or a set of entities, which have their own variables, procedures, state machines and some contained agents. Agents can be classified into three types:

System

A system is the outmost agent. It contains signal definition, channel definition, data types, and other agents. A system is separated from the environment with the system boundary. Agents inside of a system may interact with the environment though signals conveyed by channels. Inside a system, agents may also interact with each other as well. In Figure 3.1 (a), system sys contains some signal declarations, block agents, and communication channels.

Block

The blocks are the key components for SDL providing hierarchical architecture to system specifications. A block is contained in a system or another block. It may contain processes or one or more blocks. Therefore, blocks are used as containers of other agents. Inside a block, signals, channels, data types and agent types can be defined. And the communication among contained agents and outside agents can be achieved through channels and signals. As an example shown in Figure 3.1 (b), the block B1 was contained in system sys; it contains two processes P1, P2, and communication channels as well.

Process

Processes define the behavior of a system. In SDL, a process is represented as an EFSM. A process instance can be created in the system initially when the system starts, or by some other process instances during its execution. There may be multiple instances of one process type existing in one system.

Processes communicate with one another through signals sending and receiving. Each process instance has its own identity (pid), memory space, variables, states, etc. We can see in Figure 3.1 (b), two processes P1 and P2 communicate with each other through channel ch3. Figure 3.1 (c) shows behavior of P1.

A process identifies itself with the keyword **self**, and the process instance that has created it, as **parent**. Moreover, every process has an input queue for received signals. Normally this queue is First-in First-out (FIFO). However, when some signals have priority, then such signals can be selected and consumed from the queues without respecting FIFO order. A sending process instance of a signal can be identified by the keyword **sender**.

There are two process instances in Figure 3.1 (b), and Figure 3.1 (c) shows P1's behavior.

3.4 Communication

Communications among agents are done through channels using signals.

Channel

A channel represents a transportation path for signals. It has a FIFO queue. It can be unidirectional or bi-directional. A channel can connect two agents or one agent with its environment. Signals conveyed by channels are delivered to their destination, the endpoint of the channel. For example, channel ch3 in Figure 3.1 (b) is a bi-directional channel that conveys signal m3 in both directions.

There are two types of channels in terms of delay time. Some channels transport signals without delay and are denoted by **NODELAY**. If a channel has a delay, then the delay is non-deterministic.

Signal

A signal is a message instance that travels from one agent to another. It can be sent or received by either an agent or environment; m1, m2, m3, and m4 are examples of signals in Figure 3.1.

3.5 Behavior

SDL system behavior is composed by the set of process behaviors. For specifying process behaviors in form of EFSMs, there are defined constructs. Figure 3.2 shows common ones.

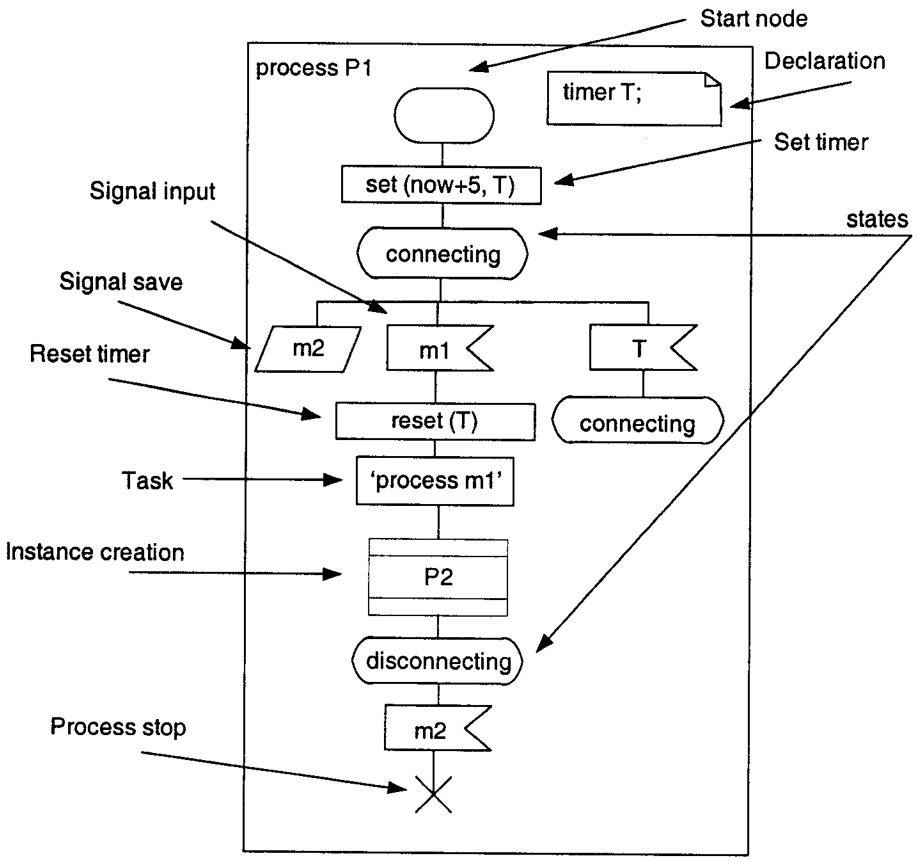


Figure 3.2: Process behavior

Start

Start construct is the starting node of a process in SDL. When a process instance is created, this node is interpreted at the starting point of the execution.

State

States represent status of EFSMs. A trigger can cause an EFSM to transit from one state to another after a series of actions. There are three types of triggers, namely **input**, **Spontaneous signals**, and **Continuous signals**. An **Input** has higher priority in terms of consumption over a **Continuous signal** within a state. However, there is no further priority defined among the three types of triggers so that in some cases, transition from one state may be non-deterministic when more than one possible transition exists. Same states can be specified in different places, this is useful sometimes to provide clear specifications.

Input

An input allows the consumption of the specified signal instance provided that the signal is in the input queue of the process instance. In addition to being a trigger for a process to transit from one state to another, the interpretation of input binds the data conveyed by the signals to variables in the receiving process instance address space. A signal also carries the pid of the sender process instance so that it is known to the receiver upon the signal consumption. An input originates from one instance would carry a different pid from any other process instances in the system.

Priority input

In one state, more than one input trigger can be specified. In the case that some input signals are favored, Priority inputs are convenient to express that the reception of one or more signals take the priority over that of other signals. For example, in Figure 3.3, m1 has priority to m2, and then as long as m1 is in the input queue, it is consumed before m2.

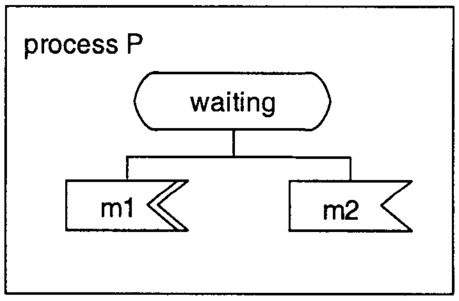


Figure 3.3: Priority input

Save

The semantics of consumption of the signals in the input queue defines that if the signal at the head of the queue is not specified for consumption in a state, then it is discarded. In some cases, if such signals may be useful for the system transitions in the future, then save constructs can be used to retain desired signals in the queue and the signals following them can still be consumed before them.

Continuous signal and enabling condition

A continuous signal is a trigger with a boolean expression for an SDL state. When the boolean expression is evaluated true, the transition is triggered without the need of consuming a real signal. Since signal inputs have higher priority, this happens only when there is no real signal that matches the input signal list of such a state. The boolean condition used in a continuous signal can also be used to enable consumption of an input signal or a spontaneous transition. In this case, it is called enabling condition. Only if the condition specified following an input signal or a Spontaneous transition for a state is evaluated true, can the transition be triggered. Figure 3.4 shows a continuous signal. If the condition is evaluated true, then the process will transfer from states s1 to s2. While in Figure 3.5, the enabling condition is imposed on the input signal m1. Only when the condition is evaluated true, upon the consumption of m1, does the process makes its transition.

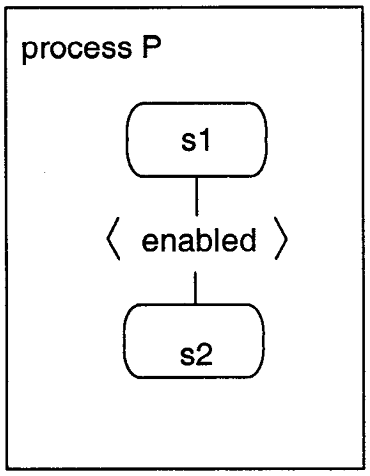


Figure 3.4: Continuous signal

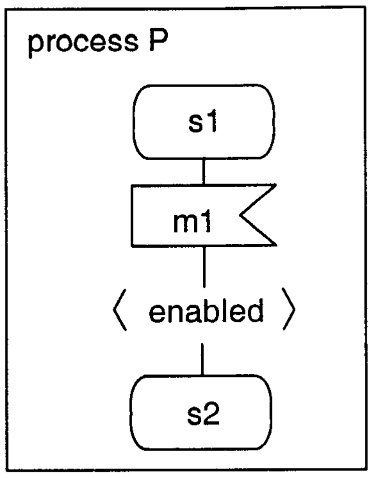


Figure 3.5: Enabling condition

Spontaneous transition

A spontaneous transition specifies a state transition without any signal consumption. In fact, **none** is used as the input clause; and **sender** expression contains **self**. The activation of a spontaneous transition is independent of the presence of signal instances in the input queue.

Task

Tasks are internal manipulations inside a process. Usually they are used for data bindings or informal tasks in the form of text. An example can be found in Figure 3.2.

Create

A process instance may create other instances. Created instances have keyword **self** containing their new pids; and a parent instance has its **offspring** containing the created instance's pid. Create cannot create more instances of one type than the specified number in the definition. In Figure 3.2, process P1 creates a process instance of P2.

Output

Process instances use output constructs to send messages to other instances. An output usually has two ways to address the destination instances: direct addressing, specifying the destination instance pid; and indirect addressing, specifying a channel by which a signal needs to go through. If there is more than one channel that carries the signal, the specification is ambiguous.

Decision

A decision construct is used to branch transitions according to some conditions. For one condition, there may be multiple answers and each of them leads to a separate transition. Answers can be formal or informal (in text). Whenever one answer is evaluated true, the transition may take the corresponding branch. However, if there is more than one answer evaluated true, the transition will be nondeterministic.

Stop

A stop causes an agent to terminate. This termination means that the agent ceases to exist and all its memory space is released. The last construct in Figure 3.2 is the stop for process P1.

3.6 Timer

Timers are objects that belong to a process instance. They are used to specify a time interval. When the specified interval has passed, a timer signal is generated and inserted into the instance input queue. Then the process instance can consume this signal as any other input signal.

There are two timer actions:

Set Starts a timer with a specified time interval, the timer becomes active.

Reset Stops a timer before or after it expires. This action makes the timer inactive. If the timer has already expired, then its signal is removed from the input queue.

3.7 Data

In order to handle the time concepts, we need to include the SDL data part (especially data carried by signals and data bindings in action boxes) into our approach.

Here we give a brief introduction to SDL data types. SDL uses the concept of abstract data types. Data types are called *sorts* in SDL. As in high level programming languages, SDL provides users some pre-defined sorts, and also allows new sorts to be created. Table 3.1 shows SDL pre-defined sorts. Sorts can be defined at any level, system, block or process, and they can be used by any entity in the same level or in a sub-level. SDL signals, Timers and variables should be declared before they can be used. SDL signal names should be declared at the system level or block level with the keyword ***signal*** (e.g., ***signal*** m1, m2 ;). SDL timers are declared within the owning process scope with the keyword ***timer*** (e.g., *timer* T1 ;). SDL variables are declared within the process scope with the keyword ***dcl*** (e.g., ***dcl*** a, b *integer* ;).

Pre-defined sorts	Examples
Boolean	True, false
Integer	1, -3, 2004
Character	‘a’, ‘O’, ‘%’, ‘6’
Charstring	“processing”, “SDL”
Time	3, 0, 102
Real	3.14, -1.745, 2
Natural	1, 2, 3
Duration	8, 20
PID (Process id)	null

Table 3.1: SDL sorts

Moreover, ASN.1 has also been added into SDL data part. One can use ASN.1 or even C language to describe data for an SDL system.

Chapter 4

The Existing MSC2SDL Approach

4.1 Background

In a previous research work, researchers from the telesoft group at Concordia devised an approach and developed a tool for generating SDL specifications from MSC specifications. This approach, initially introduced in [2], translates a bMSC specification with a given SDL architecture into SDL process behaviors. Essentially, the bMSC consists of instances that exchange messages. Then the initial approach has been extended in [7]. Instead of handling just message input and output events, the extended approach incorporated more types of bMSC events and HMSCs. The existing approach guarantees correctness of the design, and consistency between the generated SDL specification and the given MSC specification. The need of validation of the design specification against the requirement specification has been eliminated. Figure 4.1 illustrates this approach.

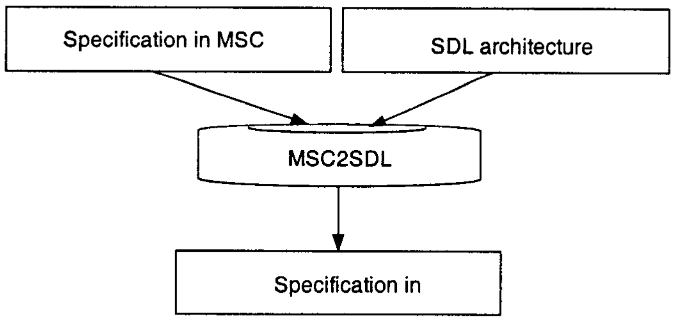


Figure 4.1: MSC2SDL apporach

The MSC2SDL approach is based on MSC 96. It handles various MSC events and structures, such as message input/output, action, timer events, instance creation, coregion, and HMSC. The approach takes MSCs and generates process behaviors in SDL. It bridges the gaps from the requirement to the design of software processes. The existing approach also has its limitations. First, it does not handle MSC inline expressions and parallel operator of HMSCs. Second, it does not handle the newly added concepts in MSC-2000: Time and Data. We review the existing approach in the rest of this chapter, since it is the framework on which our work is based.

4.2 The existing approach for bMSC

The existing approach first checks the architecture consistency between an MSC specification and a given SDL architecture, and then generates SDL process behaviors. The translation from bMSC events to SDL behavior constructs is mostly based on a mapping scheme. In order to prevent possible deadlocks of the generated SDL system, the approach also builds two tables. One is used to keep track of the order among bMSC events, and the other is used to keep track of all signals that may be in the input queue of a process upon its consumption of each signal. With these two tables, the approach analyzes possible receptions in each state of a process and preserves useful signals for future consumptions, which would be discarded otherwise. To better illustrate the existing approach, we will use an example.

4.2.1 Behavioral and architecture consistency

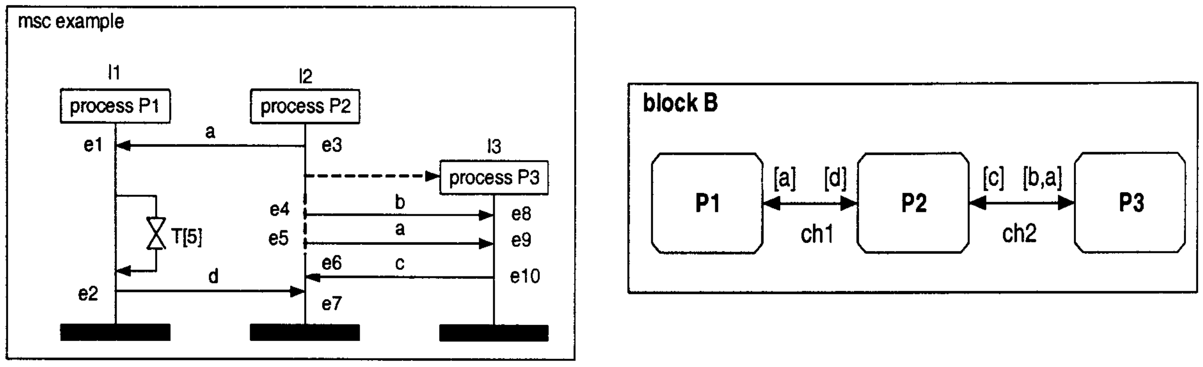
In software processes, documents produced in each phase need to conform to those of the previous phases to ensure the correctness. In telecommunication software design, an SDL specification usually needs to be validated against its MSC specifications to ensure the behavior consistency between the requirement and the design. For the purpose of generation, the existing approach states that an SDL specification is consistent to an MSC specification in their behavior if and only if the set of traces defined by the MSC is included in the set of traces of the SDL specification [7]. Since the approach implements the MSC specification with SDL so it ensures the behavioral consistency between the SDL specification and the MSC specification.

On the other hand, to make sure that the MSC can be implemented in the given SDL architecture, the architecture consistency has to be checked:

All processes described in the given MSC are present in the given SDL architecture.

There is a channel connecting the sending and the receiving processes in the given SDL architecture for each message exchange described in the given MSC.

For example, Figure 4.2(a) and (b) give an MSC specification and an SDL architecture respectively. We say that they are consistent from architecture point of view. Since for each process in the MSC specification, there is a corresponding process in the architecture; and for each message in the given MSC, there is a channel conveying the signal from the sender to the receiver.



(a) (b)

Figure 4.2: Architecture consistency example

Moreover, the SDL architecture may consist of more processes and routes than those required by the MSC specification. For instance, if in the Figure 4.2(b), there is a process P3 in block B and some channels connecting P3 with other processes, we still consider the MSC specification and the SDL architecture are consistent.

4.2.2 Event Order Table

Messages in MSCs are explicitly specified, and the order of the sending/consumption events with respect to their instances is specified [7]. However, MSCs do not specify the actual arrival order of the messages into the input queue of the destination processes. Rather, the order depends on the underlying architecture and the process behaviors interleaving. On the other hand, SDL processes implicitly discard signals, which are in the front of their input queues, and are not expected in the current state. These discarded signals may lead to a deadlock if they are needed in the next states [7]. For instance, in Figure 4.2 (a), the MSC specifies that instance I2 consumes message c before message d.

However, since message c and d are sent from instance I1 and I3 respectively, either of them may first get into the input queue of instance I2. According to the SDL specification, there is no problem in the case that c arrives before d, which conforms to the consumption order specified in the MSC. However, in the case that signal d arrives before c, process I2 discards signal d when it consumes c, since d is at the head of the input queue and not needed in this state. Therefore, I2 will never run to completion because signal d is no longer available for consumption. To avoid the possible deadlocks, the approach generates SDL save constructs for each signal in the input queue that may be ahead of the signal to be consumed. The need of save construct can be determined by checking the order relation of each input event against all the successive input events of the same instances [7].

The approach builds a table, called Event Order Table, to keep track of the orders between each pair of input/output events specified in the MSC specification. The input/output events are uniquely numbered and represented by rows and columns of the table. If the event represented by the row precedes the event presented by the column, then the corresponding cell is marked.

According to ITU Recommendation Z.120, order among events can be determined by the following temporal partial order rules:

Events are totally ordered for each instance axis.

The output event of a message precedes the corresponding input event.

Furthermore, the MSC create event defines an order relation between the created instance input/output events and all input/output events that are above the create event on the creator instance. In other words, events that precede a create event also precede all events on the created instance. The instance creation modifies the Event Order Table by adding more order relations. Events inside coregion are unordered. Consequently we get the following criteria for building the Event Order Table.

(1) An event precedes all events that follow it on the same instance axis except those in the same coregion.

(2) Output events always precede their corresponding input events.

(3) Events that precede an instance creation event precede all events in the axis of the created instance.

(4) All possible order relations among coregion events are marked in the Event Order Table.

Order among events is transitive. That is, if event e1 precedes e2, and e2 precedes e3, then e1 precedes e3.

Table 4.1 is the Event Order Table for the MSC example in Figure 4.2(a). For instance, in row e3, the table cells corresponding to column e4, e5, e6, and e7 are marked according to criteria 1; table cell (e3, e1) is marked according to criteria 2; the table cells corresponding to columns e8, e9, and e10 are marked according to criteria 3. Moreover, table cells (e4, e5) and (e5, e4) are both marked according to criteria 4. After criteria 1-4 applied to all possible pair of events, then criteria 5 is applied to update the table, and repeated until there is no further change in the table. For example, table cell (e3, e2) is marked because (e3, e1) is marked and (e1, e2) is marked.

	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
e1		T					T			
e2							T			
e3	T	T		T	T	T	T	T	T	T
e4					T	T	T	T	T	T
e5				T		T	T	T	T	T
e6							T			
e7										
e8						T	T		T	T
e9						T	T			T
e10						T	T			

Table 4.1: Event Order Table example

4.2.3 Occupancy Table

Now we have obtained the order relation among all events in an MSC specification with the Event Order Table. The approach then builds another type of tables, called Occupancy table, in order to determine whether other signals may be in the input queue when a specific signal is to be consumed.

Every process with signal input has an occupancy table. One Occupancy table maintains the order relations among input events/signals for a process. Each row of the table corresponds to an input event. Each column represents the incoming channels that convey signals to this process. Each cell in the table is filled with all signals that may be in the input queue (may be at the head of the queue) when the process is ready to consume the signal received by the row event. The following condition states whether a signal would be filled into the row of event e_c.

If Not($e_r << e_c$) *AND NOT* ($e_c << e_s$)

Then the signal received with event Er is filled into the cell corresponding Row Ec and the channel column through which the signal is delivered.

Where,

e_c *denotes the current row event in the Occupancy Table.*

e_r *denotes the current input event.*

e_s *denotes the corresponding output event of* e_r.

<< denotes order relations (proceeds).

($e_x << e_y$*) mean* e_x *precedes* e_y *in time. This relation can be extracted from the Event Order Table where ex represents the row event and* e_y *represents the column event.*

Table 4.2 shows the Occupancy Table for process P2 in the shown in Figure 4.2(a). Process P2 has two input events e6 and e7, and two incoming channels ch1 and ch2. For instance, when P2 is ready to consume signal c (event e6), the approach checks the input message c and d.

For message c,

$NOT(e_r << e_c)$ *AND* $NOT(e_c << e_s)$

NOT (e6<<e6) AND NOT (e6<<e10)

By checking Event Order Table, we know this condition evaluates true. Therefore, signal b is included in the row of event e6 and the column ch2.

For message d,

$NOT(e_r << e_c)$ *AND* $NOT(e_c << e_s)$

NOT (e7<<e6) AND NOT (e6<<e2)

This condition also evaluates true with the order information in the Event Order Table. Therefore, signal d is included in the row of event e6 and the column ch1.

input events	input message	Channel ch1	Channel ch2
e6	c	d	c
e7	d	d	

Table 4.2: The Occupancy Table for process P2 of MSC of Figure 4.2(a)

From this table, we know that signal d maybe at the head of the input queue when process P2 is ready to consume signal c; therefore an SDL save construct needs to be generated.

4.2.4 Generating SDL from bMSC

The generation of SDL constructs from MSC construct is based on a one-to-one construct mapping in most cases. Coregions have to be taken care so that the generated SDL incorporates all possible execution traces. The SDL processes transit from one state to another by consuming an input signal. The existing approach generating an SDL state for each MSC message input event. Moreover, save constructs are also generated for signals savings according to the Occupancy Tables except for messages that are sent by the same instance and travel through the same channel as the input message.

4.2.4.1 SDL process instance identification and addressing

The existing approach has also addressed process addressing, which is essential to the translation from MSC to SDL. Message exchange between instances is explicitly illustrated in the MSC specification. In SDL, the addressing is either explicit or implicit. Usually before process instances have any information of one another, implicit addressing is used, in which a channel is specified for the signal to travel from the sender. Explicit addressing uses SDL pids as the destination of the outputs and these pids can be obtained through keywords **sender**, **parent**, or **offspring**.

In order to use explicit addressing, the approach creates SDL pid variables for an instance to save the pids of the other instances that have communicated with it through the keyword **sender**, if there are outputs following the inputs to the sender instances. Likewise, the pid of a created instance is also saved if the parent instance needs to send messages to the created instance according to the MSC specification. Moreover, the approach also uses destination process names as output signals destinations if both the sender and receiver instances are in the same block.

4.2.4.2 Mapping MSC events to SDL constructs

The generation of SDL constructs from MSC events is based on a one-to-one mapping in most cases. MSC instance create events are mapped to the SDL create construct. MSC action events, message sending and receiving events are also mapped to SDL task, signal output and input constructs respectively.

4.2.4.3 Timer events

MSC timer events Starttimer and Stoptimer are mapped to SDL timer constructs set and reset, respectively. MSC Timeout event is represented in SDL as a signal input. If a timer has been set in SDL, then in every later state, the approach generates a time signal input to expect the possible timeout until the corresponding timer is reset or the timer signal is consumed in some state.

Furthermore, the existing approach has also addressed the ambiguous behavior caused by timeout events within coregions. The details can be found in [7].

4.2.4.4 Coregion

Order among events in one coregion is relaxed. The permutation of sequences of all events in a coregion indicates all possible order relations unless some general order has been defined explicitly. The approach completes Event Order Table with all possible order relations caused by coregion, while preserving any stated general order among events. To generate SDL behavior, the approach builds a coregion tree to preserve all possible execution traces. Then it translates the coregion tree into SDL constructs. For instance, I2 has a coregion including event e4 and e5 in the MSC example shown in Figure 4.1(a). Figure 4.3(a) shows the corresponding coregion tree. The partially generated process behavior of P2 is shown in Figure 4.3(b).

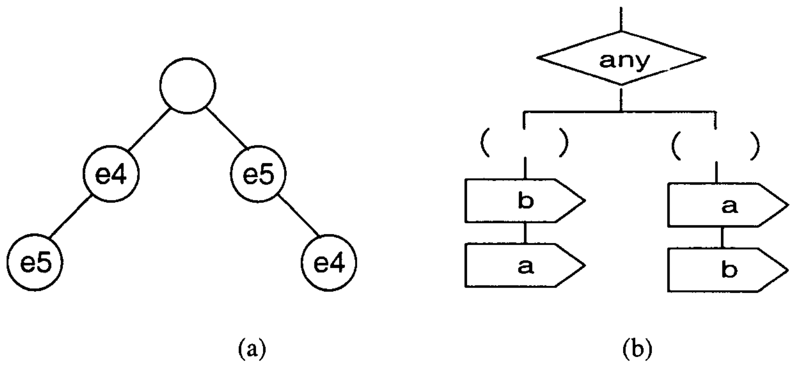


Figure 4.3: Coregion mapping

4.2.4.5 Message overtaking

Message overtaking is the situation in that two messages are sent to the same process instance whereas they are consumed in the order that is opposite to their sending order. In the case that two signals travel the same channel, to avoid possible deadlocks, the existing approach detect it and generate a save construct for the earlier sent message for its later consumption. For instance, in the MSC example Shown in Figure 4.2(a), in the coregion including event e4 and e5, in the case that e5 precedes e4 (I2 sends message a before b), a message overtaking occurs on consumption of these two messages in instance I3. Process P3's behavior in SDL has a save construct for signal a when consuming b.

Finally, following all procedures and techniques described before, the process behavior for the MSC example in Figure 4.2 (a) with the given SDL architecture in Figure 4.2 (b) can be obtained as shown in Figure 4.4.

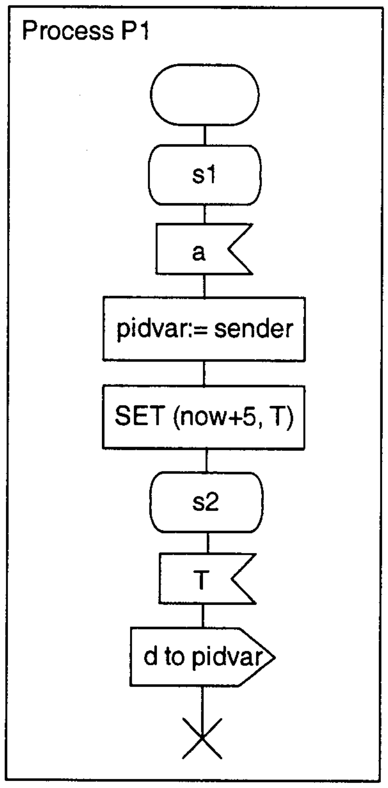


Figure 4.4: Generated process behavior in SDL of MSC for Figure 4.2(a)

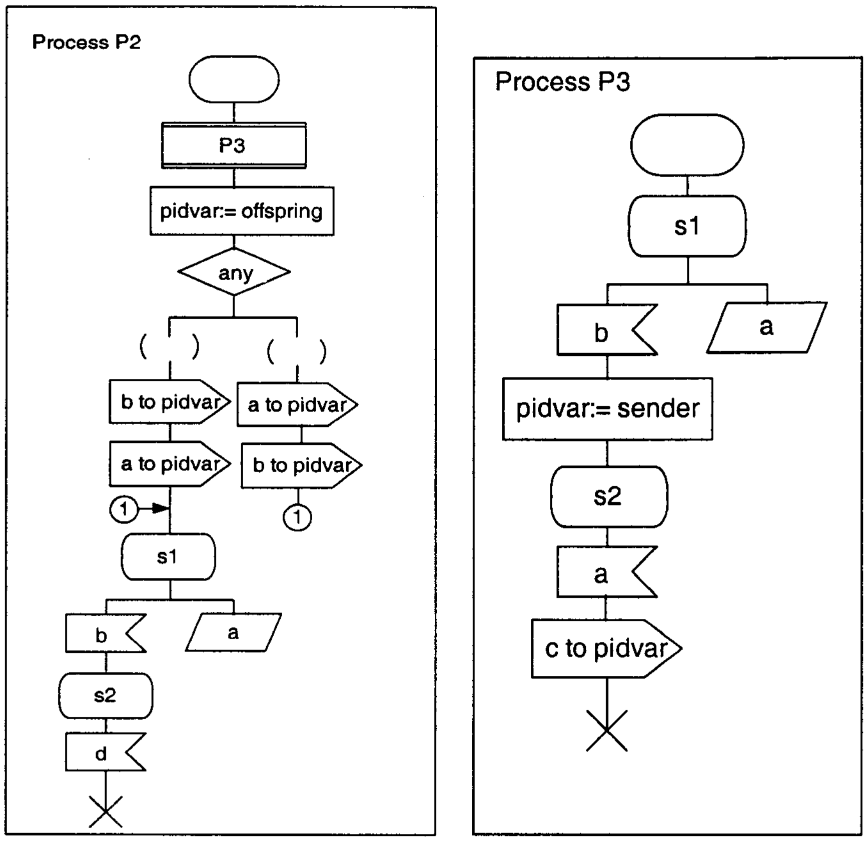


Figure 4.4: Generated process behavior in SDL of MSC of Figure 4.2(a) (continued)

Moreover, the approach assumes that the MSC environment consists of several independent instances with their independent behaviors. No assumption is made about any order between the messages sent by these instances. Therefore, the approach generates a save construct for every signal sent by the environment for all the SDL states that precede the state that in which the signal is consumed.

4.3 The existing approach for HMSC

The existing approach also translates HMSCs to SDL. Following the idea of handling bMSC, both order relation among events and possible signals in the input queues of processes at each state are tracked in order to prevent possible deadlocks. Both the approach of obtaining the Event Order Table and the translation algorithm have been extended to handle the structural concepts of HMSC.

4.3.1 Event Order Table and Occupancy Tables for HMSC

To create Event Order Tables for a given HMSC, the existing approach first assigns a unique number to all input/output events in the HMSC specification. Next, it creates an individual Event Order Table of each bMSC in the HMSC by applying the bMSC Event Order Table algorithm. For example, Figure 4.5 shows an HMSC, and the individual Event Order Tables for each bMSC are shown in Table 4.3.

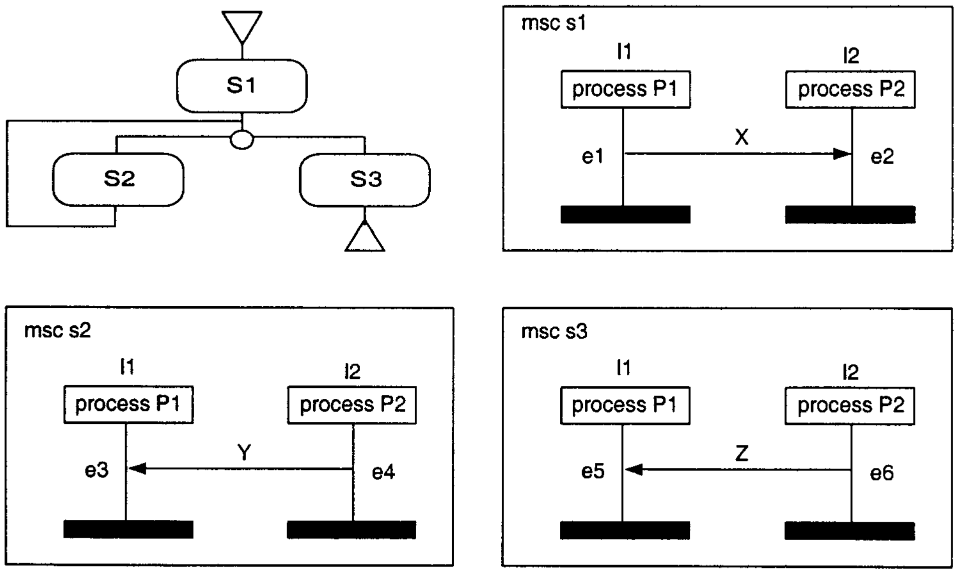


Figure 4.5: An HMSC example with a given SDL architecture

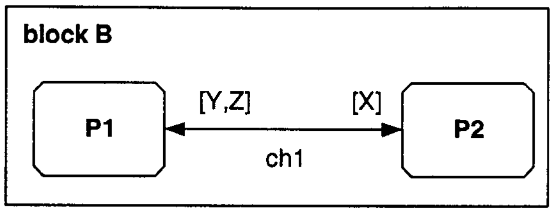


Figure 4.5: An HMSC example with a given SDL architecture (continued)

	el	e2
el		T
e2		

	e3	e4
e3		
e4	T	

	e5	e6
e5		
e6	T	

Table 4.3: Individual Event Order Tables of HMSC example of Figure 4.5

The approach then updates the order among all events of each instance in the Event Order Table according to the order relations among their owning bMSCs. If two bMSCs have the same instances, then the events of an instance in the preceding bMSC precede all events of the same instance in the other bMSC, which is the weak sequencing semantics. For example, bMSC S1 precedes S2 and S3 in Figure 4.5, then event el precedes e3 and e5; event e2 precedes e4 and e6. All events are incorporated into an Event Order Table for HMSC, with the newly added order relations and existed order relations in each bMSC Event Order Table. An intermediate Event Order Table is obtained as shown in Table 4.4 (a).

	e1	e2	e3	e4	e5	e6
e1		T	T		T	
e2				T		T
e3					T	
e4			T			T
e5						
e6					T	

(a)

	e1	e2	e3	e4	e5	e6
e1		T	T	T	T	T
e2			T	T	T	T
e3					T	
e4			T		T	T
e5						
e6					T	

(b)

Table 4.4: Event Order Tables of HMSC example of Figure 4.5

The next step is to update each Event Order Table internally as described in the previous section; this step is repeated until there is no change in the Event Order table. The final Event Order Table for above example is show in Table 4.4 (b).

Occupancy Tables are still used for HMSC to generate save construct for signals which may be at the head of the input queues when each signal is ready to be consumed. The Occupancy Tables can be generated for each process with message inputs after the Event Order Table for the HMSC is obtained. The algorithm remains the same as for bMSCs described before.

4.3.2 Translation of HMSC operators

The existing approach discusses translation of HMSC in terms of HMSC operators, since basically referenced bMSCs can be translated as described in the previous sections, and the operators specify the control flow among the referenced bMSCs. The defined control flow has to be respected when generating SDL specification.

4.3.2.1 Sequential operator

HMSC Sequential operators compose bMSCs in sequential order. And the order among events in the same instance from these bMSCs is determined according to the weak sequencing semantics. When generating process behaviors in SDL, the approach refers to the bMSCs in a sequential order. For example, if there is another referenced bMSC, say S2, connected with a sequential operator after bMSC S3 in the example shown in figure 4.5, the approach generates an SDL output construct for sending message Z from S3, followed by another SDL output construct for sending message Y from S2, which follows bMSC S3. The partial HMSC specification and generated process behavior for process P2 is shown in Figure 4.6.

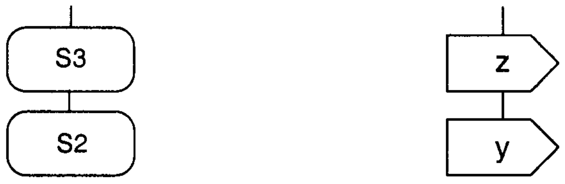


Figure 4.6: An HMSC sequential operator example

4.3.2.2 Alternative operator

An alternative operator is translated into SDL states and/or SDL decisions for each process depending on the role of a process in the alternative scenario. If a process is the initiator of alternative scenarios, i.e. it determines the alternative that is going to be executed; an SDL decisions construct is generated. Otherwise, the approach generates an SDL state, whose name can be determined by the conditions in the MSC specification.

Global Initial Condition

If the operator is preceded by a bMSC global condition or HMSC condition, then this condition becomes the initial condition of operator. SDL state construct is generated for each process in the alternating bMSCs, and the events from each alternating bMSC of the same process becomes a branch in the process behavior in SDL. For instance, if there is a global condition, say "init", following bMSC S1 in the HMSC example in Figure 4.5, then this condition is the initial condition for the following alternative operator. For process P1, which is the initial process, an SDL state construct and a spontaneous transition followed by a non-deterministic decision are generated. For process P2, a state is generated. The partial HMSC specification and generated process behaviors are shown in Figure 4.7.

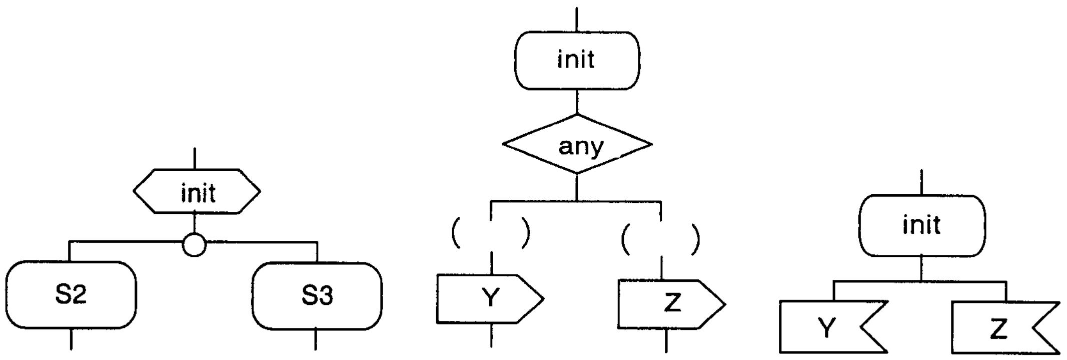


Figure 4.7: An alternative operator with global condition

Local Initial Condition

If there is no Global Initial Condition found, then each instance in the alternating bMSCs will be analyzed separately. If a common condition found for an instance in all the bMSCs, then an SDL state construct is generated for this process with its condition name. Finally, all events of the same process from each alternative bMSC become a branch in the SDL process behavior. Suppose that bMSCs S2 and S3 in the example in Figure 4.5 both have local conditions in the two instances I1 and I2, then the local condition names becomes the state names in SDL specification. Figure 4.8 shows the bMSC specifications and the partial generated process behaviors.

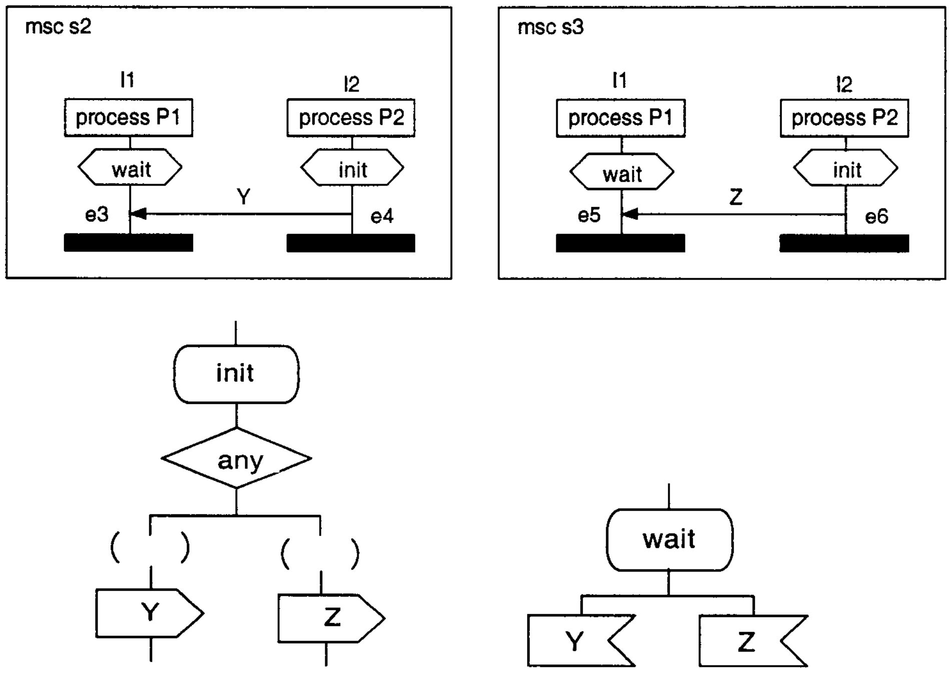


Figure 4.8: An alternative operator with local conditions

No Initial Condition

In the case that there is no Global Initial Condition or Local Initial Condition found, for the initiator process, a non-deterministic SDL decision is generated and all events of the same process from each alternative bMSC become a branch in the SDL process behavior. For a non-initiator process, an SDL state construct is generated with a generated name. This is the case in Figure 4.5. The partially generated process behavior is shown in figure 4.9.

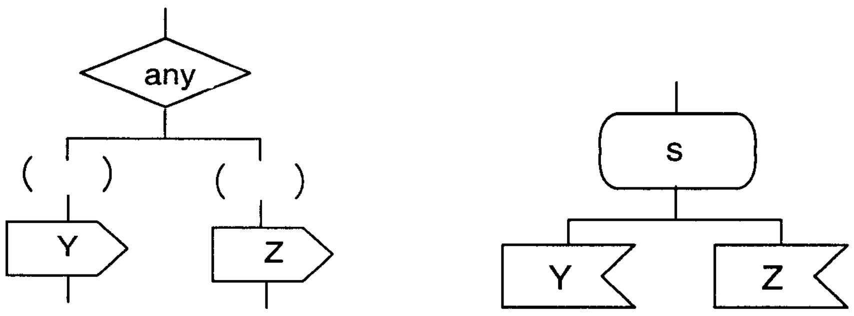


Figure 4.9: An alternative operator with no initial condition

Furthermore, the existing approach supposes all alternative scenarios have to join to a common point after the alternative operator. If the specification after the alternative operator begins with a global condition, local condition or input events, then all alternative scenarios are ended by SDL nextstate construct. Otherwise, an SDL label is generated to refer to the finally condition.

4.3.2.3 Iterative operator

The approach checks the initial state of an Iterative operator in order to refer to at the end of the iteration. In the case that the initial state is neither a condition nor an input event, an SDL label is generated; and after translating the last MSC event in the loop, an SDL join construct with the label name is generated to implement the loop. Otherwise, the approach generates an SDL nextstate construct with the condition name or the state name before the input. The example in Figure 4.5 has an Iterative operator. The partial generated process behavior is shown in figure 4.10.

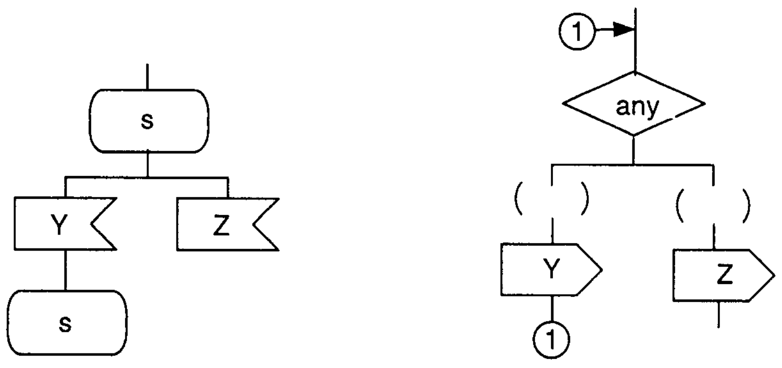


Figure 4.10: Process behavior for iterative operator

Finally, following all procedures and techniques described previously, the process behaviors for the HMSC example in Figure 4.5 can be obtained as shown in Figure 4.11.

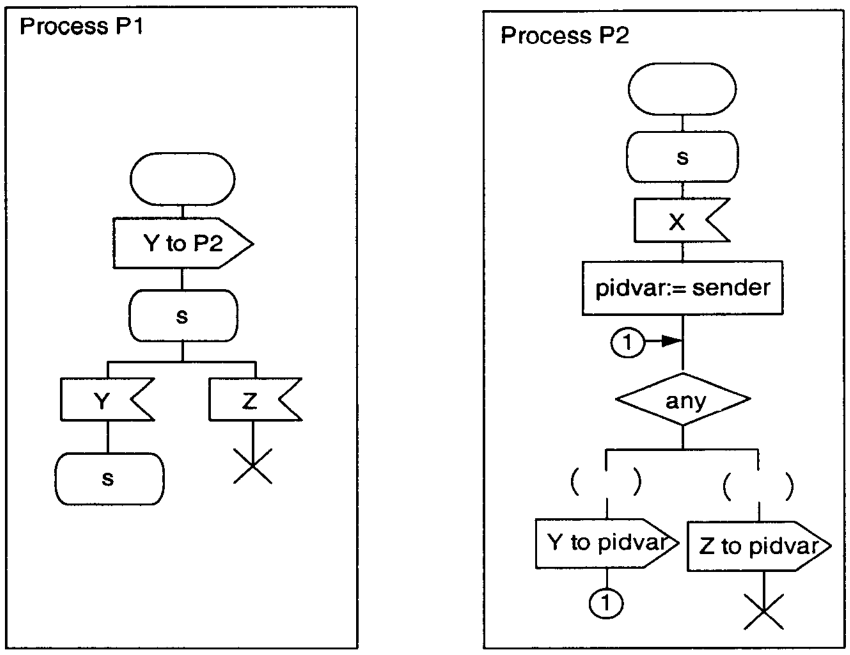


Figure 4.11: Generated process behavior in SDL for MSC of Figure 4.5

The previous research has addressed problems that may be caused by Multi-instances, Shared conditions, and MSC semantic errors, making the existing approach even more complete in terms of semantic checking while translating from MSC to SDL. We do not present all the details in this thesis. A complete discussion can be found in [7].

4.4 Discussion

The existing approach translates MSC-96 specifications into SDL specifications. It handles various MSC events and structures, such as message input/output, action, timer events, instance creation, coregion, and HMSC. It also checks various aspects of MSC semantics before generating SDL. It guarantees correctness of the design, and consistency between the SDL specification and MSC specification.

However, the existing approach has some limitations. First, it does not handle MSC inline expressions and parallel operator in HMSC. Second, some assumptions in the previous work may limit the flexibility of the approach, such as it assumes that all alternative scenarios have to join to a common point after the alternative operator, which is often not the case in HMSC specifications. Moreover, the existing approach cannot handle the newly added concepts in MSC-2000: Time and Data, which enable real-time system specifications and more sophisticated MSC specifications with data manipulation. Our interest is mainly on MSC time concepts. The rest of the thesis illustrates our contributions for translating timed MSC into SDL.

Chapter 5

Timed MSC2SDL

5.1 Introduction

MSC-2000 has introduced time concepts. This new feature makes it possible to specify distributed real-time system with MSCs. Generating SDL specification from timed MSC specification becomes our goal and new challenge. The translation of timed MSC specification to SDL specification will ease the transition from the requirement to the design of real-time software processes.

Time requirements in MSCs may include time offsets for the systems, absolute and relative time constraints or measurements on events, and requirements through the use of timers. For example, the MSC specification in Figure 5.1 has two time requirements. One is a relative constraint specified between event e0 and e1, the other is a relative time measurement between event e0 and e2. The previous approach that we have discussed in Chapter 4 does not handle such a MSC specification. First of all, there is no internal structure that keeps track of the time requirements information. Second, the mapping algorithm does not take time requirements into consideration.

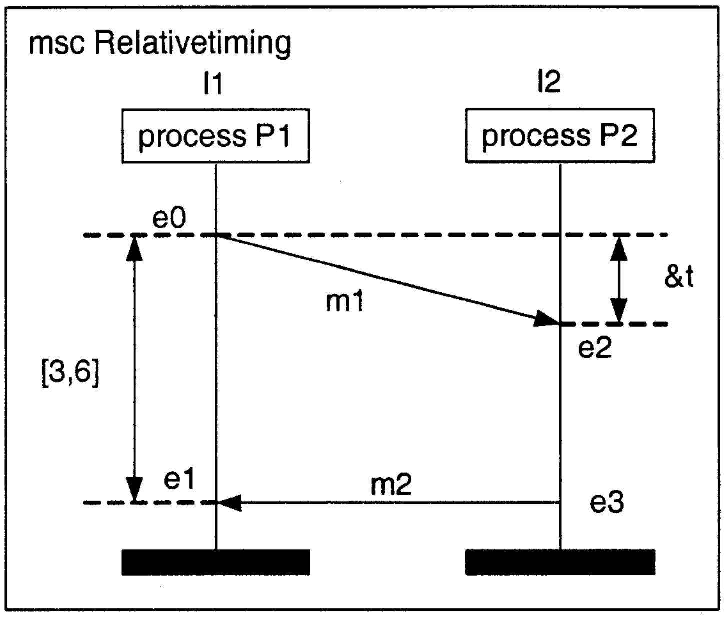


Figure 5.1: A simple MSC specification with real time requirements

In this chapter, we present the categorization of time requirements in MSC specification, re-designed and newly added internal structures that keep track of time information.

In terms of mapping MSC time requirements to SDL, we have studied the time related features in SDL. SDL has the notion of global time (allowing to measure durations throughout the system by means of appropriate time stamps) and allows time dependent decisions in the functional design (timeouts and time dependent enabling conditions allow to define constraints on the triggering time) [19]. These features make possible to specify time constraints and measurements from timed MSCs in SDL. Consequently, a new translation algorithm has been introduced.

5.2 Enhanced Event Order Table

Timed MSC has real-time requirements that the system must satisfy. The time requirements are associated with events, such as absolute time constraints on individual events, or relative time constraints between pairs of events. In order for the generated SDL specifications to satisfy these time requirements, we need to map them to SDL constructs associating to the constrained events. As we described in the previous chapter, the translation is mainly based on mapping from MSC events to SDL constructs. Therefore, if we record all time requirements for an MSC event, at the time of mapping this event to SDL constructs, the associated time requirements can be translated at the same time. For example, when translating the output event of message m1 in the MSC specification of Figure 5.1, we can generate the necessary SDL constructs to specify the two time requirements associated to it, provided that we have the time requirement information at the translation time.

To gather time requirements for each MSC event, one choice is to design some structure to record the time requirements in MSC. However, considering all time constraints and measurements are related to one event or a pair of events, and that Event Order Table contains all input and output events and indicates pair wise order among the events, a good choice is to re-design the Event Order Table and add timing information into it.

5.2.1 Enhanced Event Order Table structure

Besides containing order information of the corresponding to the row and column events, the modified Event Order Table cells should also contains time requirements associated with the pair of events. They should include whether a time requirement is a time constraint or measurement, and whether it is of absolute time or relative time. Other information such as time values and the boundary inclusion of the time values, and user defined time variable names in case of measurements also need to be included. Now, each cell of the table, instead of being only a boolean value indicating pair wise event order, becomes an object that encapsulates the following information:

Precedence, which takes value "T" to indicate pair wise event order.

Time Interval, which may record a relative time constraint, or a relative time measurement between the row and the column event of a cell. In the case of absolute time measurements or constraints specified for individual events, they are recorded in the cells that the corresponding to the identical row and column events (which are the cells on the diagonal of the table and are not used in the existing approach). Symbols "&" and "@" are used to indicate whether the real-time requirement is relative or absolute. A time interval may contain two time values used as upper bound and lower bound of the time constraint, or contain a time variable used to store the time measurement result. Furthermore, information about whether that the time bounds are inclusive or not is also recorded, which is indicated by left/right brackets or braces.

MSC inline expressions can be specified with time requirements; we will discuss them in detail in the following sections. Here we want to incorporate order information of events in inline expressions in to this extended Event Order Table. We make a modification to the Precedence attribute in the Event Order Table cells. If an event is in an opt inline expression, it is optional to the whole MSC. Therefore, we use "O" as the Precedence to indicate this property in its corresponding cell (both row and column events correspond to the same event); if two events are in different operands of an alt or exc inline expressions, they are exclusive in terms of the execution traces. Therefore, we use "E" as the Precedence to indicate this property in the cells corresponding to these two events.

We call this re-designed Event Order Table an Enhanced Event Order Table.

Figure 5.2 shows an example of a timed MSC with different types of real-time requirements, such as absolute time measurements and constraints, relative time measurements and constraints. All the time information is recorded properly in the Enhanced Event Order Table shown as Table 5.1.

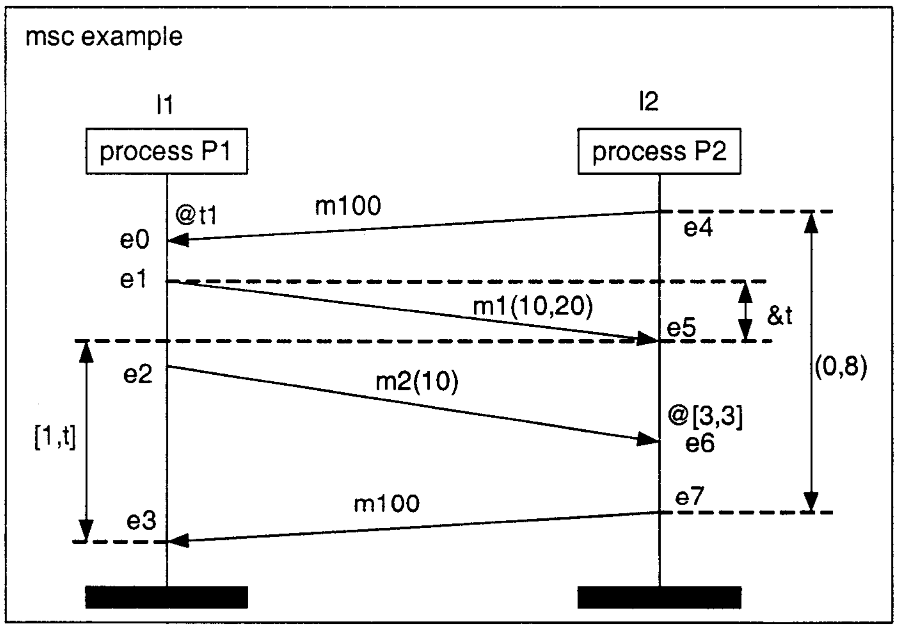


Figure 5.2: A timed MSC specification

	e0	e1	e2	e3	e4	e5	e6	e7
e0	@t1	T	T	T		T	T	T
e1			T	T		T &t	T	T
e2				T			T	T
e3								
e4	T	T	T	T		T	T	T&(0,8)
e5				T&[1,t]			T	T
e6				T			@[3,3]	T
e7				T				

Table 5.1: The Enhanced Event Order Table of MSC of Figure 5.2

For instance, in cell (e0, e0), @t1 means an absolute time measurement is specified for event e0. In cell (e1, e5), T indicates that e1 occurs before e5, there is also a relative time measurement and the value will be saved into variable t. In cell (e4, e7), a relative time constraint is recorded for these two events. The time boundaries are 0 and 8; neither of the boundaries is inclusive. In cell (e5, e3), a relative time constraint using the previous measurement result is recorded for these two events. Finally, event e6 has an absolute time constraint, which is saved in cell (e6, e6). Both time boundaries are 3, inclusive.

5.2.2 Time requirements on events in coregions

Time requirements can be specified on events inside a coregion. Since the order of the events in a coregion is relaxed, either of the two events associated with a relative time requirement can occur before the other one. The existing approach marks "T" in both cells associating to the two events if a relative time requirement exists. In SDL process behavior, each possible trace becomes an alternative branch. In order to find time requirement information easily through either of the two events in the SDL generation phase, we record the time information in both cells in the Enhanced Event Order Table.

5.2.3 Time requirements for inline expressions

Time constraints can be specified with MSC inline expressions, which actually constraints the starting and/or ending events of the inline expressions. In this case, building Enhanced Event Order Table needs one more step. That is to determine the starting and/or ending events of the inline expressions by using order information recorded in the Enhanced Event Order Table. There are three types of time constraints with inline expressions. One is of time constraints specified between the starting events of an inline expression and another event outside of the expression. In this case, the starting events of each operand need to be decided (if possible) and the time constraints are specified in the cells corresponding to each of the starting events and the other event. The second is of time constraints specified between the ending events of the inline expression and another event outside of the expression. In this case, the ending events of each operand need to be decided (if possible) and the time constraints are specified in the cells corresponding to each of the ending events and the other event. The third is of time constraints specified between the starting and ending events of the inline expression. In this case, both the starting and ending events of each operand need to be decided (if possible) and the time constraints are specified in the cells corresponding to the starting and ending event of each operand.

Figure 5.3 shows an MSC specification with time constraints specified for an alt inline expression. When building the Enhanced Event Order Table, we first complete filling the order information into the table and obtain an intermediate table shown as Table 5.2.

Note that since event e1 is in one operand of the alt inline expression; e2 is in the other operand of the same inline expression. Therefore, the Precedence attributes in the corresponding cells (e1, e2) and (e2, e1) are indicated by "E". The same happens to event e5, e2, etc.

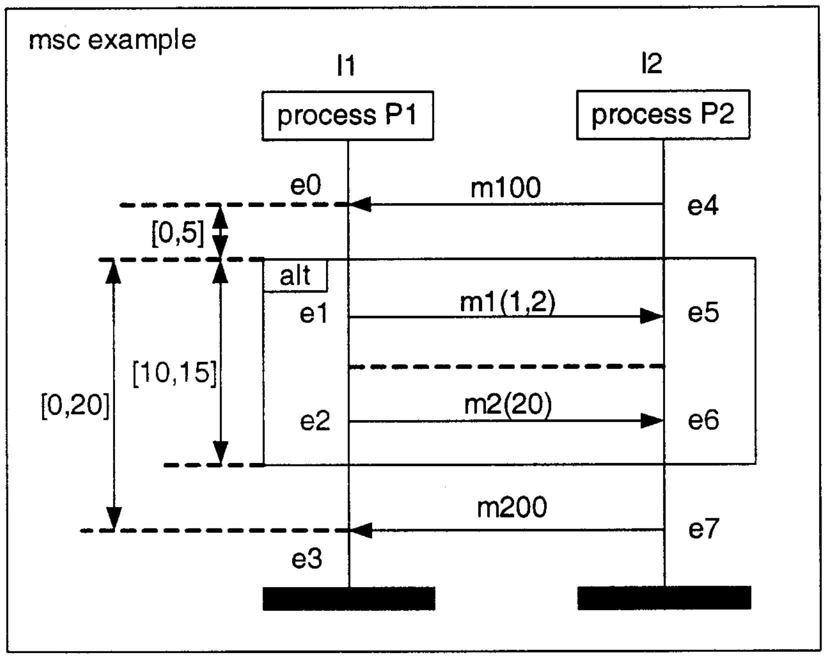


Figure 5.3: An MSC with a timed inline expression

	e0	e1	e2	e3	e4	e5	e6	e7
e0		T	T	T		T	T	T
e1			E	T		T	E	T
e2		E		T		E	T	T
e3								
e4	T	T	T	T		T	T	T
e5			E	T			E	T
e6		E		T		E		T
e7				T				

Table 5.2: Event Order Table with inline expression timing information

By using the order information in this table, for the alt expression, we can identify that for the first operand, the starting event is e1 and the ending event is e5, for the second operand, the starting event is e2 and the ending event is e6. Therefore, time constraint information can be added into proper table cells and result Table 5.3, the final Enhanced Event Order Table.

	e0	e1	e2	e3	e4	e5	e6	e7
e0		T&[0,5]	T&[0,5]	T		T	T	T
e1			E	T&[0,20]		T&[10,15]	E	T
e2		E		T&[0,20]		E	T&[10,15]	T
e3								
e4	T	T	T	T		T	T	T
e5		E		T			E	T
e6		E		T		E		T
e7				T				

Table 5.3: The Enhanced Event Order Table of MSC of Figure 5.3

5.2.4 Time offset

Time offsets can be specified to MSCs. According to MSC semantics, a time offset offsets all absolute time values in one MSC. Therefore, all absolute time constraints in the Enhanced Event Order table become their original values plus the time offset. For example if a time offset 5 is specified for the MSC shown in figure 5.2, then in the table cell corresponding to e6 in table 5.1, the absolute time constraint values become [8,8].

5.3 Occupancy Table

Before generating process behaviors in SDL, Occupancy Tables are built for the purpose of saving signals and avoiding possible deadlocks. The Occupancy Table structure remains the same as in the existing approach. However, the building algorithm has been slightly modified because of the two following reasons.

Events in loops

Events in a loop inline expression are executed repeatedly each iteration, which may cause the need of more signal saves. For example, in the MSC specification shown as Figure 5.4, instance I2 receives message m1 and m2 repeatedly and we suppose m1 and m2 travel different channels. We can get the Enhanced Event Order Table shown as table 5.4. Using the existing condition for building Occupancy Tables that has been presented in the previous chapter, we have NOT (e4<<e2) AND NOT (e2<<e3). Therefore m2 should enter the row of event e2 of the Occupancy Table for instance I2.

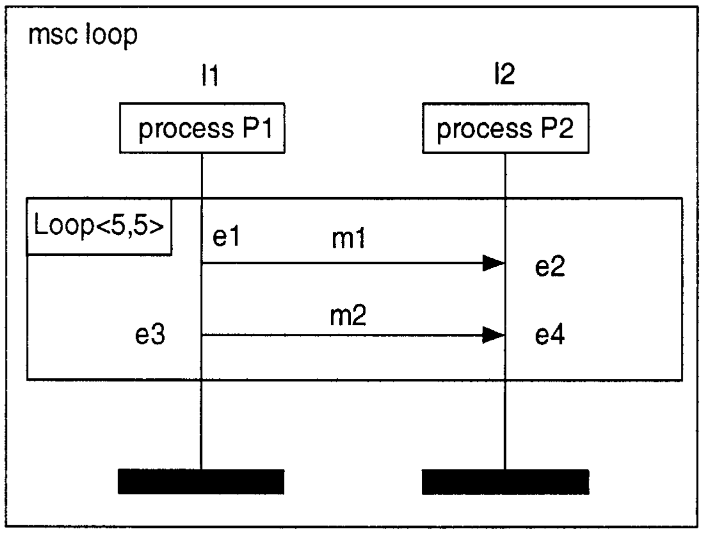


Figure 5.4: An MSC with a loop

	e1	e2	e3	e4
e1		T	T	T
e2				T
e3				T
e4				

Table 5.4: Event Order Table for an inline expression without the loop unfolded

However, when instance I2 consumes message m2, m1 for the next loop iteration may be at the head of I2's input queue. Therefore, m1 should also be in the row of event e4. By using the order information in the Enhanced Event Order Table, this result cannot be obtained directly. To solve this problem, we unfold the loop once (shown as Figure 5.5) and build a local Event Order Table for the loop (shown as Table 5.5).

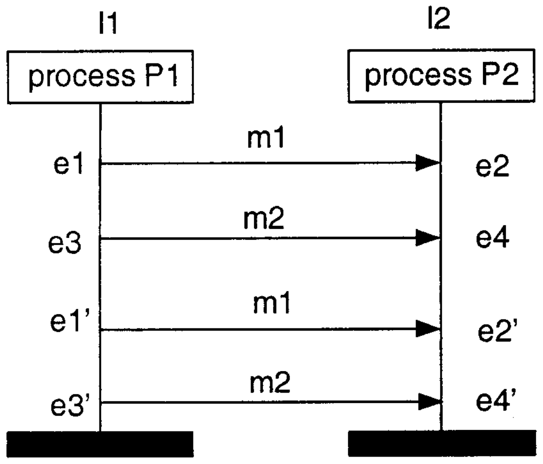


Figure 5.5: MSC of Figure 5.3 with the loop unfolded

	e1	e2	e3	e4	e1'	e2'	e3'	e4'
e1		T	T	T	T	T	T	T
e2				T		T		T
e3				T	T	T	T	T
e4						T		T
e1'						T	T	T
e2'								T
e3'								T
e4'								

Table 5.5: A local Event Order Table with the loop unfolded

Events e1', e2', e3' and e4' are duplicates of e1, e2, e3, and e4. They are used for evaluating the conditions in building the Occupancy Tables. For example, at the row of event e4, we have NOT (e2'<< e4) AND NOT (e1'<<e4) evaluating true. Therefore, the message received by event e2' (m1) enters the row of event e4, which will result a signal save for m1 when instance I2 consumes signal m2 in the SDL generating phase.

Events in alt inline expressions

The Precedence between a pair of events in the Enhanced Event Order Table is no longer a boolean variable as described in the previous section; it may not only hold true or false, but also exclusive for a pair of events that can not occur in the same execution trace.

The condition for input event Er enters the row of Ec becomes

(NOT (e_r<< e_c) AND NOT (e_c << e_s)) AND (NOT ($e_r \Phi e_c$) AND NOT ($e_c \Phi e_s$))

where

e_c denotes the current row event in the Occupancy Table.

e_r denotes the current input event.

e_s denotes the corresponding output event of e_r.

<< denotes the order relation (preceding).

Φ denotes the order relation (exclusive).

The reason of adding more constraints to the existing condition is that we want to make sure the input events are not exclusive to the row event. Because if they are exclusive, they can never both occur in the one execution trace, there is no need for one input event to save the signal of the other.

5.4 Mapping between bMSC and SDL

Process behaviors are generated based on one to one mapping from MSC constructs to SDL constructs in most cases. Therefore, for some basic events such as MSC action, input/output without time constraints, etc, translation can be done directly; and this has been discussed in [7]. In this section, we will discuss how time-related MSC events can be mapped to SDL process behaviors according to different categories of time requirements.

5.4.1 Absolute time

MSC assumes global clock. All instances in one MSC specification have a common global clock value at any given time. The clock values can be used to constrain an event or measured at the time that the event occurs. In SDL, the global time also exists. Each process instance can access it through the construct **now**, which holds the current global time value. Therefore, a direct mapping from MSC global time to SDL global time exists.

5.4.1.1 Absolute time constraint

Absolute time constraints using global clock values can be specified for individual MSC events. Semantically, such a time constraint requires the event occurs at the specified time or within the specified time range. In SDL, one can use conditions to constrain transitions of processes. Specifically, these conditions are in form of continuous signals or enabling conditions. According to the SDL semantics, such a constrained transition occurs only when the condition is satisfied. Otherwise, the transition is delayed or fails. Therefore, conditional transitions can be used to map MSC absolute time constraints of events.

For example, Figure 5.6(a) and 5.6(b) show an MSC with absolute time constraints and the corresponding process behavior design in SDL. A continuous signal is formed for event e1 using its absolute time constraint @[2,4] to enable the transition to occur, so that process P1 can send signal m1. An enabling condition is also formed for event e4, which enables the transition to occur only if the absolute time is 6 when process P1 receives signal m2.

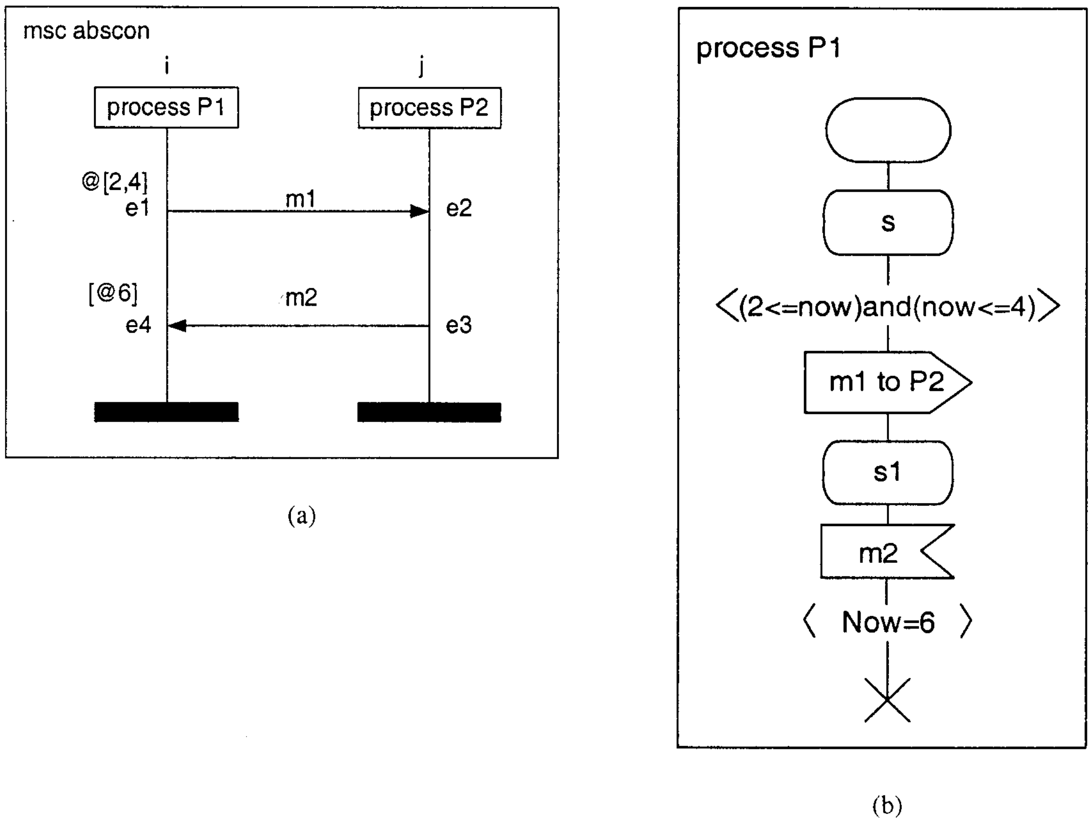
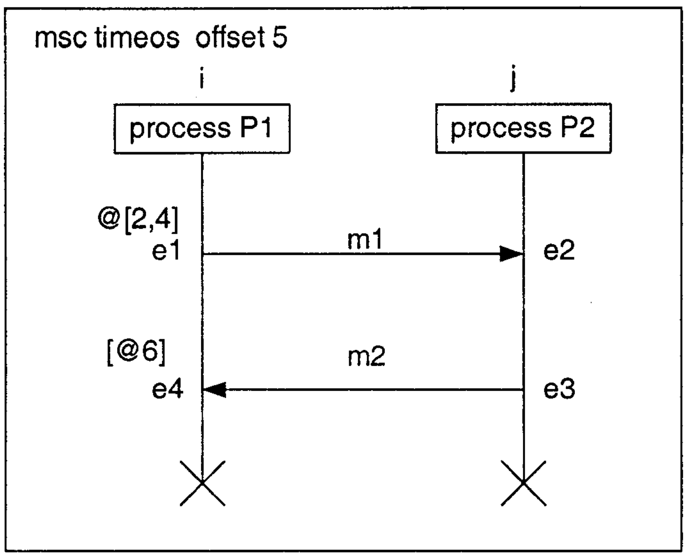


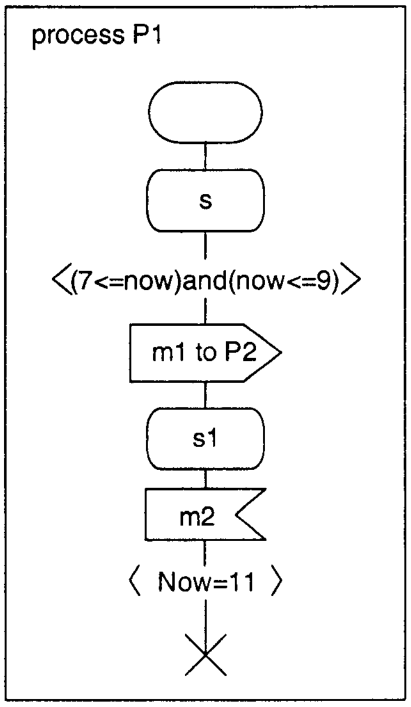
Figure 5.6: An MSC with absolute time constraints and the process behavior in SDL

5.4.1.2 Time offset

In the case that a time offset is specified for an MSC specification, the time offset offsets all the absolute time values in this MSC. The Enhanced Event Order Table contains the absolute time constraint values obtained from their original values plus the time offset. Therefore, generating process behaviors in SDL can be done transparently by taking absolu e time values in the Enhanced Event Order Table. An MSC example is shown in Figure 5.7(a), which is obtained from adding a time offset 5 to the example shown in Figure 5.6(a). The behavior of process P1 in SDL is shown in figure 5.7(b), where the resulted absolute time constraints become their original values plus the time offset.



(a)

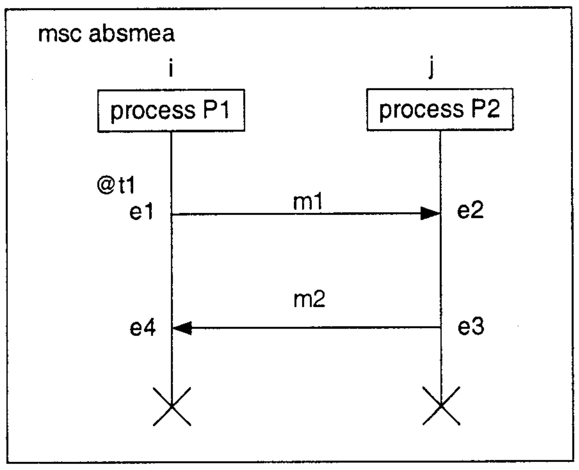


(b)

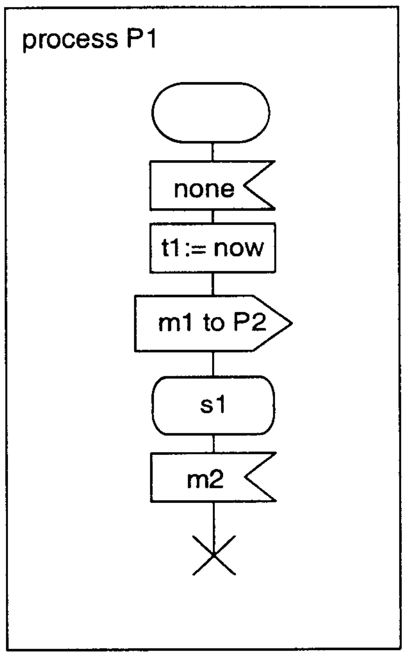
Figure 5.7: An MSC with a time offset and the process behavior in SDL

5.4.1.3 Absolute time measurements

The global time at which an event occurs can be measured using MSC time measurement. As we explained previously, global clock values can be accessed through the construct **now** in SDL. Moreover, the values of **now** can be assigned to variables of time type or real type through SDL task constructs. This requires that the specified time patterns to store the measurement results must be declared as time or real variables in the corresponding processes. In Figure 5.8(a), the absolute time of event e1 needs to be measured and stored into the time variable t1. In figure 5.8(b), an SDL task construct is used to assign the value of **now** to time variable t1, which needs to be declared in process P1.



(a)



(b)

Figure 5.8: An MSC with an absolute time measurement and the process behavior in SDL

5.4.2 Relative time

A relative time requirement relates a pair of events. It specifies the time difference between the occurrences of the two events. To obtain the duration, we need to calculate the difference between the global time at which the first event occurs and that of the second event. As we have discussed before, the global time can be obtained through construct **now** in SDL. Therefore, we can first record the global time when the first event occurs. When the second event occurs, we calculate the difference between the global time **now** and recorded time for the first event. Now, there are two problems we may encounter in doing above.

The first is that since variables belong to individual SDL processes, if two events are on the process, then the time variable storing the occurrence time of first event is accessible to calculate the duration when the second event occurs. However, when the two events are not on the same process, then the time variable, belonging to the process on which the first event occurs, is not accessible to the process on which the second event occurs, when the duration need to be calculated.

The second problem is that when the second event occurs, we have to the take the time variable storing the occurrence time of the corresponding first event to calculate the duration. In order for the time duration can be calculated correctly with the right time variables, we need a mapping from time variables to their corresponding events since there may be many such variables in our context.

5.4.2.1 Signal extension

To solve the first problem mentioned previously, we consider that the occurrence time of the first event of a relative time requirement must be known to the process that the second event is on when the second event occurs. This can be achieved through some message passing by having the time value sent from one process to another. The assumption here pertaining to the translation from MSC to SDL is that we do not intend to add extra message passing to the existing MSC specification, but rather, to find existing message passing to convey the time information needed as the message parameters.

There are two types of message passing can be used for conveying time information from one process to another.

(1) A proper message passing between the two processes can be found to carry the timing information.

(2) A proper message passing between the two processes cannot be found but a chain of message passing (via some intermediate processes) can be found to relay the necessary timing information.

For example, Figure 5.9 shows an MSC with relative time requirements. For the relative time measurement between event e3 and e1, a direct message passing can be found to convey a time value at which e3 occurs from process P2 to P1, which is the sending signal of m1. For the time constraint between event e2 and e6, no direct message passing from process P1 to P3 can be found, but a message passing chain can be used to convey a time value, which is the sending of m2 from P1 to P2, and the sending of m3 from P2 to P3.

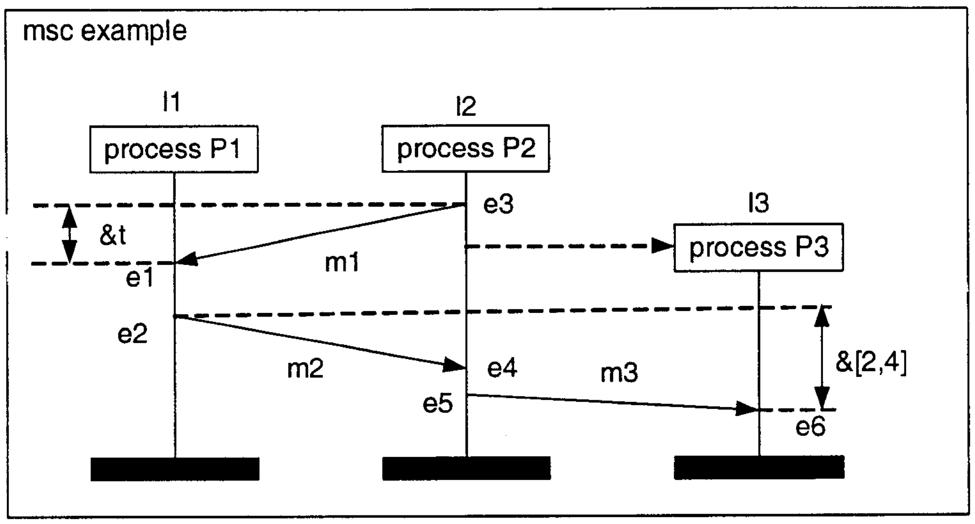


Figure 5.9: Example of Proper Message passing in a timed MSC

However, even if a message passing or a chain of message passings can be found from one process to another, it does not ensure that they are proper to convey time value for a relative requirement. In order for a message passing or a chain of message passings to be proper for a relative time requirement, the time information from the process in which the first event occurs has to be delivered to the process in which the second events occurs at the time that is not later than the occurrence of the second event. That is, suppose the two events involved in a time requirement are e1 and e2, e1 precedes e2; e1 is on process P1, and e2 is on process P2.

(1) If there is a message passing m from P1 to P2 with the sending event e3 and receiving event e4. For message passing m being proper to convey the timing information, e3 must not precede e1 and e4 must not succeed e2. Moreover, e3 may be e1 and e4 may be e2.

(2) If there is a message passing from P1 to P3 with the sending event e3 and the corresponding receiving event e4; there is another message passing from P3 to P1 with the sending event e5 and the receiving event e6. Then the message passing chain can [illegible] proper to convey the timing information if e3 does not precedes e1, e4 precedes e5 and e2 does not succeed e6. Likewise, e3 may be e1 and e6 maybe e2. Similarly, this chain could have more than two messages passing with more intermediate processes involved.

For example, there are two relative time requirements in the MSC shown in Figure 5.10. Even though there is a direct message passing m1 from process P2 to P1, it is not proper to convey time value for the time measurement between event e4 and e1. This is because the consumption of m1 succeeds event e1, and any time value conveyed by m1 is available only when e2 occurs. It is too late to calculate the time duration between e1 and e4. Similarly, message-passing m1 from P2 to P1 is not proper for relative time constraint between event e5 and e3. The reason is when m1 is sent, the occurrence time for e5 is not available since e4 precedes e5.

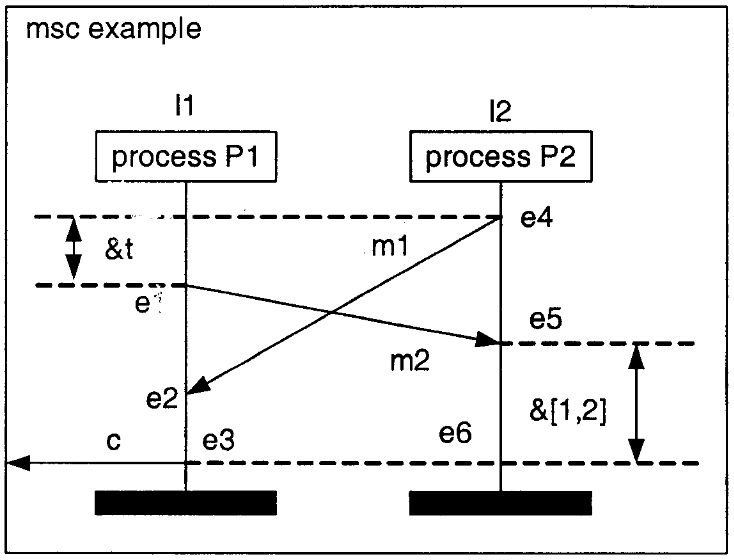


Figure 5.10: A timed MSC with no proper message passing for signal extension

To find proper message passing or chain of messages passing, we need only the order information in the Enhance Event Order Table, with which we can examine each pair of sending and receiving events to determine if they are proper for a relative time requirement according the above two rules. Of course finding a proper message passing chain may be of high complexity in computing. Since all possible message passing combinations have to be tested until a proper message chain is obtained. After finding the proper messages passing, we can extend their parameter lists to incorporate the needed timing value for the relative time requirements, hence the signal extension. For the example in Figure 5.9, signal m1 needs to be extended as m1 (time) to convey a time value from process P2 to P1. Similarly, signal m2 and m3 need to be extended same way since they form a message passing chain to convey a time value from Process P1 to P3.

Finally, if any signals are extended, in order to make the generated SDL system syntactically correct, these signals must be declared as the extended format in their scopes. Moreover, all messages passing of these signals must be in the same format even if some of them do not need to convey any time value for relative time requirements. For example, suppose we add another message passing of m1 from process P3 to P1 following event e6 in Figure 5.9, in the generated process behaviors, both P3 and P1 have to use the extended m1 format for this newly added signal passing, and the parameter in this signal is a dummy. We need to declare a dummy variable on each of the processes as well.

So far, the problem of making the time value known to another process has been solved with signal extension. However, a new problem comes with this solution. Signal extension is done before generating process behaviors; the sending event and receiving event of an extended signal are not always the pair of events that are involved in the relative time requirements. Furthermore, process behaviors are generated one process after another; the two events involved in one relative time requirement may be not in the same process. We need to know, at the generating time, the mapping from a signal extension to a relative requirement. This is similar to the problem of mapping declared time variables to relative time requirements that we have mentioned in the previous section. Both of the problems can be solved if we design a schema that relates a relative time requirement to the extended signal and the corresponding time variables.

5.4.2.2 Handling time variables

In order to record the relationship among a time constraint, events associated with the constraint, the extended signal with its sending and receiving events, and corresponding time variables, we have proposed a structure, called Variablemap. The Variablemap records all necessary information for a relative time requirement when it is translated into process behavior in SDL.

A Variablemap is tuple including the following elements:

(1) Frontevent: the first event involved in a time constraint, which precedes the second event in time.

(2) Backevent: the second event involved the time constraint, which succeeds the first event in time.

(3) Carryoutevent: the sending event associated with the extended message passing.

(4) Carryinevent: the receiving event associated with the extended message passing.

(5) FrontVariable: a time variable generated for the Frontevent to save its occurrence time, which may be sent to another process by Carryoutevent along the extended signal.

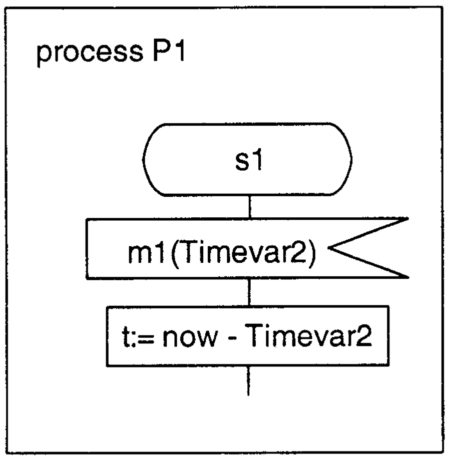
(6) CarryVariable: a time variable generated for Carryinevent to bind the value of FrontVariable that is sent along the extended signal.

(7) UserVariables: the user defined variables associated with the time constraint or measurement.

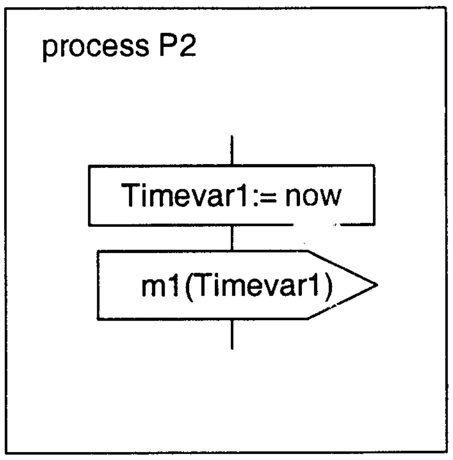
(8) StampVariable: a time variable generated for recording the occurrence time of the Backevent.

All generated time variables are unique.

For example, let's consider the relative time measurement between event e3 and e1 in Figure 5.9 and see how the Variablemap is built and used in the SDL generation phase. Since event e3 precedes e1, then the Frontevent takes e3 and the Backevent takes e1. In order to record the occurrence time of e1, we need a FrontVariable, say Timevar1. After taking the global time value, it is sent by the extended signal m1 (Timevar1). The Carryoutevent takes e3 and the Carryinevent takes e1, since they are the sending and receiving event of m1. Upon consumption of the signal m1 by process P1, it has to bind the value carried by Timevar1 to local time variable, say, Timevar2, which is the CarryVariable. Finally the UserVariables takes t, which is defined by the user and used to store the measurement result. With this Variablemap built beforehand, process behavior can be generated easily through looking up the needed information in it. When we generating process behavior for P1, since we know that event e1 is involved in a relative time measurement, we simply declare a time variable called Timevar2. Because the Carryinevent is also e1, we generate an SDL input construct with the extended signal format m1 (Timevar2), then an SDL task construct to bind the difference of **now** and the value of Timevar2 into the UserVariables t. For process P2, similarly, we declare a time variable Timevar1, then generate an SDL task construct to record the occurrence time of e3. Because the Carryoutevent is e3, we generate an SDL output construct with the extended signal format m1 (Timevar1). Figure 5.11 shows the partial behaviors of the two processes described above.



(a)



(b)

Figure 5.11: Partial process behavior generated by using Variablemap

Variablemaps can also be used for absolute time measurements to keep track the user defined variables. In Fact, our approach generates a Variablemap for each time requirement in an MSC specification. The advantage of doing this is that we only need to search the Variablemaps to know whether a time requirement is specified on an event in the SDL generation phase. The following are the rules for generating Variablemaps.

(1) If an absolute time measurement is processed, the Frontevent takes the first event and the UserVariables takes the user defined variables to store the measurement result. All other attributes remain null.

(2) If a relative time measurement is processed, the Frontevent takes the first event; the Backevent takes the second event; a FrontVariable is generated for the first event to store its occurrence time value in the generation phase; the UserVariables takes the user defined variable to store the measurement result.

- If the two events are on the same instance, then all other attributes remain null.

- If the two events are not on the same instance, then a signal extension must be resolved. The Carryoutevent takes the sending event of this message passing, and the Carryinevent takes the receiving event of this message passing. Furthermore, a CarryVariable is also generated for binding the time value in the generation phase.

(3) If an absolute time constraint is processed, the Frontevent takes the only event involved and the UserVariables takes the user defined variables for the time constraint, or it remains null if the constraint is specified with concrete time values. All other attributes remain null.

(4) If a relative time constraint is processed, the Frontevent takes the first event; the Backevent takes the second event; a FrontVariable is generated for the first event to record its time value in the generation phase, the UserVariables takes the user defined variables that specify the constraint, or it remains null if the constraint is specified with concrete time values.

  - If the two events are on the same instance, then all other attributes remain null.
  - If the two events are not on the same instance, then a signal extension must be resolved. The Carryoutevent takes the sending event of this message passing, and the Carryinevent takes the receiving event of this message passing. Furthermore, a CarryVariable is also generated for binding the time value in the generation phase. One important point is that if the Backevent is also the Carryinevent, then a StampVariable must also be generated for recording the occurrence time of the second event. This is because the value carried by a

signal cannot be used in its enabling condition [15]. Therefore, we use a conditional signal in the following SDL state to check whether the time constraint is met with the values of the StampVariable and the CarryVariable. Examples showing this can be found in both section 5.3.3 and 5.3.5.

Note that one time constraint may have more than one Variablemap in the case that a chain of message passing is involved.

In the SDL generating phase, the Variablemaps are referenced to ensure that each event associated with a time requirement is generated using the proper variables and signal extensions. We will describe how the Variablemaps are used with example in the following sections. Here are some examples about how Variablemaps are built according to above rules.

If we consider the scenario in Figure 5.2, some Variablemaps can be formed as the following.

For the absolute time measurement at e0, the Frontevent takes event e0, UserVariables takes user defined variable t1, and all other attributes remain null.

For the relative time measurement between event e1 and e5, the Frontevent takes the first event e1; the Backevent takes the second event e5. The extended signal should be message m1 with sending event e1 and receiving event e5. Therefore, the Carryoutevent takes event e1; the Carryinevent takes e5. The FrontVariable is generated and called timevar0 for storing the global time at e1; the CarryVariable is also generated and called timevar1 for binding the value of the FrontVariable upon message consumption. The UserVariables takes the user defined variable t. since e5 is both the Carryinevent and the Backevent; a StampVariable timevar5 is also generated. Note that the original message m1 has format m1 (integer, integer). After signal extension, it becomes m1 (integer, integer, time) in the generated process behaviors.

For the relative time constraint between event e4 and e7, the Frontevent takes the first event e4, the Backevent takes the second event e7. Since the two involved events are on the same process, no signal extension is needed. Therefore, both the Carryoutevent and the Carryinevent remain null, the generated FrontVariable is timevar2; the CarryVariable remains null, and there is no UserVariables.

For the relative time constraint between event e5 and e3, the Frontevent takes the first event e5, the Backevent takes the second event e3. The signal to be extended is m100, then the Carryoutevent takes its sending event e7 and the Carryinevent takes its receiving event e3. The generated FrontVariable is timevar3, and the CarryVariable is timavar4; finally the UserVariables is t as specified. A StampVariable timevar6 is also generated because e3 is both the Carryinevent and the Backevent.

All Variablemaps are kept in the Variablemap Table shown as Table 5.6, and the information is accessed in the SDL generation phase, which we will continue to discuss in the following sections.

Front_ event	Back_ event	Carryout_ event	Carryin_ event	Front_ Variable	Carry_ Variable	User_ Variables	Stamp_ Variable
e0	NIL	NIL	NIL	NIL	NIL	t1	
e1	e5	e1	e5	timevar0	timevar1	t	timervar5
e4	e7	NIL	NIL	timevar2	NIL	NIL	
e5	e3	e7	e3	timevar3	Timevar4	t	timevar6

Table 5.6: The Variablemap Table of MSC of Figure 5.2

5.4.2.3 Relative time measurements between two events in the same instance

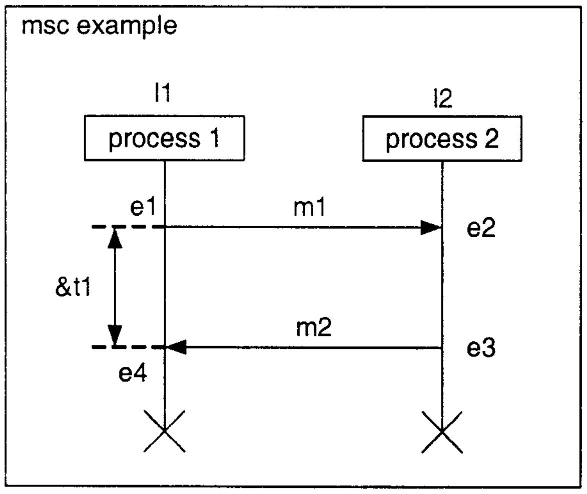
An MSC relative time measurement is used for the observation and recording of the distance between the occurrence times of a pair of events. When a time measurement is specified between two events in one MSC instance, the variables used for the calculation are local to its corresponding process. Indeed, the Variablemap contains only the FrontVariable, the UserVariables, the Frontevent and the Backevent. In generating the process behavior, at the Frontevent, a SDL task construct is generated to record the current global time into the FrontVariable through **now,** which indicates the time at which the Frontevent occurs. Then at the Backevent, the construct **now** holds the occurrence time of the second events; another task construct is generated to calculate the measurement result by subtracting the value of the FrontVariable from **now**, and bind the result to the UserVariables.

In Figure 5.12(a), the time interval between events e1 and e4 need to be measured. Its corresponding Variablemap is shown as Table 5.7.

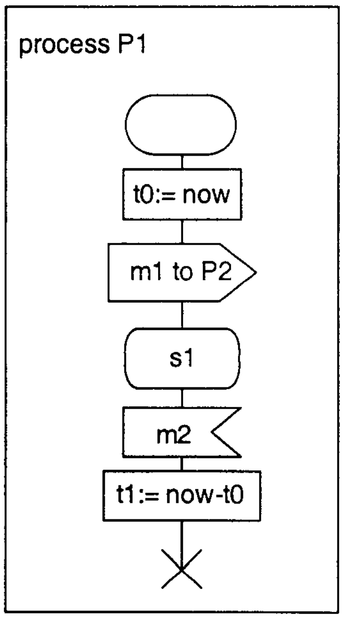
Front_ event	Back_ event	Carryout_ event	Carryin_ event	Front_ Variable	Carry_ Variable	User_ Variables	Stamp_ Variable
e1	e4	NIL	NIL	t0	NIL	t1	NIL

Table 5.7: The Variablemap Table of MSC of Figure 5.12(a)

In Figure 5.12(b), we can see that at the Frontevent e1, an SDL task construct is generated to record the current global time **now** into the FrontVariable t0. And at the Backevent e4, we also generate an SDL task construct to calculate the difference between the current global time **now** and the time value stored of the FrontVariable; the result is bound into the UserVariables t1.



(a)



(b)

Figure 5.12: An MSC with relative time measurement within one instance and process behavior in SDL

5.4.2.4 Relative time constraints between two events in the same instance

An MSC relative time constraints is used for specifying a duration requirement between the occurrence times of a pair of events. Similar to a relative measurement, when a time constraint is specified between two events in one MSC instance, the variables used for the calculation are local to its corresponding process. The Variablemap contains only the FrontVariable, the UserVariables, the Frontevent and the Backevent. In generating the process behavior, at the Frontevent, a SDL task construct is generated to record the global time into the FrontVariable through **now,** which indicates the time at which the Frontevent occurs. Then at the Backevent, the construct **now** holds the occurrence time of the second event; and an SDL continuous signals or enabling condition construct is generated to enforce the time constraint with the condition checking whether the difference between the value of the FrontVariable and **now** is within the specified range. The time constraint range can be concrete time value or using user defined variables.

In Figure 5.13(a), a time constraint is specified between events e1 and e4 using concrete time values. Its corresponding Variablemap is shown as Table 5.8.

Front_ event	Back_ event	Carryout_ event	Carryin_ event	Front_ Variable	Carry_ Variable	User_ Variables	Stamp_ Variable
e1	e4	NIL	NIL	t0	NIL	NIL	NIL

Table 5.8: The Variablemap Table of MSC of Figure 5.13(a)

In Figure 5.13(b), we can see that at the Frontevent e1, an SDL task construct is generated to record the current global time **now** into the FrontVariable t0. And at the Backevent e4, we generate an SDL enabling condition checking whether the difference between the value of FrontVariable and the current global time **now** is within the constraint range (2, 6).

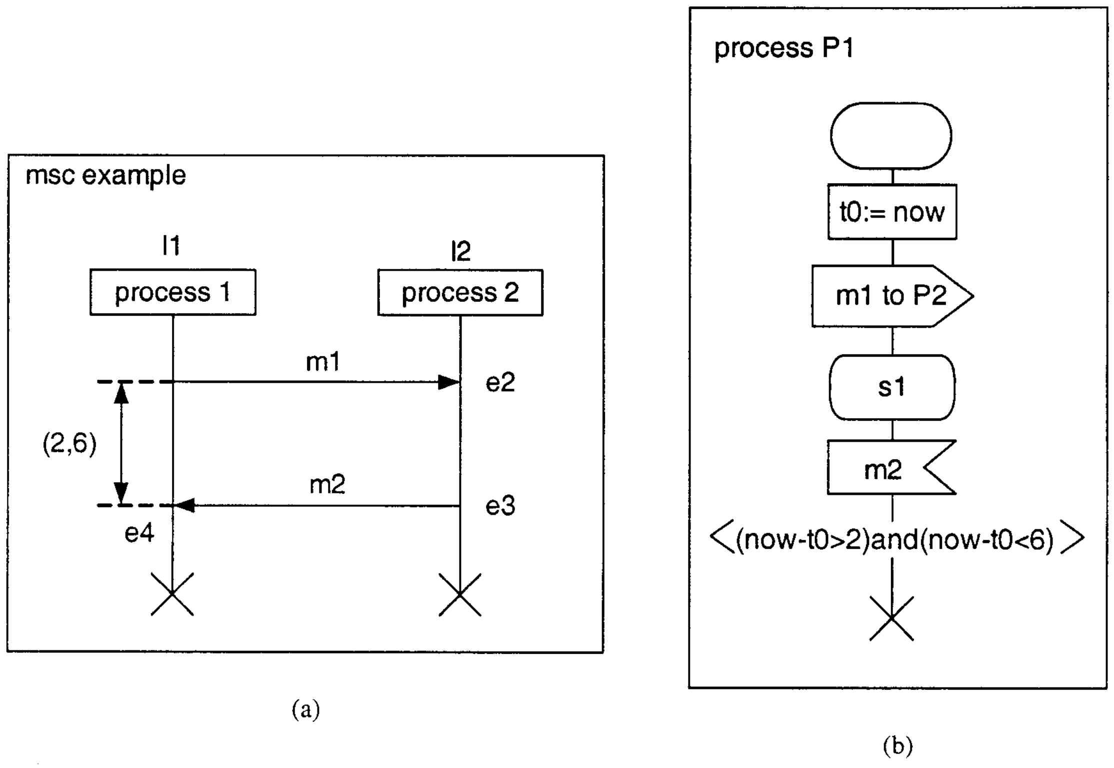


Figure 5.13: An MSC with relative time constraint within one instance and process behavior in SDL

5.4.2.5 Relative time measurements between two events in different instances

When a relative time measurement specified for a pair of events on two different MSC instances, the occurrence time of the first event is local to the process in which the Carryoutevent resides, and the process in which the Backevent resides needs it to calculate the time duration. Therefore, a signal extension must be resolved to convey the time value. The Variablemap contains all its attributes. In SDL generation, at the Frontevent, a task construct is generated, which binds the occurrence time of the Frontevent **now** into the FrontVariable. The Carryoutevent and the Carryinevent correspond to the sending and receiving event of the extended signal respectively. At Carryoutevent, the FrontVariable is inᴄ.:porated into the signal as one of its parameters sent to the other instance. Upon the receiving of this signal, which refers to the occurrence of the Carryinevent, for its SDL input construct, the CarryVariable is used to bind the time value carried by the FrontVariable. Finally on occurrence of the Backevent, an SDL task construct is generated to calculate the difference between current time **now** and the value of CarryVariable; the result is bound to the UserVariables.

In Figure 5.14(a), a time measurement is specified between events el and e4. Its corresponding Variablemap is shown as Table 5.9.

Front_ event	Back_ event	Carryout_ event	Carryin_ event	Front_ Variable	Carry_ Variable	User_ Variables	Stamp_ Variable
el	e4	el	e2	t0	tl	t2	NIL

Table 5.9: The Variablemap Table of MSC of Figure 5.14(a)

Figure 5.14(b) shows the process behavior of P1. We can see that at the Frontevent el, an SDL task construct is generated to record the occurrence time of the Frontevent **now** into the FrontVariable t0. The extended signal is ml. The Carryoutevent sends ml with the FrontVariable to Process P2. Figure 5.14(c) shows the process behavior of P2. The Carryinevent receives ml and binds the value of the FrontVariable to the CarryVariable tl. Then at Backevent, an SDL task construct is generated to calculate the difference between the global time **now** and the time value of the CarryVariable; the result is bound to the UserVariables t2. For simplicity of the illustration, we assume signal m1 and m2 travel the same channel in the SDL architecture, therefore, in Process P2, there is no save construct needed for signal m2 on consumption of m1.

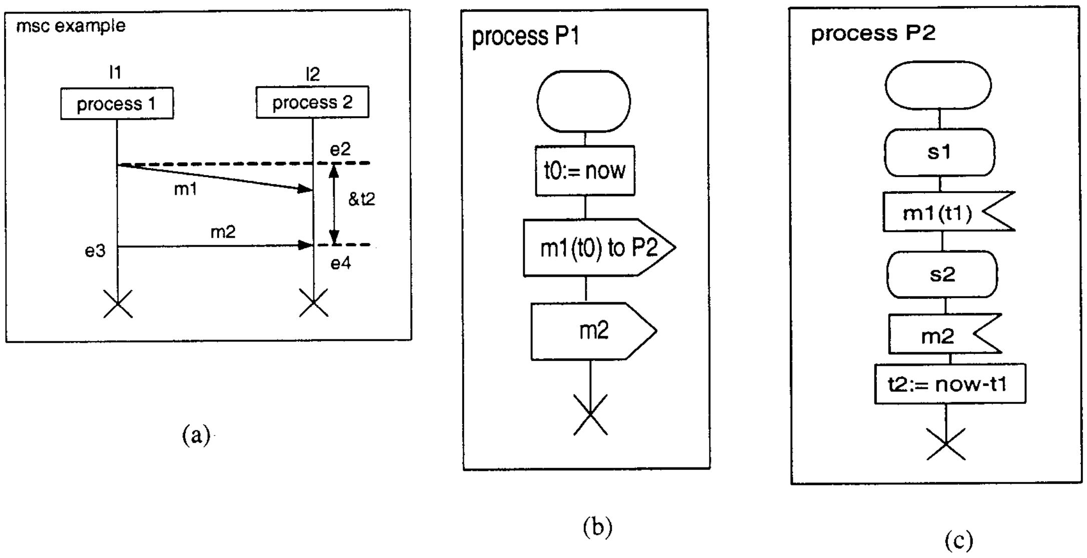


Figure 5.14: An MSC with relative time measurement between two events in different instances and the process behavior in SDL

5.4.2.6 Relative time constraints between two events in different instances

Similarly, when a relative time constraint is specified for a pair of events on two different MSC instances, the occurrence time of the first event is local to the process in which the Carryoutevent resides. The process in which the Backevent resides needs it to enforce the time constraint. Therefore, a signal extension must be resolved to convey the time value. The Variablemap contains all its attributes. In SDL generation, at the Frontevent, a task construct is generated, which binds the occurrence time of the Frontevent **now** into the FrontVariable. The Carryoutevent and the Carryinevent correspond to the sending and receiving event of the extended signal respectively. At Carryoutevent, the FrontVariable is incorporated into the signal as one of its parameters sent to the other instance. Upon the receiving this signal, which refers to the occurrence of the Carryinevent, for its SDL input construct, the CarryVariable is used to bind the time value carried by the FrontVariable. Finally on occurrence of the Backevent, an SDL continuous signals or enabling condition construct is generated to enforce time constraint with the condition checking whether the difference between the value of CarryVariable and the current global time **now** is within the specified range. The time constraint range can be concrete time value or using user defined variables.

In Figure 5.15(a), a time constraint is specified between events e1 and e4 using concrete time values. Its corresponding Variablemap is as Table 5.10.

Front_ event	Back_ event	Carryout_ event	Carryin_ event	Front_ Variable	Carry_ Variable	User_ Variables	Stamp_ Variable
e1	e4	e1	e2	t0	t1	NIL	NIL

Table 5.10: The Variablemap Table of MSC of Figure 5.15(a)

Figure 5.15(b) shows the process behavior of P1. We can see that at the Frontevent e1, an SDL task construct is generated to record the occurrence time of the Frontevent **now** into the FrontVariable t0. The extended signal is m1. The Carryoutevent sends m1 with the FrontVariable to Process P2. Figure 5.15(c) shows the process behavior of P2. The

Carryinevent receives m1 and binds the value of the FrontVariable to the CarryVariable t1. Then at Backevent, we generate an SDL enabling condition checking whether the difference between the value of CarryVariable and the current global time **now** is within the specified range (3, 6). For simplicity of the illustration, we assume signal m1 and m2 travel the same channel in the SDL architecture, therefore, in Process P2, there is no save construct needed for signal m2 on consumption of m1.

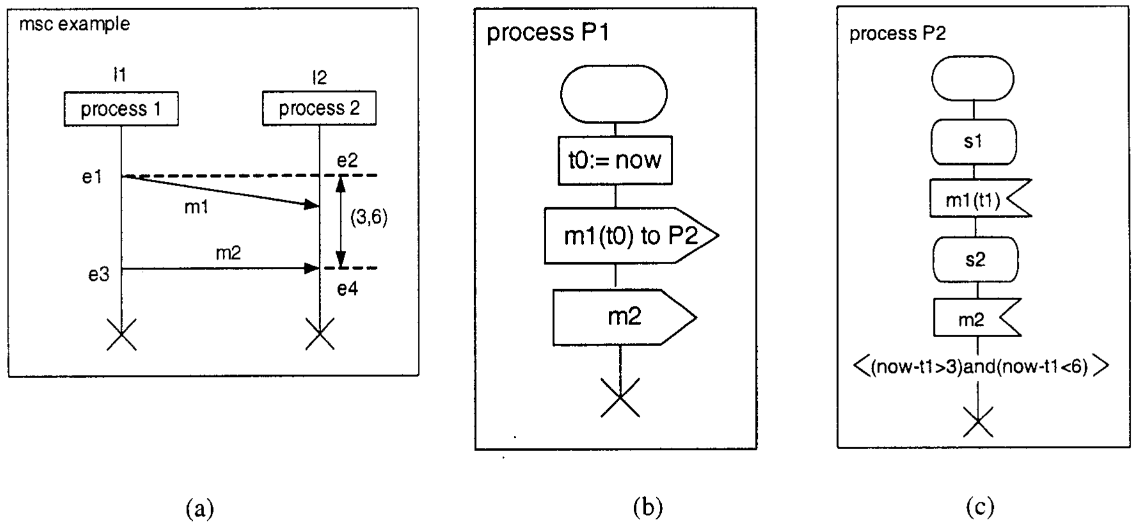


Figure 5.15: An MSC with relative time constraint between two events in different instances and the process behavior in SDL

5.4.3 Time requirements in coregions

As we have mentioned in the previous section, time requirements can be specified for events in a coregion. Since a coregion relaxes the order among events, therefore, there are usually more than one possible execution traces for one coregion. For each trace, time constraints and measurements must be translated into process behavior in SDL.

We can classify time requirements in coregions into three categories.

5.4.3.1 Absolute time constraints in a coregion

If there is any absolute time constraint or measurement specified in a coregion, in each trace it is handled the same way as described in the previous section.

5.4.3.2 Relative time constraints or measurements between one event inside a coregion and one event outside the coregion

For a relative time constraint or measurement between one event inside the coregion and anther event outside the coregion, it is still handled same way except that in each trace, the time constraint or measurement must be enforced for the two events. In the case that the relative time constraint is specified for the two events that are not in the same instance, a signal extension must be resolved before the translation phase.

For example, in the MSC specification shown in Figure 5.16, event e1 and e2 are in a coregion. A relative time constraint is specified between event e1 and e2, and an absolute time measurement is specified for event e4. The Enhanced Event Order Table is shown in Table 5.11.

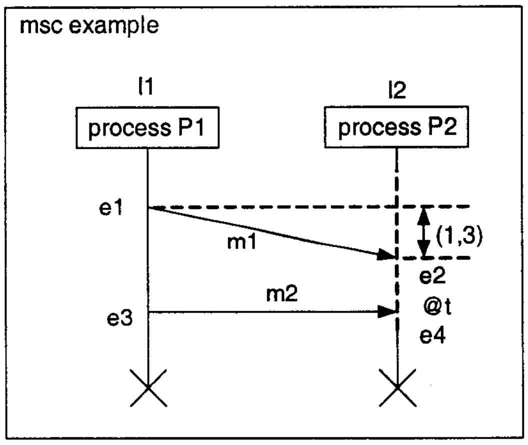


Figure 5.16: A timed MSC with coregion

	e1	e2	e3	e4
e1		T&(1,3)	T	T
e2				T
e3		T		T
e4		T		[illegible]

Table 5.11: The Enhanced Event Order Table of MSC of Figure 5.16

The coregion introduces two possible execution traces for process P2, which are e2 then e4 and e4 then e2. For both traces, m1 can be found as the extended signal for conveying a time value for translating the relative time constraint. Moreover, the absolute time measurement for event e4 is handled in each trace. The Variablemap is shown as Table 5.12.

Front_ event	Back_ event	Carryout_ event	Carryin_ event	Front_ Variable	Carry_ Variable	User_ Variables	Stamp_ Variable
e1	e2	e1	e2	t0	t1	NIL	t2
e4	NIL	NIL	NIL	NIL	NIL	t	NIL

Table 5.12: The Variablemap Table of MSC of Figure 5.16(a)

The Generated process behaviors in SDL are shown in Figure 5.17. For the relative time constraint between event e1 and e2, at the Carryoutevent, m1 is sent with the value of the occurrence time of the Frontevent. In each possible trace of process P2, at the Carryinevent, m1 is received and the value of FrontVariable is bound to the CarryVariable. Moreover, since the Carryinevent is the same as the Backevent, the occurrence time of the Backevent is recorded in the StampVariable t2. Because SDL does not allow values carried by a signal to be used in its enabling condition, we generate another state s2 or s5 to enforce the time constraint with the StampVariable and the CarryVariable. Furthermore, the absolute time measurement is handled in both possible traces.

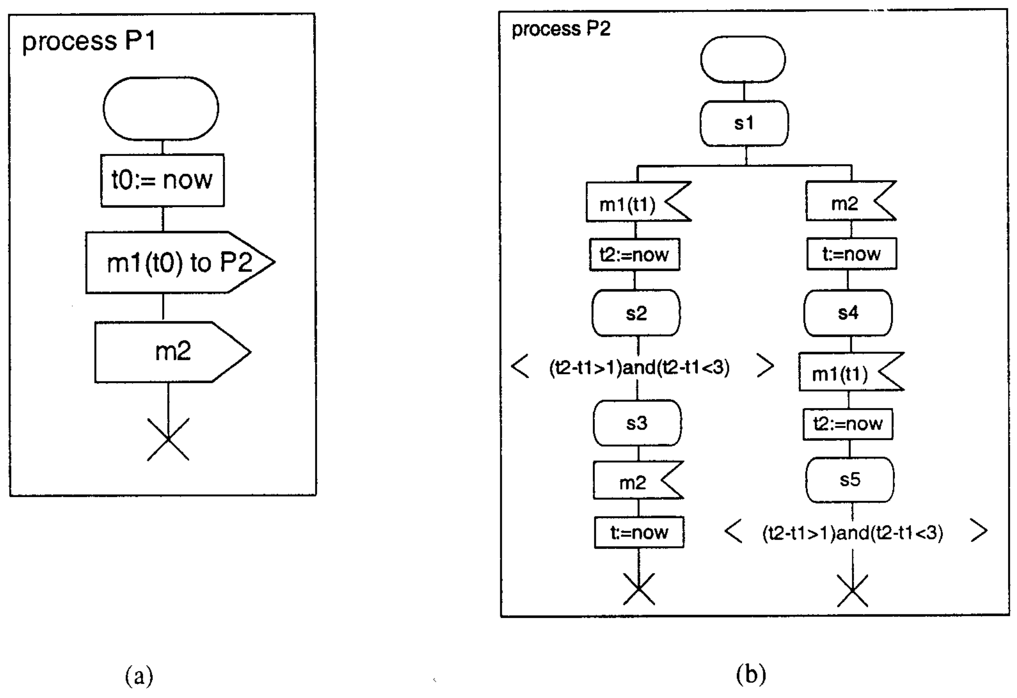
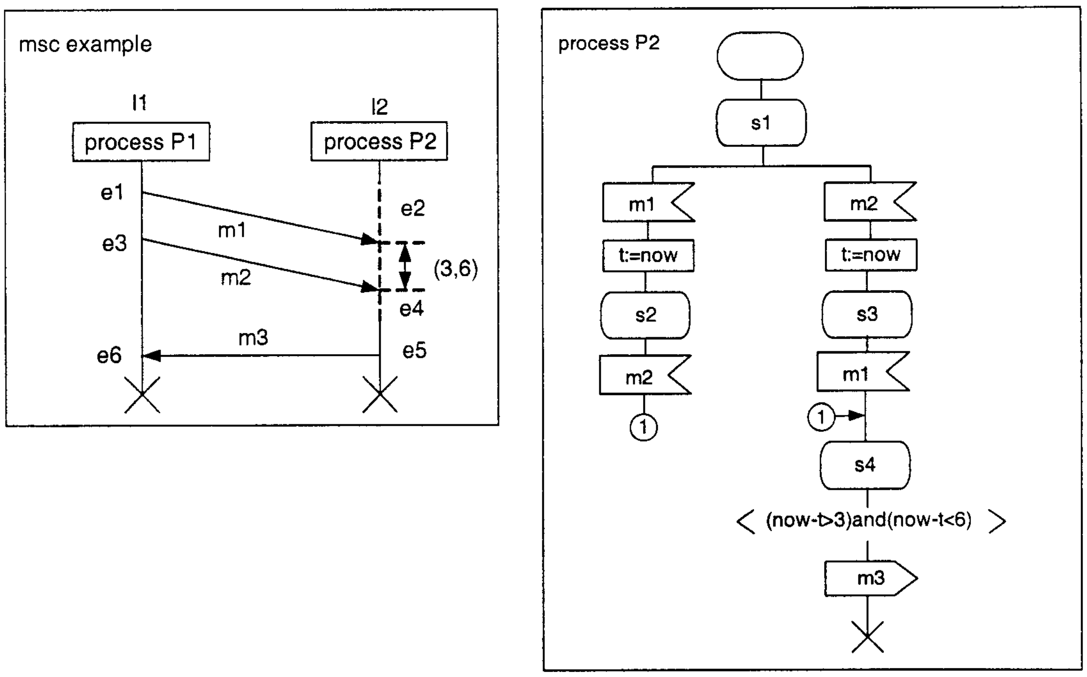


Figure 5.17: Generated process behavior in SDL for MSC of figure 5.16

5.4.3.3 Relative time constraints or measurements between two events inside a coregion

The third case is that a time constraint or measurement is associated with two events inside a coregion. Since the relative order of the two events can be different in different execution traces, then two Variablemap is generated for each possible trace. In SDL generation, each trace uses a Variablemap according to the relative order of the two events, ensuring the correct Frontevent and Backevent, etc.

Figure 5.18(a) shows a scenario that a time constraint is specified between events e2 and e4. The two Variablemaps are generated and shown in Table 5.13. One of the Variablemaps is for e2 precedes e4, and the other is for e4 precedes e2. In both Variablemaps, FrontVariables are both t since there is only one trace can be executed at a time. Similarly, if the time requirement is a relative time measurement, UserVariable is same too for both Variablemaps. In SDL behavior, in each trace, the global time value **now** is recorded into the FrontVariable at the Frontevent, and at the Backevent, time constraint is enforced. The generated process behavior for P2 is shown in Figure 5.18(b).



(a) (b)

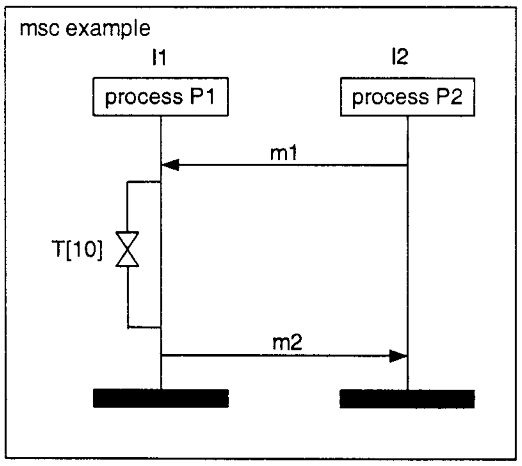
Figure 5.18: A timed MSC with coregion and process behavior in SDL

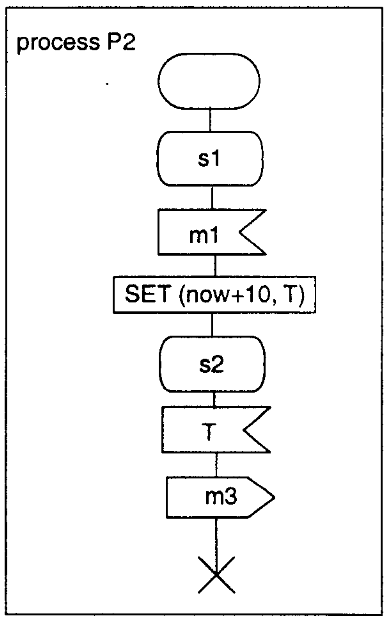
Front_ event	Back_ event	Carryout_ event	Carryin_ event	Front_ Variable	Carry_ Variable	User_ Variables	Stamp_ Variable
e2	e4	NIL	NIL	t	NIL	NIL	NIL
e4	e2	NIL	NIL	t	NIL	NIL	NIL

Table 5.13: The Variablemap Table of MSC of Figure 5.18(a)

5.4.4 Timer events

Time constraints can be specified in form of timer events. Semantically MSC starttimer and stoptimer events are equivalent to SDL set timer and reset timer constructs respectively. When a timer expires in SDL, a timer signal is generated and inserted into the owning process's input queue. Therefore, timeout event in MSC has its equivalence in SDL. Timer events and possible problems when they are in a coregion have been discussed in our existing approach [7]. The translation is straightforward. An example is shown in Figure 5.19.



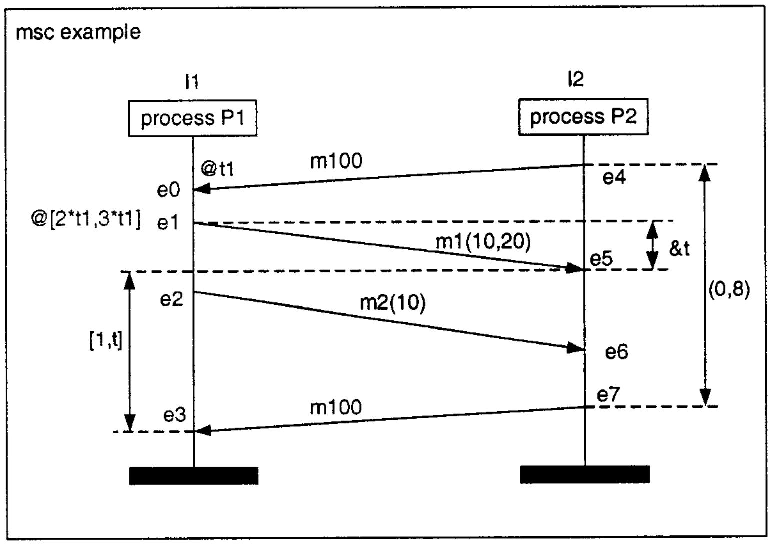


(a) (b)

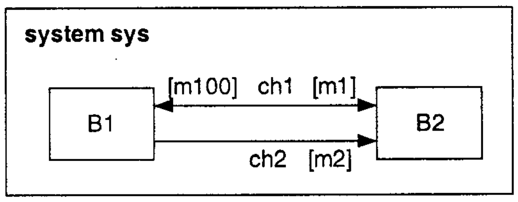
Figure 5.19: An MSC with timer events and the process behavior in SDL

5.4.5 A complete example of translation from a timed bMSC to SDL

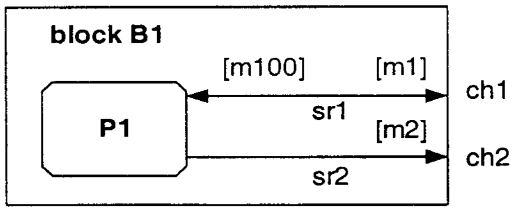
A timed bMSC specification and an SDL architecture are given in Figure 5.20. We present the major steps to illustrate the translation process.

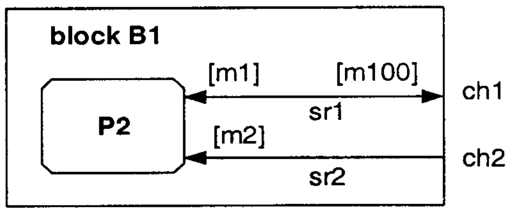


(a)



(b)





(c) (d)

Figure 5.20: A timed bMSC example with SDL architecture

First of all, we build the Enhanced Event Order Table containing the order relation between each pair of events and all the time requirements shown as Table 5.14. For example, the absolute time measurement for event e0 is recorded in table cell (e0, e0). The result of this measurement is used to constrain event e1; the information is also filled in cell (e1, e1). The relative time measurement between event e1 and e5 is recorded in cell (e1, e5). The measurement result is used to constrain event e5 and e3, etc.

	e0	e1	e2	e3	e4	e5	e6	e7
e0	@t1	T	T	T		T	T	T
e1		@[2*t1,3*t1]	T	T		T &t	T	T
e2				T			T	T
e3								
e4	T	T	T	T		T	T	T&(0,8)
e5				T&[1,t]			T	T
e6				T				T
e7				T				

Table 5.14: Enhanced Event Order Table of MSC of Figure 5.20(a)

Since both of them have input events, we build Occupancy Tables for process P1 and P2 shown as Table 5.15. For P1, both event e0 and e3 are inputs of signal m100, and there are no other signals can be in the input queue. For P2, event e5 is the input of signal m1, and signal m2 may be in the input queue when P2 consumes m1.

input events	input message	Channel sr1
e0	m100	m100
e3	m100	m100

input events	input message	Channel sr1	Channel sr2
e5	m1	m1	m2
e6	m2	m2	

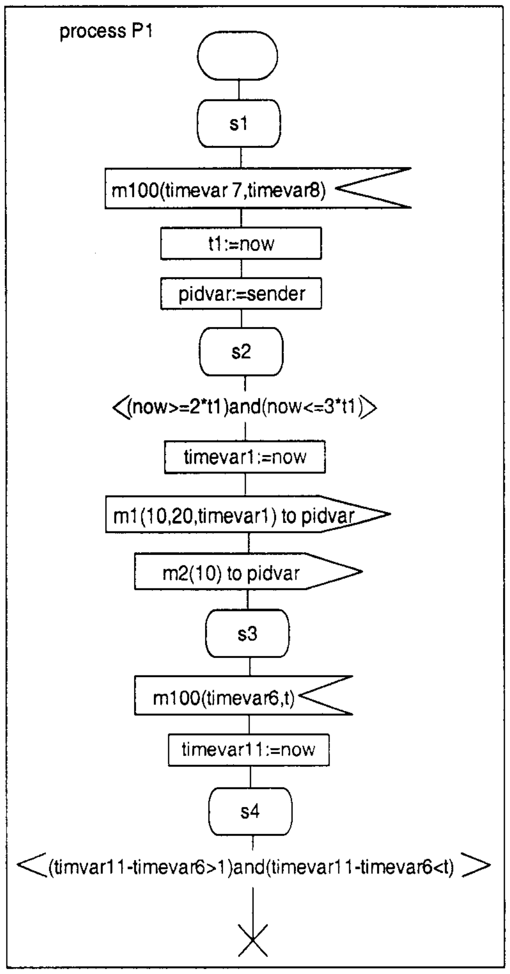
Table 5.15: Occupancy Tables of the example of Figure 5.20

The next step is to build Variablemap table (shown as Table 5.16), which contains the necessary information to translate time requirements into SDL. In the mean time, signal extensions are resolved as m1 (integer, integer, time) and m100 (time, time). For example, both the absolute time measurement and constraint for event e0 and e1 has a Variablemap respectively, in which only UserVariable is used. For relative time measurement between event e1 and e5 (which belong to two different instances), the Variablemap indicates the corresponding Carryoutevent, Carryinevent, as well as the necessary time variables. Time constraints between event e4 and e7 is local to instance I2, therefore, Carryoutevent, Carryinevent and CarryVariable are not used. A StampVariable is used for relative time constraint between e5 and e3, since event e3 is both Backevent and Carryinevent.

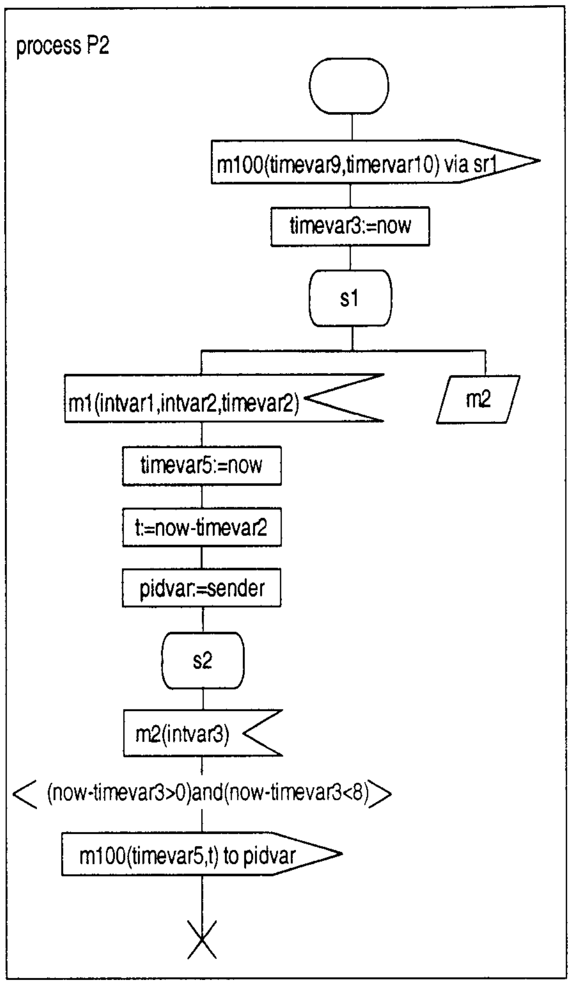
Front_ event	Back_ event	Carryout_ event	Carryin_ event	Front_ Variable	Carry_ Variable	User_ Variables	Stamp_ Variable
e0	NIL	NIL	NIL	NIL	NIL	t1	NIL
e1	NIL	NIL	NIL	NIL	NIL	t1	NIL
e1	e5	e1	e5	timevar1	timevar2	t	NIL
e4	e7	NIL	NIL	timevar3	NIL	NIL	NIL
e5	e3	e7	e3	timevar5	timevar6	t	timevar11

Table 5.16: Variablemap Table of the MSC of Figure 5.20(a)

Finally, we generate the process behaviors in SDL using the mapping techniques we have described in the previous sections. All the time requirements in MSC specification are translated into SDL as shown in Figure 5.21. For example, in the generated process behaviors for P1, absolute time measurement for the input of signal m100 is done with an SDL task that binds the value of global time **now** into the Uservariable t1. The absolute time constraint for sending signal m1 is enforced with a continuous signal in state S2. Since the output event of m1 is both the Frontevent and the Carryoutevent for relative time constraint between event e1 and e5, signal m1 is extended to send the occurrence time of event e1, which has been recorded in timevar1. The following input event of m100 is both the Carryinevent and the Backevent of relative time constraint between event e5 and e3. Therefore, a StampVariable is used here to record the occurrence time of e5. Moreover, since m100 is extended with two time parameters, the input of signal m100 in state s1 has two dummy parameters timevar7 and timevar8. This is solely for the syntax purpose.



(a)


process P2
m100(timevar9,timervar10) via sr1
timevar3:=now
s1
m1(intvar1,intvar2,timevar2)
m2
timevar5:=now
t:=now-timevar2
pidvar:=sender
s2
m2(intvar3)
(now-timevar3>0)and(now-timevar3<8)
m100(timevar5,t) to pidvar

(b)

Figure 5.21: Generated process behavior in SDL for the example of Figure 5.20

5.4.6 Inline expressions with time

As we have already mentioned in the section of the Enhanced Event Order Table, time requirements can be associated with inline expressions. The key to handling them is to determine the involved starting/ending events of each operand in these inline expressions by checking the order information among events in the Enhanced Event Order Table. Then we can apply the techniques we have discussed in the previous sections to translate the specified time requirements into process behaviors in SDL.

However, besides showing how SDL process behaviors can be generated from timed inline expressions in the previous sections, we also need to study how inline expressions can be mapped to SDL constructs. Since inline expressions are structures in MSC, they can not be mapped to SDL constructs on a one to one basis. Because SDL is strong in its expressiveness, there may be more than one SDL behavior equivalent to an MSC inline expression. For the purpose of translation, we want to regulate SDL designs for different types of MSC inline expressions.

Alt

Each operand of an alt inline expression becomes a branch of the process in SDL. All the branches start with a common state that we generate for the beginning of the alt inline expression. If any operand has guarding condition, we generate a continuous signal with the condition. Finally, all branches join with an SDL label at the end of the inline expression. Figure 5.22 shows an example of the design. For illustration purpose, we assume message m3 and m4 travel the same signal channel.

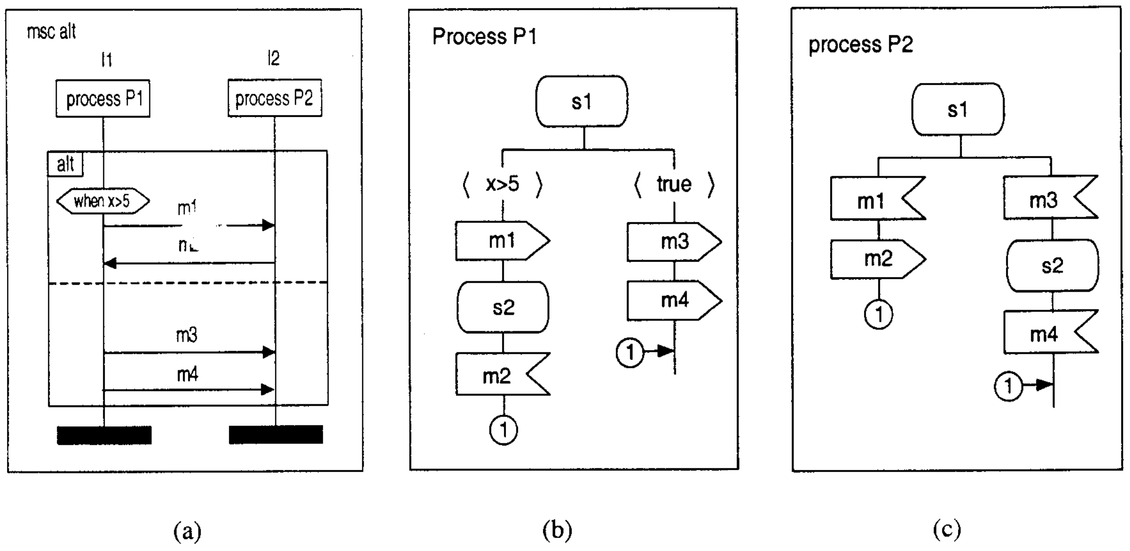
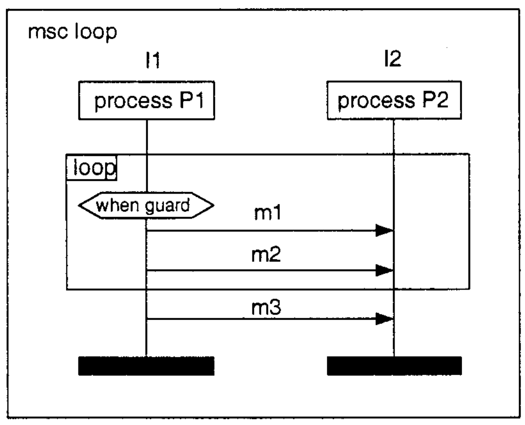
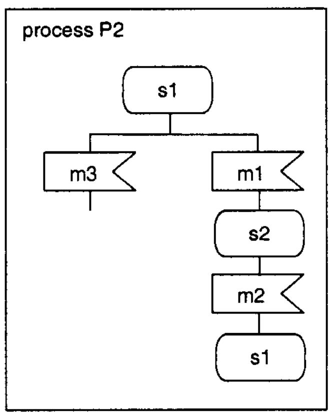


Figure 5.22: A typical alt inline expression in MSC and process behavior in SDL

Loop

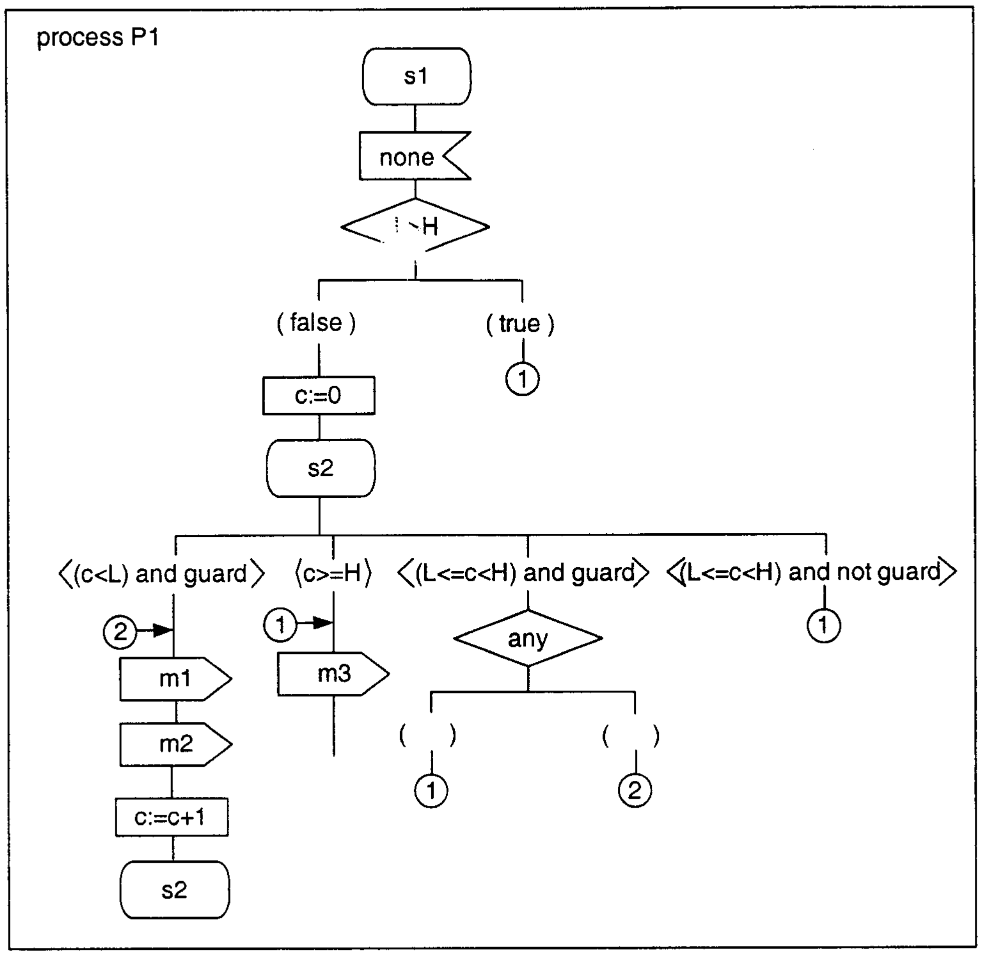
Loop expressions are rather complex due to the loop boundaries. First of all, we generate an SDL state followed by input of **none** and a decision of testing whether the lower bound is greater than the upper bound. If so, the execution of the loop frame is skipped. Otherwise, an SDL task construct is generated to initialize an loop counter to zero; it then is followed by another state as a decision point. In the following, four continuous signals are generated with different combinations of the guarding condition and the number of times of iterations executed. If the loop counter is less than the lower bound and the guard condition is true, the loop frame is executed once and the loop counter increments by one. Then the second state is used to take the control flow back to the decision point. If the value of the loop counter is in between the lower bound and upper bound and the guard condition is true, the instance has nondeterministic decision on continuing to execute the loop frame or skipping it. If the loop counter is in between the lower bound and upper bound and the guard condition is false, the instance skips executing the loop frame. Finally if the value of the loop counter is greater than the upper bound, the execution of the loop frame is skipped. We use SDL labels to join different branches, and a state to achieve the iterations. Note that this process behavior can be only for the instance which makes the decision about on executing the loop. Any other participating instance behavior can be much simplified. Figure 5.23 shows an example of the design. For illustration purpose, we assume message m1, m2 and m3 travel the same signal channel. In the example, guard is a boolean expression; L and H are the upper and the lower bounds of the loop; and c is the loop counter.





(a) (b)

Figure 5.23: A typical loop inline expression in MSC and process behavior in SDL

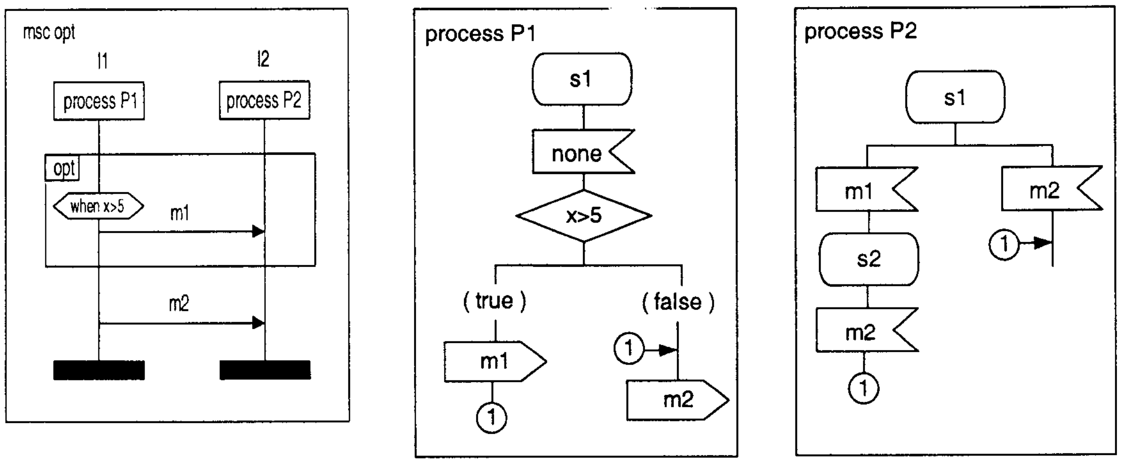


(c)

Figure5.23: A typical loop inline expression in MSC and process behavior in SDL (continued)

Opt

We generate an SDL state for the beginning of an opt expression. The semantics of MSC requires that a guarded opt expression always goes through the option operand if the guard is **true**. Therefore, if a guarding condition is specified, we generate an SDL input construct of **none** followed by a decision construct with the guard as its condition. On the true answer side, we translate all events in the opt frame; on the false answer side, we translate all events after the opt frame. Finally, the two braches join with an SDL label. If there is no guarding condition for the opt inline, expression, and SDL decision construct becomes a nondeterministic decision. Figure 5.24 shows an example of the design. For illustration purpose, we assume message m1 and m2 travel the same signal channel.



(a) (b) (c)

Figure 5.24: A typical opt inline expression in MSC and process behavior in SDL

5.4.7 Time constraints and measurements specified with other orderable events

One assumption we have held through all previous discussion is that we consider only time requirements specified with message input and output events in MSC specifications. However, time constraints and measurement can be also associated to action events, creation events, timer events etc. Since these events currently are not in the Enhanced Event Order Table, to handle such timed events, we can simply include them into Event Order Table. We skip the detailed discussion here, since the modification is straightforward.

5.5 Problems encountered during translation to SDL

We have discussed the approach of translating timed bMSC specification into an SDL specification. However, not all timed MSC specifications can be implemented in SDL. There are several causes we will discuss in the following sections. Similar to handling the consistency checking between the MSC and SDL architectures that we have discussed in the previous chapter, our approach will be checking the scenarios which cause the implementability problems before the actual SDL generation starts. If any of the problems found, we report it and the translation terminates.

5.5.1 Time related implementability

Time requirements may introduce some implementability problems. We can categorize them as follows: Some MSC specification with time constraints is not implementable with SDL due to conflicts among time requirements or under specification. In this case, information needed to check these problems can be found in the Enhanced Event Order Table. Some other implementability problems may also come from the incompleteness of the semantics of MSC standard or problematic MSC scenarios such as non-local choice.

5.5.1.1 Time consistency

Time consistency means the consistency among different time constraints in an MSC specification. Time constraints specify temporal orders between events. These orders have to be consistent with the causal orders between events [6]. However, this is not always the case in MSC specifications if time constraints are not specified carefully. There are two cases we describe in the following:

(1) Absolute time constraints violating the causal order.

(2) Absolute time constraints violating relative time constraints

(3) Relative time constraints violating time interval to one another.

The theoretical definition of time consistency and the checking algorithm are discussed in [8] and [6] respectively. Here we just explain above scenarios by several examples.

For example, in figure 5.25, event e2 and e4 are ordered, but their absolute time constraints conflict to each other in terms of the time range. Event e2 precedes e4, therefore, e2 cannot occur in absolute time (4, 6) while e4 occurs in (1, 3). Moreover, the relative time constraint between event e1 and e4 also conflict with the absolute time constraints on e1 and e4. In figure 5.26, two types of time inconsistency occur. First, on the same instance I1, the relative time constraint between e1 and e4 contradicts the relative time constraint between e1 and e5. If e5 occurs in the interval of (1, 3) after e1, then e4 can not occur in the interval of (5, 7) after e1, because e4 precedes e5. Second, in instances I2, the relative time constraint between e2 and e3 contradicts the relative time constraint between e1 and e4, because e2 and e3 succeed e1, precedes e4, and if events e2 and e3 take an interval of (8, 10), then the interval of (5, 7) between e1 and e4 can never be satisfied. The MSC is syntactically correct, but semantically incorrect. Before doing any translation, the time consistency checking has to be done. This step is critical for handling timed MSC to generate SDL.

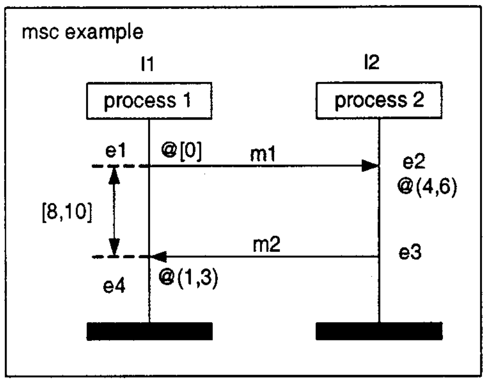


Figure 5.25: A timed MSC with absolute time constraint conflicts

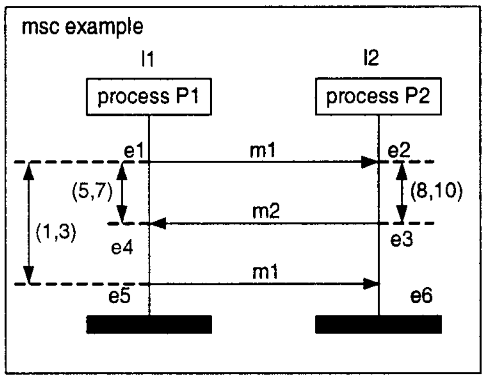


Figure 5.26: A timed MSC with relative time constraint conflicts

5.5.1.2 Time-order related under-specifications

A relative time constraint is specified between two events; however, their temporal order may be not implied by any causal order rule. MSC specifications with this kind of scenarios are not implementable with SDL.

For example, in figure 5.27, events e1 and e2 are not ordered. However, a relative time constraint is specified between them. This scenario is not implementable in SDL. Note the MSC is not an erroneous specification. It can be seen as an under-specification case and the order between e1 and e2 may be refined in future development of the specification. In the enhanced event order table, the scenario will cause the precedence in two table cells associated with e1 and e2 unmarked, and this tells us to terminate the translating procedure.

Time-order related under-specification checking rule: for any cell in the Event Order Table, if a relative time constraint is specified for the two associate events and precedence is unmarked, then report under-specification for the MSC and terminate translation.

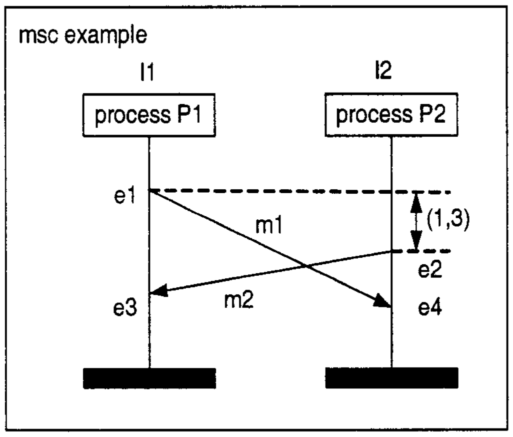


Figure 5.27: An unimplementable MSC with a time-order related under-specification

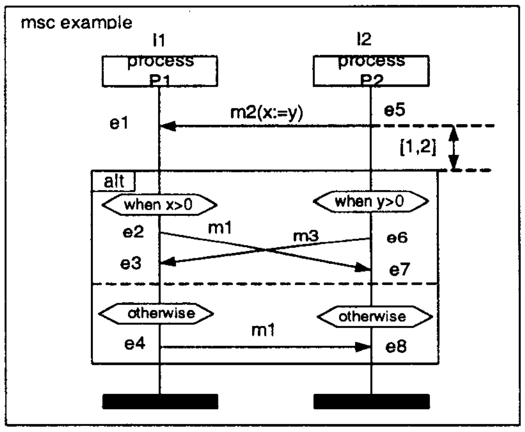


Figure 5.28: An unimplementable MSC with a time-inline expression related under-specification

5.5.1.3 Time-inline expression related under-specification

MSC allows time requirements specified between for inline expressions. As discussed in the previous sections, we handle this kind of time requirement by determining the starting and/or ending events in each operand through order information in the Enhanced Event Order Table. However, this is not always possible. There are scenarios that the starting and/or ending events in some operands are not decidable. MSC specifications with those scenarios can not be implemented.

For instance, in figure 5.28, a relative time constraint is specified between event e5 and the starting events of the alt inline expression. However, in the first operand, it is impossible to decide whether e2 or e6 is the starting event. Therefore, we do not know which event should be imposed with the time constraint. During the process of building the enhanced event order table, if this scenario happens in MSC specification, the algorithm detects it and terminates the translation.

Time-inline expression related under-specification rule: for any inline expression specified with time constraints, use order information in the Enhanced Event Order Table to determine the starting or ending event for each operand, if the event in interest cannot be determined, then report under-specification for the MSC specification and terminate the translation.

5.5.1.4 Time constraints in Loops

Time requirements in loop inline expressions are rather complex. Most of the problems are caused by the interaction among the loop guarding condition, loop boundaries, and time constraints on events inside a loop. This interaction affects the number of iterations that a loop executes. Sometimes an inconsistency occurs when a loop can not execute even the number of iterations specified by its lower bound, and this is because the guarding condition or absolute time constraints on events inside the loop prevent it from more iteration. [6] has addressed the time consistency problem in loops.

Moreover, an event inside a loop is actually a serial of events in execution. When an absolute time requirement is specified on such an event, the MSC standard does not say whether this time constraint need to be respected all iterations or just in the first one. Figure 5.29 shows an example. The loop should iterate 5 times, event e2 is within a loop and with an absolute time constraint. Respecting the absolute time constraint in all interactions definitely is a stricter constraint on the system. Similarly, the MSC standard is not clear on relative time requirements in a loop. Does the time constraints needs to be respected between e2 and e4 only, or also between e4 and e2' (where e2' is e2 in the next iteration).

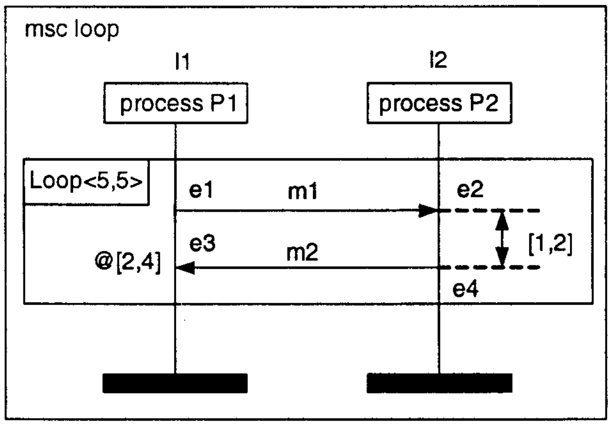
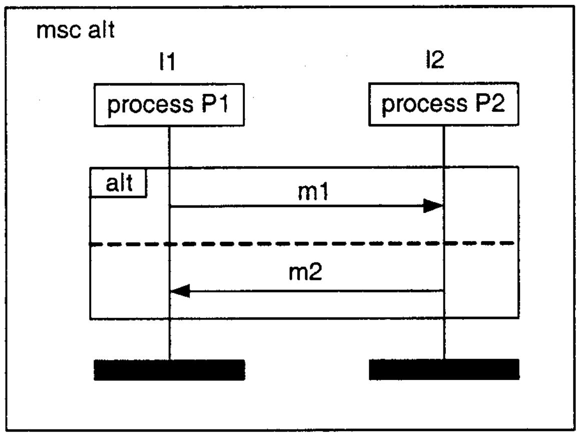


Figure 5.29: A timed MSC with loop

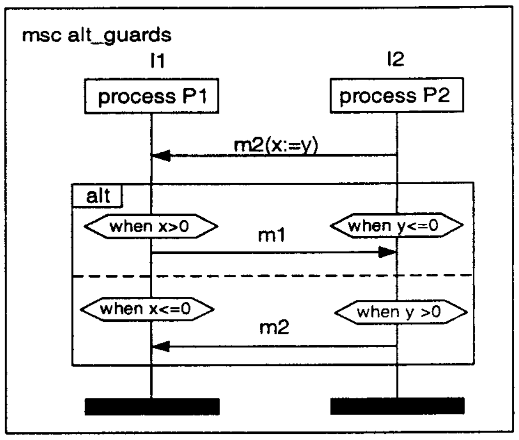
At the current stage, we adopt the assumption that absolute time requirements need to be respected all iterations and relative time requirements need to be respected between events in the same iteration of the loops.

5.5.2 Non-local choice

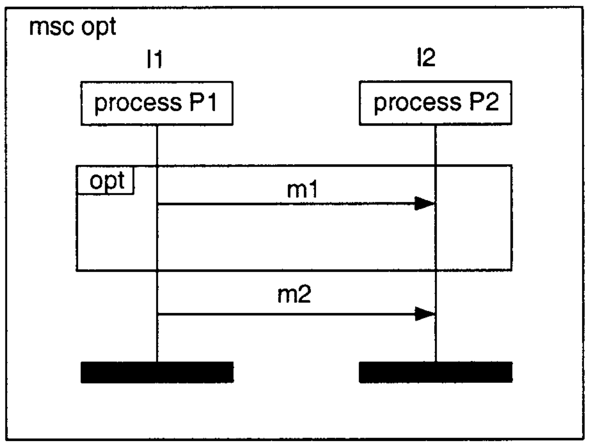
The previous work has addressed non-local choice problem in HMSCs that makes them unimplementable. Similarly, inline expressions in bMSCs can also cause non-local choice, when two or more instances may choose to execute different operands. The bMSCs with non-local choice are not implementable. For example, Figure 5.30(a) shows a simple alt inline expression, where instance I1 may chose to execute sending m1 to I2, while I2 may choose to execute sending m2 to I1. The decisions are made purely based on individual instances. Moreover, a non-local choice can also be caused by guarding conditions for inline expression operands. Figure 5.30(b) shows this kind of scenario. The value of variable x and y are the same through a binding in passing message of m2. The value of x and y are used in the guarding conditions in the alt inline expressions. Apparently, instance I1 and I2 can not have their conditions evaluated to the same boolean value for any operand. So they can not choose the same operand to execute by MSC semantics. There is another type of non-local choice can be caused by the system architecture. Figure 5.30(c) shows an opt inline expression, in which instance I1 decides whether the opt operand is executed or not, while instance I2 simply follow I1 by getting a different message. However, if message m1 and m2 travel through different signal channels, a scenario might occur in which instance I1 decides to execute opt operand and sends m1 and then sends m2, but m1 gets delayed in the channel and I2 sees message me first. In this case, I2 has no idea about the decision on executing opt operand, which has been made by I1.



(a)



(b)



(c)

Figure 5.30: MSCs with a non-local choice

We can generalize the causes of non-local choice through the examples.

(1) For the first example, there is no instance which can solely decide the operand to execute, or we say that there is controller instance for the inline expression.

(2) For the second example, the guard conditions yield conflict operand choice among the participating instances.

(3) For the third example, a decision can be solely made by an instance. However, it might be mistaken by some participating instances due to the system architecture.

The following is a further analysis from the three generalized aspects and a devised non-local choice detecting algorithm.

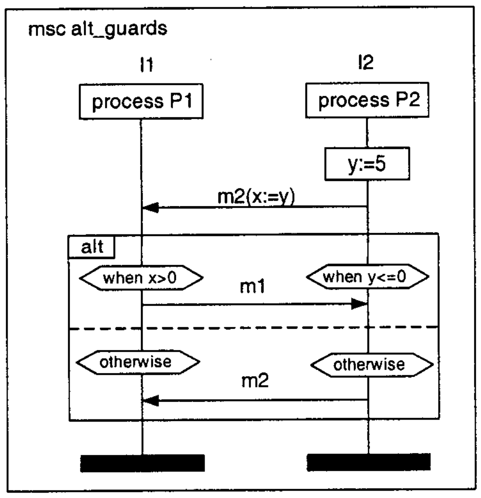
First of all, since the non-local choice in inline expressions are due to the possibility of participating instances executing different operands, for non-local choice detecting purpose, we need to define the operands for each type of inline expression. For an alt inline expression, the operands are same as defined in MSC standard, because each participating instance has to execute one of them. For an exc, an opt or a loop inline expression, the first operand is the enclosing scenario in the inline expression, and the other is the rest of the enclosing frame, because the decision is about executing the first operand or not every time when the control flow reaches the beginning of the inline expression.

Our detecting approach starts with "Mark True operands", which intends to find the possible operands to execute for each participating instance.

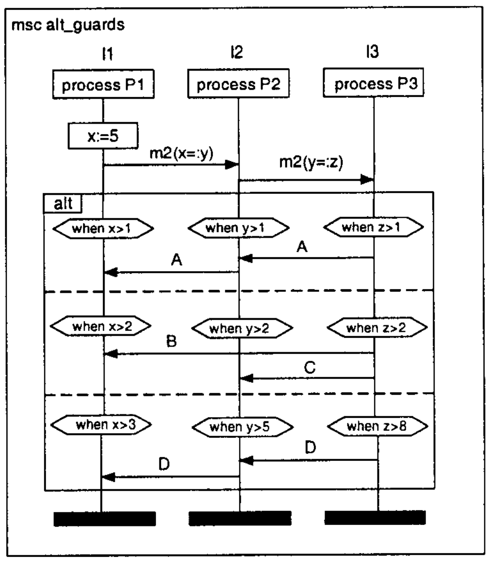
For each participating instance, evaluate the guarding conditions in each operand and mark the operand if the guard is evaluated **true.** If there is no guard for some operands, they are also marked (no guard implies a guarding condition **true**).

STEP 1

If for each instance, there is only one operand marked and this operand is common among all participating instances, non-local-choice does not appear. If these marked operands are not common to all participating instances, non-local choice exists. If any instance has more than one operand marked **true**, we need to do further analysis.



(a)



(b)

Figure 5.31: MSCs with alt inline expression

In figure 5.31(a), for both instance I1 and I2, there is only one operand that can be marked **true** (since x and y take the same value 5 through binding along message m2), and their operands marked **true** are common to each other, so that there is a common choice between the two instances. Non-local choice does not exist.

If we can not draw a conclusion from STEP 1, further analysis is needed. The next step of analysis is about whether there is an instance can solely decide the operand to be executed among the possible operands.

Definition: *if an instance I sends a signal S before any other instances in one operand of an inline expression, and any other participating instance receives a signal following S before all its other events, we say that instance I controls the operand of this inline expression, and S is called the control signal.*

If all operands of an inline expression are controlled by instance I, and any other shared instances receive different signals (or the same signal with different data values along the signal) following each control signal Si in each operand, we say that instance I controls the inline expression. If such an instance I does not exist, we say this inline expression is controlled by multiple instances.

STEP 2

If there exist some instances having more than one operand marked, form operand sets with these marked operands for all instances, say s_1, s_2 … s_n (without losing generality, we assume that size $(s_1)<=$ size $(s_2)<=\ldots<=$ size (s_n)). Non-local choice does not exist for alt and exc inline expressions, if these sets satisfy $s_1 \subseteq s_2$, $s_1 \subseteq s_3$ … $s_1 \subseteq s_n$., and the instance corresponding to s_1 is the controller instance of this inline expression. Otherwise, non-local-choice exists.

This condition basically says that if there is more than one possible operand for an instance to execute, then the instance with the fewest possible operand must be the controller instance and that the choice made this instance must be one of the possible operands of any other participating instance.

In figure 5.31(b), since variables x, y and z all takes the same value 5, for instance I1, all its three alt operands are marked **true**, and for instance I2 and I3, only the first two operands are marked **true**. Therefore, this satisfies the condition, $s_3 \subseteq s_2 \subseteq s_1$.
Moreover, instance I3 sends different control signals to other two instances in each operand of the inline expression (note: in the first operand, instance I3 sends A to instance I2 and instance I2 sends A to instance I1, this is still a case that I3 is the controller, since instance I1 and I2 has no event before they receive signal A as the result of sending signal A from instance I3), therefore, I3 is the controller, non-local choice does not exist.

If we modify the above example slightly, it gives us another scenario that shows non-local choice. In figure 5.32, since x, y and z all take the same value 5, for instance I1, all its three alt operands are marked **true**, for instance I2 the first and the third operands are marked **true** and for instance I3, the first and the second operands are marked **true**. That is, $s_3 \subseteq s_1$ is true; but, $s_3 \subseteq s_2$ is false. Non-local choice exists.

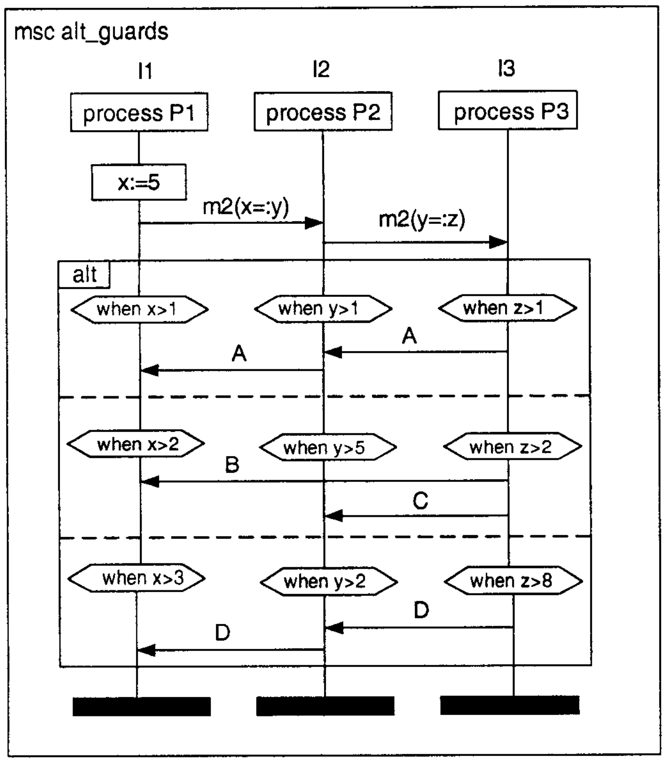


Figure 5.32: An MSC with non-local choice

For opt and loop inline expressions, STEP 2 must also be successful in order to apply STEP 3.

STEP 3

If control signals to any participating instance for the two operands travel different channels in the given SDL architecture, non-local choice exists. Otherwise, non-local choice does not exist.

We have this condition for loop and opt inline expressions because the controlling signal for the second operand can be received by a non-controller instance both when the controller decides to execute the opt or execute at least one iteration of the loop, and when the controller instance decides to execute the second operand directly (loop may be sipped). These two scenarios can only be distinguished by allowing the control signals to any participating instance travel the same signal channel.

We have already shown an example in Figure 5.30 (c) and explained the reason how non-local choice can occur because of the SDL architecture. However, if m1 and m2 traverse the same channel and if m1 has been sent, I2 will receive m1 before m2. I2 knows that opt is executed. If m2 is received without receive m1 before, I2 knows that opt should not be excused. The reason is that in the same channel, signals are FIFO in SDL.

5.6 SDL Generation from timed HMSC

HMSCs provide a means to geographically define how a set of MSCs can be combined. HMSCs also allow the use of time requirements. Therefore, we have also done study on translating timed HMSC specifications into SDL specifications. Our existing approach handles translation from HMSC to SDL. However, it adopted the syntax of HMSC from ObjectGeode, which is not the MSC-2000 standard specified in [5]. In the study of translating timed HMSC into SDL, we have taken the HMSC standard syntax. Consequently the algorithms of generating Enhanced Event Order Table, building Occupancy Table and the translating from HMSC to SDL have been redesigned. As the previous work, we do not handle parallel frames in HMSCs. In the following sections, we will describe our approach.

5.6.1 Translating HMSC into SDL

An HMSC specification is composed with nodes that can be a starting symbol, end symbols, MSC references, conditions, and connection points. The flow lines connect the nodes indicates the possible sequencing among the nodes [5]. In fact, we can view an HMSC specification as a directed graph, with all previous mentioned nodes and the flow lines as its directed edges. To generate process behaviors, we need to gather events for each process from all nodes in the HMSC into an SDL process with respect to the sequencing that the flow lines specify.

5.6.1.1 Event Order Table for HMSC

Like the approach for translating timed bMSC specifications to SDL specifications we still need to keep track of the ordering and timing information among events in the Enhanced Event Order Table when handling timed HMSCs. This information facilitate to the building of Occupancy Tables later on to avoid deadlocks and Variablemaps to handle time requirements in the translation phase. However, the order among all events from different MSC references in an HMSC specification cannot be built only according to the partial order rules described previously, since the flow lines of an HMSC also help to specify orders among events. In fact, we can build individual Enhanced Event Order Tables for each bMSC reference, and then obtain a global Enhanced Event Order Table by properly combining them according to the order relationship of the referenced bMSCs specified by the flow lines.

To find the order relationship among referenced bMSCs, we can traverse the HMSC graph and calculate paths between each pair of them. A table is constructed with its rows and columns representing the nodes in the HMSC. We fill table each of the table cells a found path weight from the row node to the column node. If there is no path from one node to another, the weight takes infinity (inf). This table can be seen as a Meta "Event Order Table", since it provides an order map at the level of bMSCs for calculating order among all events eventually.

For example, Figure 5.33 shows an HMSC with its referenced bMSCs, in which the HMSC nodes are labeled except the starting node as they appear in the textual form. All Enhanced Event Order Tables for the individual referenced bMSCs are built and shown as Table 5.27.

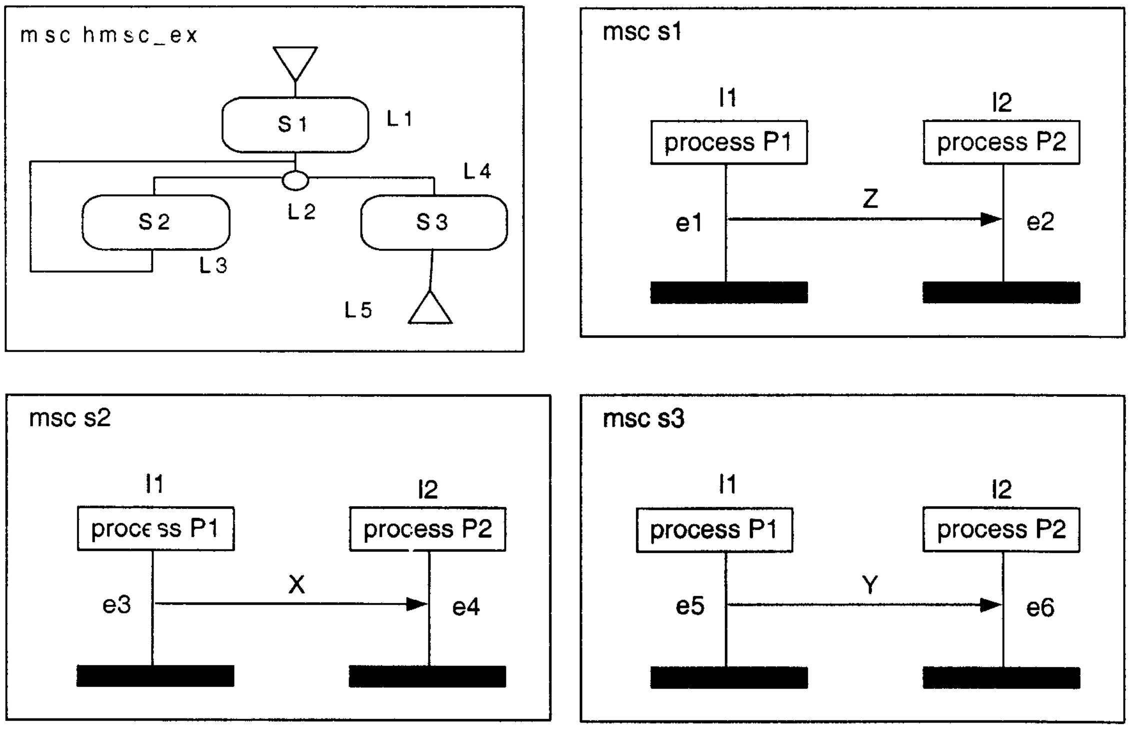


Figure 5.33: An HMSC specification

	e1	e2
e1		T
e2		

	e3	e4
e3		T
e4		

	e5	e6
e5		T
e6		

Table 5.17: Enhanced Event Order Tables of individual bMSCs of Figure 5.33

We calculate all pair shortest path for the HMSC graph and obtain the Table 5.18. The number in each table cell indicates the minimum number of edges from the row node to the column node. An inf indicates no path is available.

	L1	L2	L3	L4	L5
L1	0	1	2	2	3
L2	2	0	1	1	2
L3	1	2	0	3	4
L4	inf	inf	inf	0	1
L5	inf	inf	inf	inf	0

Table 5.18: All pair shortest path for the HMSC example of Figure 5.33

Now we can combine the individual Enhanced Event Order Tables into a global Table using the order information provided in Table 5.15. In fact, the events in different referenced bMSCs respect the weak sequencing order. Let W (L1, L2) denotes the shortest path weight of two different nodes from L1 to L2. Rules to interpret order relationships between each pair of nodes and their contained events are described as follows:

Precedence

If W (L1, L2) is not inf, we say that node L1 precedes L2. Therefore, if L1 and L2 contain a common instance, then all events in L1 on this instance precede all events in L2 on the same instance.

Exclusive

If W (L1, L2) and W (L2, L1) are both inf, we say that L1 and L2 are exclusive. Therefore, all events in L1 are exclusive to those in L2, and vice versa. This case is similar to the situation of events in different operands from alt inline expression. The corresponding cells are marked with "E".

We can obtain a global Table for all events in the HMSC shown as Table 5.19, which retains the order from individual Enhanced Event Order Tables with new orders information added according to the rules. For example, since W (L1, L3) is not inf, for instance, event e1 precedes e3. Similarly, e1 also precedes e5; e2 precedes e4 and e6; e3 precedes e1 and e5; e4 precedes e2 and e6. Finally, the table is updated by itself by calculating a transitive closure for each event. Table 5.20 shows the final global Enhanced Event Order Table.

	e1	e2	e3	e4	e5	e6
e1		T	T		T	
e2				T		T
e3				T	T	
e4						T
e5						T
e6						

Table 5.19: An intermediate Event Order Table of HMSC of Figure 5.33

	e1	e2	e3	e4	e5	e6
e1		T	T	T	T	T
e2				T		T
e3				T	T	T
e4						T
e5						T
e6						

Table 5.20: The global Enhanced Event Order Table of HMSC of Figure 5.33

5.6.1.2 Occupancy Table for HMSC

As described previously, Occupancy Tables for processes need to be built to avoid deadlocks due to discard of signals by SDL, which that are not expected in one state but maybe needed in later transitions. If a loop presents in an HMSC specification, similar to the approach of handling loop inline expressions in bMSCs, we gather all referenced bMSCs that form the loop, unfold the loop once with all events duplicated and build a local Event Order Table for the loop, then apply the condition to add signals into the Occupancy Tables that may otherwise be missed. For example the MSC specification shown as Figure 5.34, bMSC S1 and S2 forms a loop and we suppose that message m1 and m2 travel different channels.

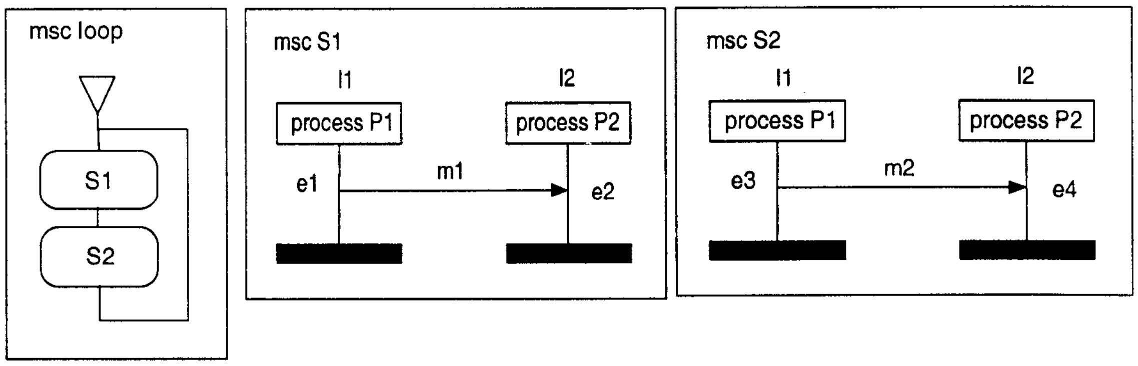


Figure 5.34: An HMSC specification with a loop

We unfold the loop once and get a local Event Order Table for the loop shown as Figure 5.35 and Table 5.21 respectively.

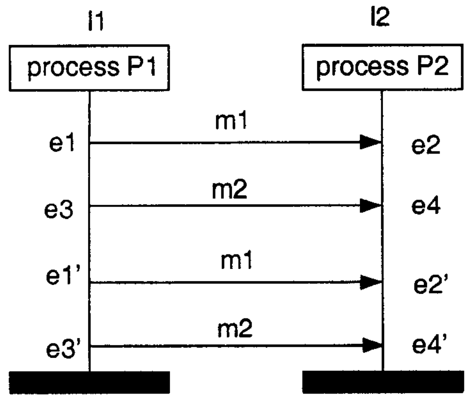


Figure 5.35: Loop unfolded

	e1	e2	e3	e4	e1'	e2'	e3'	e4'
e1		T	T	T	T	T	T	T
e2				T		T		T
e3				T	T	T	T	T
e4						T		T
e1'						T	T	T
e2'								T
e3'								T
e4'								

Table 5.21: A local Event Order Table for an unfolded loop

Finally, the condition for building Occupancy Tables is used; we get m2 enters the row of event e2 and m1 enters the row of event e4 in I2' Occupancy Table.

5.6.1.3 Connecting SDL behavior with states

When generating process behaviors in SDL, we direct the control flow of a process from one HMSC node to the next. SDL states and nextstate construct can be used, if we first assign a state to each process in each node. When the control flow directed by a flow line goes from on node to another, we can take SDL nextstate constructs to make sequential connection. Using this technique, we can simply handle sequential, loop and alternative scenarios in HMSC.

Sequential composition

Consider the following sequential scenario shown as Figure 5.36(a), bMSC S1 is followed by S2; S1 and S2 come from Figure 5.33. Suppose we have assigned an SDL state name for instance I1 in S1, named st1, an SDL state name for the same instance in S2, named st2. The process behavior can be easily generated though connecting the two states as shown in Figure 5.36 (b).

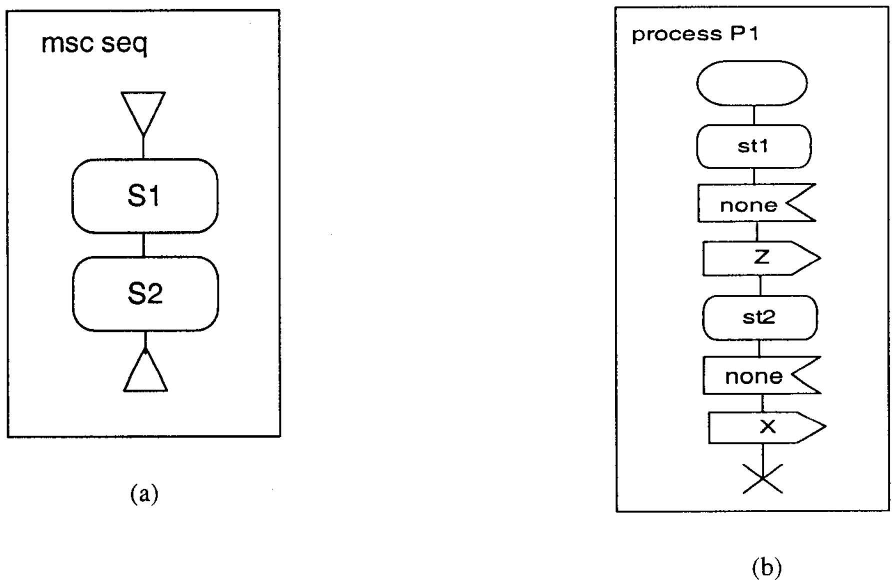
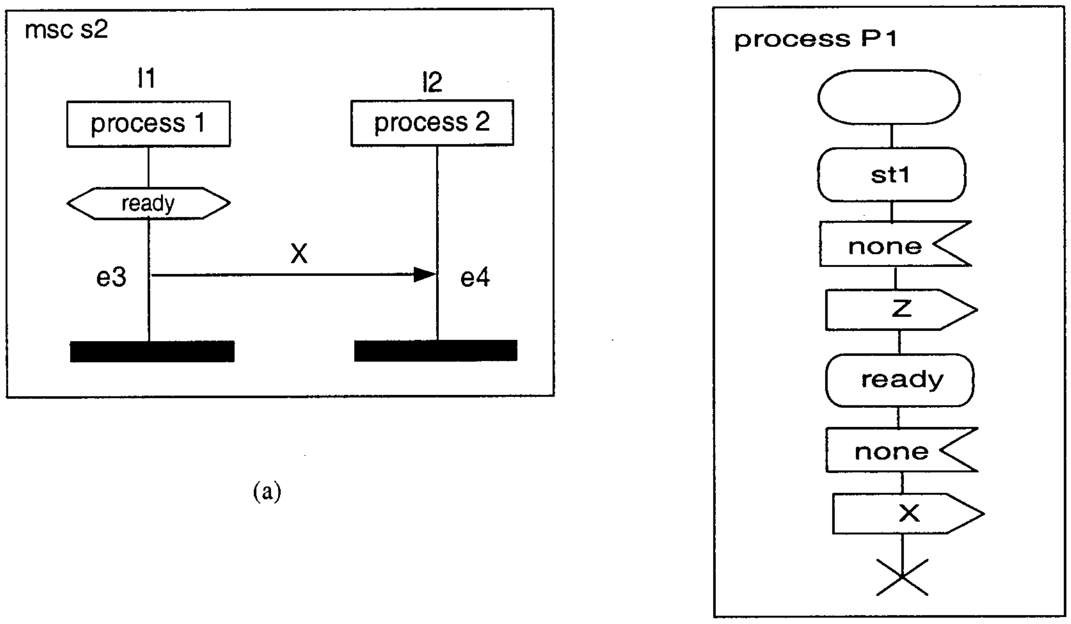


Figure 5.36: An HMSC sequential scenario and generated process behavior in SDL

The assigned state names are unique, but instances in one bMSC node may use the same state name, since state names are local to individual processes. In the case that there is a setting condition at the beginning of an instance in a bMSC, the condition name is used as the state name for the instance in this bMSC, instead of generating one. This is because we can map MSC setting conditions as a SDL states. For example, if S2 is the one shown in Figure 5.37(a) and S1 remains the same, the generated process behavior for P1 uses the condition name as its state as shown in Figure 5.37(b).



(a)

(b)

Figure 5.37: An example of MSC setting condition becoming SDL state

Loop

Using SDL states to connect process behavior for a HMSC loop scenario makes no difference from the way we handle sequential scenarios. For example, if we have an

HMSC specification shown in Figure 5.38(a), and bMSC S1 and S2 still come from Figure 5.33, the generated process behavior uses the state to achieve the iterations as shown in Figure 5.38(b).

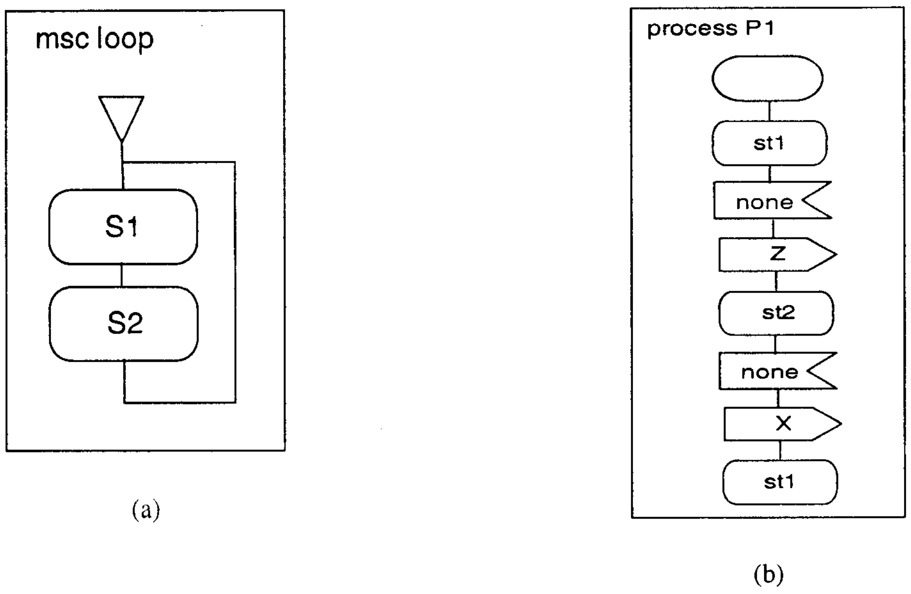


Figure 5.38: An HMSC loop scenario and generated process behavior in SDL

Similarly for loop scenarios, if there is a setting condition at the beginning of an instance in a bMSC, the condition name is used as the state name for the instance in this bMSC, instead of generating one.

Alternative composition

Similar to MSC inline expressions, an alternative scenario in HMSC usually has a controller instance that decides which alternative to take through sending other instances messages and other instances follow the decision. The approach handles the controller instance and the non-controller instances differently. For the controller instance, we assign a state name to the connection point. In its process behavior, the state is followed by a spontaneous transition and a nondeterministic decision. Then all the alternative bMSCs are connected to the answers by using their assigned states. For any non-controller instance, the state name assigned for the connection point is used for all alternative bMSC nodes as a decision point, and state names assigned for individual alternative bMSCs are not used. In fact, the inputs of different signals in the alternative bMSCs are connected directly the decision point.

Consider the HMSC shown in Figure 5.39(a), with bMSCs S1, S2 and S3 coming from Figure 5.33. Suppose we have assigned a state name called alter to the connection point L2, state names st1, st3 and st4 to L1, L3 and L4 (bMSCs S1, S2 and S3) respectively. The end node L5 also has a state name st5. The process behaviors can be obtained as we have described, shown as Figure 5.39(b) and (c).

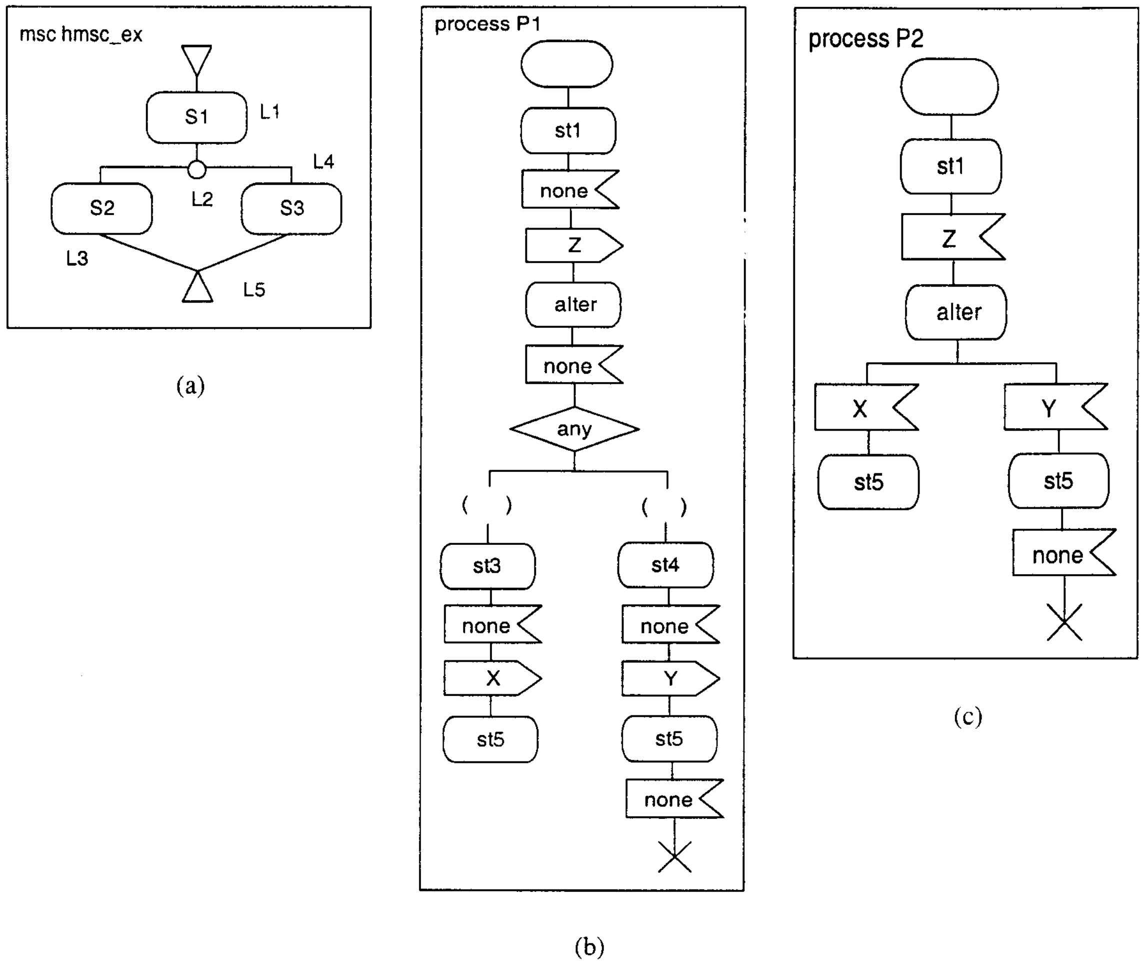
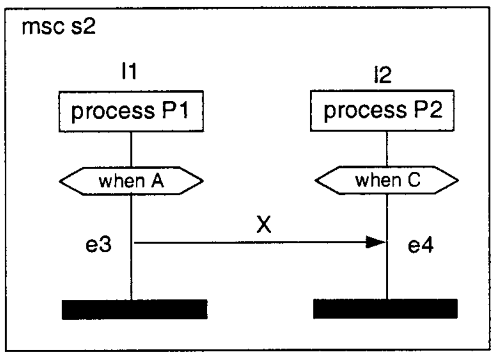


Figure 5.39: A HMSC alternative scenario and generated process behavior in SDL

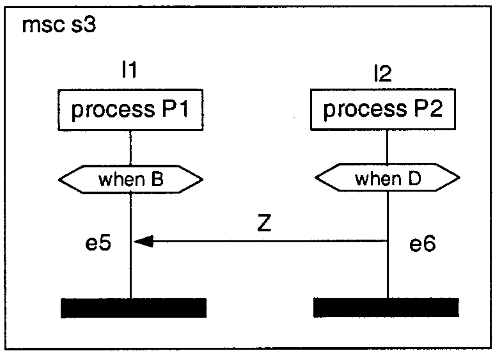
Finally, if the alternative scenario finishes with all alternatives join together as shown in example, the process behavior is connected with the state name of the following node (st5). Otherwise, each alternative branch connects to its own successor's state.

In the cases that there is a setting condition in any non-controller instance at the beginning of alternative bMSCs, we take the condition name as the state name of the connection point for the non-controller instance. For example, if a setting condition named “waiting” is specified for instance I2 both at the beginning of bMSCs S2 and S3, the “waiting” should replace “alter” in the process behavior of P2.

There are also cases that in an alternative scenario, there is no controller instance, but decision is made purely by guarding conditions. In SDL generation, we treat all instances as non-controller instances. A state is still used at the connection points, and then the guarding condition node from each alternative bMSC becomes a continuous signal following this state. For example, we consider the same HMSC shown as Figure 5.39(a), but S2 and S3 become as shown in Figure 5.40 (a) and (b), suppose A, B, C, and D are boolean expressions. Process behaviors can be generated shown as Figure 5.40(c) and (d).

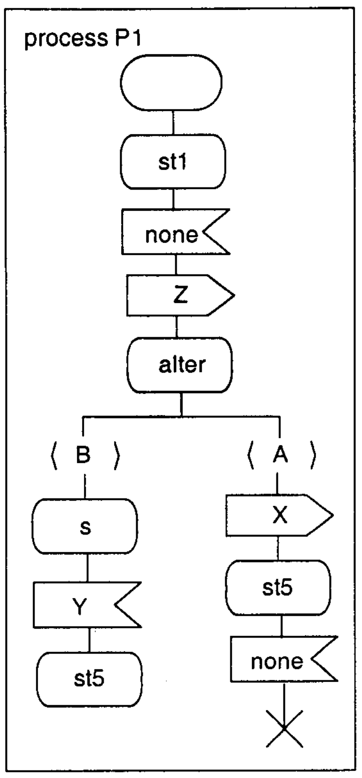


(a)

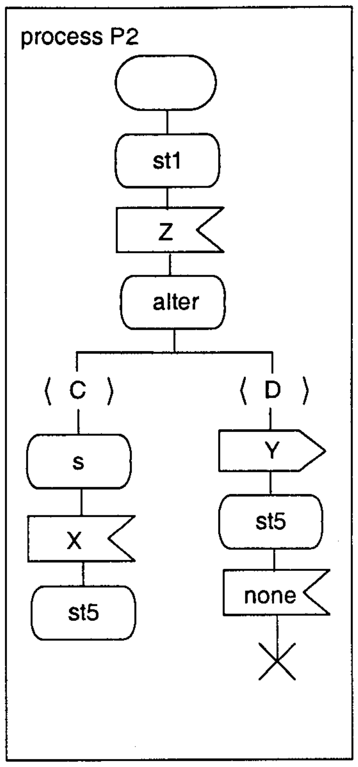


(b)

Figure 5.40: An HMSC alternative scenario with guarding conditions and generated process behavior in SDL



(c)



(d)

Figure 5.41: An HMSC alternative scenario with guarding conditions and generated process behaviors in SDL (continued)

5.6.2 Generating SDL from timed HMSC: An example

A timed HMSC specification and SDL architecture is given in Figure 5.41. We present the major steps to show the whole process that of translating a timed HMSC to SDL.

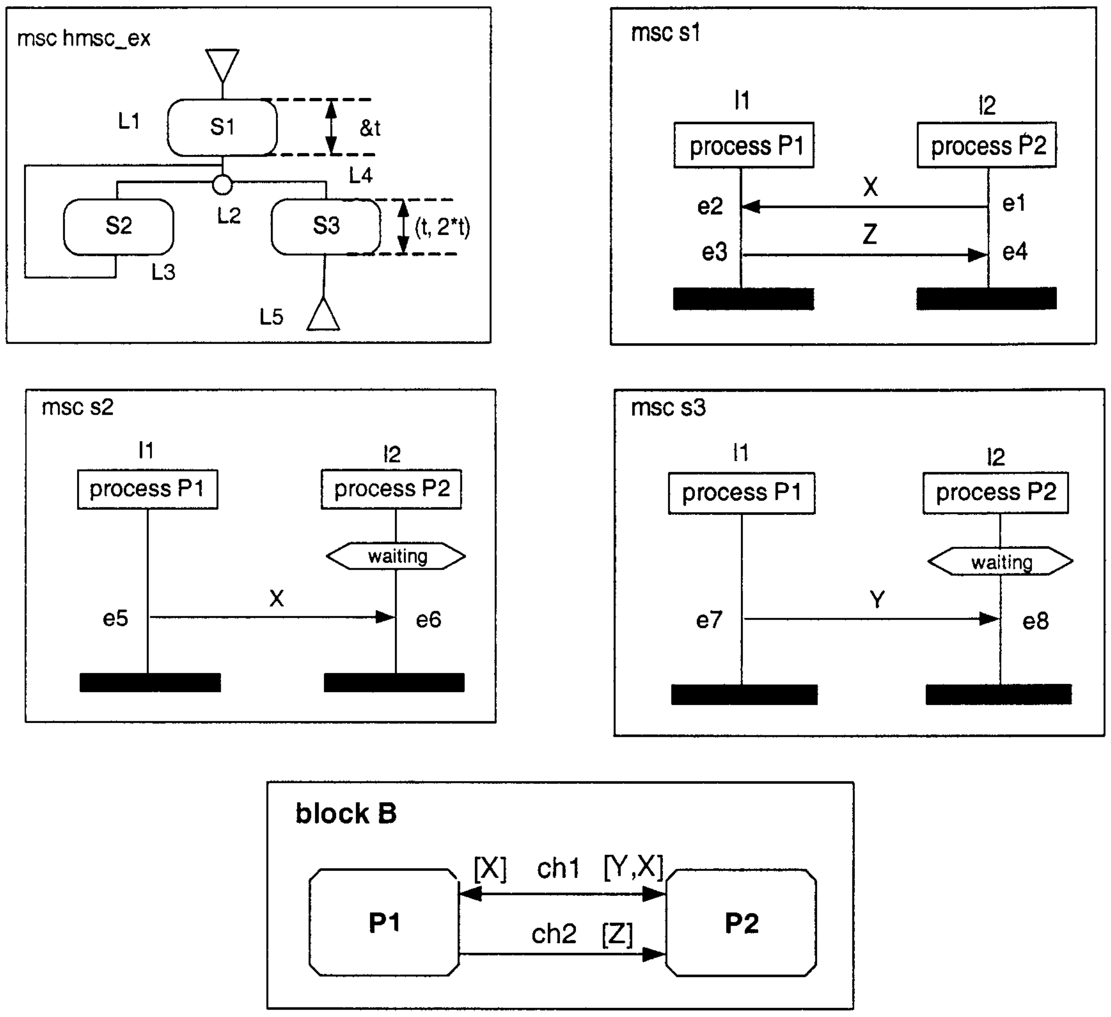


Figure 5.41: A timed HMSC and given SDL architecture

First, we can build the individual Events Order Tables as shown in Table 5.22. Note that for the time measurement of bMSC S1, the first and last events are identified from the order relation among the events as e1 and e4 respectively. The time requirement is then filled into the table cell (e1, e4). Similar for the relative measurement for bMSC S3, the first event is e5 and the last event is e6. Then the relative time constraint is filled in the cell (e5, e6).

	e1	e2	e3	e4
e1		T	T	T&t
e2			T	T
e3				T
e4				

	e5	e6
e5		T
e6		

	e7	e8
e7		T&(t,2*t)
e8		

Table 5.22: Individual Enhanced Event Order Tables of bMSC of Figure 5.41

Then we can build an order table for HMSC nodes (Table 5.23) and then obtain the global Enhanced event Order Table (Table 5.24) through applying the stated combining rules. For example, since the shortest path from node L1 to L4 is 2, node L1 precedes L4. According to the weak sequence semantics, event e1 precedes event e8. Then the Enhanced Event Order Table updates itself by calculating order transitive closure for each event. The results are shown as Table 5.19 and Table 5.20.

	L1	L2	L3	L4	L5
L1	0	1	2	2	3
L2	2	0	1	1	2
L3	1	2	0	3	4
L4	inf	inf	inf	0	1
L5	inf	inf	inf	inf	0

Table 5.23: All pair shortest path for the HMSC of Figure 5.41

	e1	e2	e3	e4	e5	e6	e7	e8
e1		T	T	T&t	T	T	T	T
e2			T	T	T	T	T	T
e3				T	T	T	T	T
e4						T		T
e5						T	T	T
e6								T
e7								T&(t,t*2)
e8								

Table 5.24: Enhanced Event Order Table of HMSC of Figure 5.41

The next step is to build Occupancy Tables for process P1 and P2, since both of them have input events. Table 5.25 shows the two tables. For example, when P2 consumes message Z, both signal X and Y maybe in the input queue as shown in Table 5.25 (b).

input events	input message	Channel ch1
e1	X	X

(a)

input events	input message	Channel ch1	Channel ch2
e4	Z	X,Y	Z
e6	X	X,Y	
e8	Y		

(b)

Table 5.25: Occupancy Tables of example of Figure 5.41

For time requirements in the specification, Variablemap table is also built, which is shown as Table 5.26. For example, the time measurement specified for node L1 actually has e1 and e4 as its Frontevent and Backevent respectively. Since both of them are in the same instance, only FrontVariable and UserVariable are used in its Variablemap. The relative time constraint for node L4 has e7 and e8 as its Frontevent and Backevent respectively. A StampVariable is also used because event e8 is both Backevent and Carryinevent

Front_ event	Back_ event	Carryout_ event	Carryin_ event	Front_ Variable	Carry_ Variable	User_ Variables	Stamp_ Variable
e1	e4	NIL	NIL	timevar1	NIL	t	NIL
e7	e8	e7	e8	timevar2	timevar3	t	timevar4

Table 5.26: Variablemap Table of HMSC of Figure 5.41

Next, we assign states to HMSC nodes. States names st1, st2, st3, st4, and st5 are given to HMSC nodes L1 to L5 respectively.

Finally, we generate process behaviors. For each process, we BFS (Breath First Search) traverse the HMSC graph from the start node and connect behavior of the process using the assigned states. The generated process behavior is as shown in Figure 5.42. Note that process P2 uses the setting condition name as the state name of the decision point, instead of using the assigned state name "st2" for L2. This is because P2 is the non-controller process in the alternative scenario, and there is an setting condition "waiting" at the beginning of both bMSCs S2 and S3; The state names "st3" and "st4" are also not used by process P2. Moreover, there is no signal save construct for Y when process P2 is consuming signal X, since signal X and Y travel the same channel.

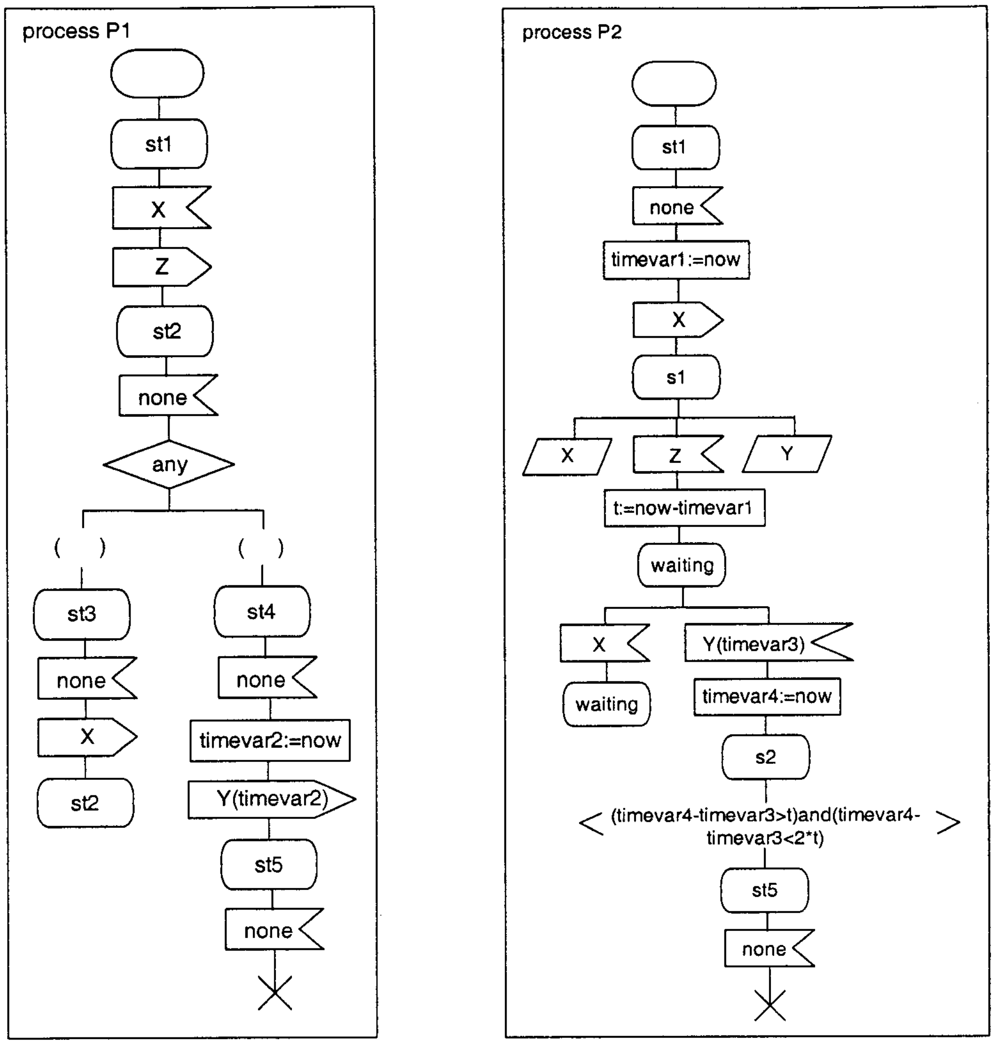


Figure 5.42: Generated process behavior in SDL for the example of Figure 5.41

5.6.3 Non-local choice in HMSC

We have discussed the approach of translating HMSC specifications into SDL specifications in the previous sections. However, not all HMSC specifications can be implemented in SDL. Similar to bMSCs with inline expressions, HMSC may have non-local choice situations. [7] has addressed this problem in HMSC and has given criterion to detect non-local choice in HMSC.

(1) If bMSCs of the same alternative are controlled by different instances, non-local choice exists.

(2) In case of nested alternatives, the following two conditions have to be satisfied to ensure the presence of non-local choice:

- The lower alternative and the higher alternative are not controlled by the same instance.
- The controller instance of the lower alternative is not depending on the controller instance of the higher alternative.

However, as non-local choice scenarios in inline expressions that are caused by system architecture, HMSCs may also have such situations. For instance, if signal X and Y travel the different signal channels in the example shown in Figure 5.41. Instance I2 cannot find out the decision made by I1 if it receives signal Y. This is because instance I1 may decide execute bMSC S2 then S3; it sends signal X then Y. However, it is possible that signal X is delayed and signal Y is received by instance I2 first. Instance I2 does not know the decision made by I1 was either to execute S2 then S3 or to execute S3 directly.

We can generalize the cause of this kind of non-local choice as the controlling signals for the alternative do not travel the same channel, and one or more alternative bMSCs can be optional in the execution. Therefore, we give the following criterion to detect non-local choice in HMSC that caused by system architecture.

(1) There are one alternative bMSC precedes another according to the order relation calculated for HMSC nodes.

(2) The control signals for these two alternative bMSCs travel different signal channels by the given system architecture.

If both of the conditions are satisfied, non-local choice exists.

5.7 Discussion

We have also explored other MSC2SDL related problems that we have encountered along our study.

5.7.1 Shared condition as synchronization point

In MSC, a global or non-local condition shared by more than one instances requires synchronization among the sharing instances. The static requirements of shared condition say "if instance a and b share the same <condition> then for each message exchanges between these instances, the <message input> and <message output> must be placed either both before or after the <condition>" [5]. In other words, between instances a and b, a message sent before the condition need to be received before the condition. That is, if we consider the shared condition a system state, then each of the sharing instances has to wait all other sharing instances reach this state before proceeding further.

For example, in the MSC specification shown in Figure 5.43, Instance I1 sends m1 to instance I2 before shared condition "ready", and then it sends m2 to instance I2 after. According to the requirement stated above, after sending m1, instance I1 reaches state "ready", it cannot send m2 until instance I2 has received m2 and also reaches this state.

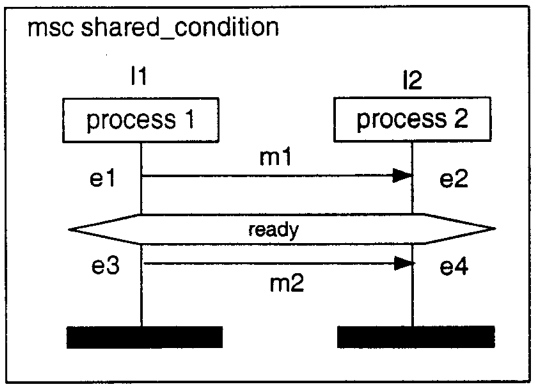


Figure 5.43: An MSC with shared condition

A simple way of implement this synchronization is to use a shared variable among related instances as a semaphore, which blocks evolution of shared instances until they all reach the intended state. In SDL-2000, a variable can be defined in block level so that all contained processes have access to it (read and write). This mechanism is done through two implicit remote procedure calls set_v and get_v provided by the generated state machine of the block. Therefore, this synchronization required by MSC shared condition should get translated into SDL gracefully using this mechanism. However, the current ObjectGeode version does not support block level variables.

Another possible way is to use export/import or revealed/view construct provided by SDL. However, since these two types of commands only grant a process to read the value (not to write) of a variable from a different process, in orcer to achieve a full synchronization among a group of processes, every pair of processes have to declare both revealed and view (export/import) variables for each other, or we can use a tree-like structure with intermediate instances to reduce the amount of revealed and view (export/import) variables. In other words, the synchronization is achieved in a distributed manner. This solution obviously will cause a lot of overhead to the generated SDL design.

Therefore, we decided release this requirement on this stage and merely see shared a shared condition a state with no synchronization. When tools such as ObjectGeode supports block level variables, this requirement can be fulfilled gracefully.

5.7.2 Guarding conditions in MSC

MSC does not provide a formal semantics for conditions. Setting conditions set the system conditions and guarding condition can reference them to constraint the execution traces of the system. However, a condition name is not in the data part of MSC, yet guarding condition may check a condition name as a boolean variable.

For SDL generation purpose, we may define all condition names as boolean variables, and all setting conditions set the corresponding condition name boolean variables to true so that guarding conditions may check their values. This sounds a good solution. However, in many cases more than one condition names may be needed to be combined into one condition according to usage of them. For example, an MSC system might have setting conditions as both "connected" and "disconnected", and normally these two condition names should be two values of one system state variable. If we define two boolean variables for these two conditions, this most likely does not conform to the idea of the system designer. Without knowing the logic relation among condition names, a "merge" of them is impossible. Therefore, in our approach, we do not handle guarding conditions using condition names to avoid the possible ambiguity.

5.7.3 Referenced bMSC into SDL procedures

An HMSC can reference a bMSC more than once. If we generated SDL code for the bMSC every time it is referenced, it seems that we are doing some redundant work since the same bMSC should have the same process behaviors, and repeated behaviors in SDL can be defined as procedures, which can be called by the process state machine. However, we have found that the same bMSCs in different context may result slightly different behavior and the difference comes from signal saves.

For example, Figure 5.44 shows an HMSC specification, where bMSC S1 is referenced twice in this HMSC. For the first reference (Label L1), upon instance I2 consuming m2, m1 (from L2) maybe in its input queue, then an SDL save construct must be generated here. However, upon instance I2 consuming m2 again (from L2), there is no SDL save construct is needed, since all other signals must be consumed before, so m2 is the only signal in the input queue. Therefore, two same bMSC references result different SDL behavior due to the different contexts; we cannot reuse one's SDL code for the other. Therefore, we generated SDL code for a bMSC every time it is referenced.

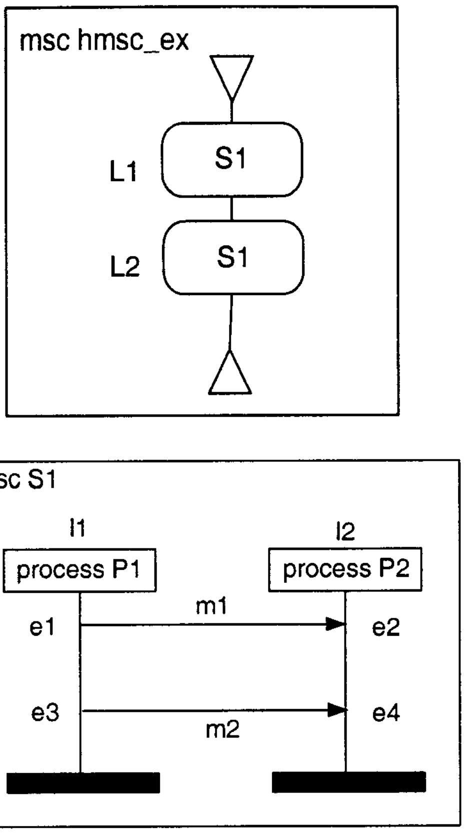


Figure 5.44: An HMSC specification with a bMSC referenced repeatedly

5.8 Related work

In this chapter, we have presented our approach for generating SDL specifications from timed MSC use cases, under a given SDL architecture. The approach checks the syntactical and semantics errors of MSCs, as well as their implementability. In the generated SDL specification, we ensure its equivalence to the given MSC without inducing any design error.

Research on generating SDL from MSC has been active and a lot of work has been done in the recent years. Mansurov and Zhukov proposed an approach to synthesize SDL from

MSC [9]. It has been turned into a commercial product by Klocwork. The main objective of this work is to provide SDL executable specifications and give early feedbacks for the phases of requirement analysis and design [9]. In contrast, in addition to generate SDL behavior, this approach generates SDL architecture as well. However, sometimes this approach has a consistency problem between the given MSC specification and the generated SDL architecture as well as SDL behaviors. Moreover, we have noticed that this approach is not flexible with SDL architecture. Since different SDL architectures may result different process behaviors, and one requirement specification may be used for different target system.

Dulz and Gruhl presented an approach to generate prediction models from MSC specifications. In contrast to our approach, their objective is solely for the purpose of performance prediction [10]. They assume a memoryless process model in that process transitions are atomic and the order of transitions does not affect the state of a process, hence a one state process model. Therefore, the ability of handling complex process behavior is limited. Furthermore, this approach also derives SDL architecture from given MSC using an architecture specification language. This approach is not suitable for prototyping, refinement and simulation purpose.

In his Master thesis [11], Persson has also presented an approach of extracting SDL architecture from MSC. By analyzing MSC specification, it creates SDL system, blocks and processes. This approach also creates a signal channel between each communicating processes in the architecture to convey the exchanging signals. However, this approach is very limited since the SDL architecture is the only concern of this work. The SDL specification is incomplete.

SDL generation from MSC with Real-time requirements has not been addressed in the above-mentioned research. We believe our approach is unique in handling timed MSC into a correct SDL model.

Chapter 6

The MSC2SDL2004 Tool and Case studies

6.1 Overview

The approach described in the previous sections has been implemented in C/C++ on Windows 2000. We have used MFC (Microsoft Foundation Class) for the Graphic User Interface. If needed, the core of this tool can be easily ported to other platforms such as Solaris, Unix or Linux.

MSC2SDL2004 Tool is built to generate SDL designs from MSC specifications. This version is a successor of our existing tool MSC2SDL that handles MSC-96. The tool emphasizes the newly introduced concepts of time and data in MSC-2000. It also handles inline expressions in bMSC. It interfaces with ObjectGeode, as shown in Figure 6.1 We have adopted the textual and graphical representation format of SDL; however, the current version of ObjectGeode does not support MSC-2000 Time and Data concept, and its syntax for HMSC is not the standard form specified in [5]. Therefore, we have used the MSC-2000 standard textual form to represent MSC specification. The MSC2SDL tool takes two files as inputs. One contains the MSC specification, and the other contains the target SDL architecture.

The SDL architecture can be edited using ObjectGeode. Since there is no commercial tool that currently supports MSC-2000 time and data, users need to directly provide MSC specification in a textual form that contains time and data. MSC2SDL2004 Tool reads MSC specification and SDL architecture specification, and builds their internal representations. Then, the inputs are analyzed and SDL specification is generated if there is no semantic error detected by our algorithms. The generated SDL behavior is combined with the architecture to produce an output file, which is in pr file format of ObjectGeode. Therefore, this file can be viewed and modified with ObjectGeode SDL editor, as well as be simulated with ObjectGeode tools.

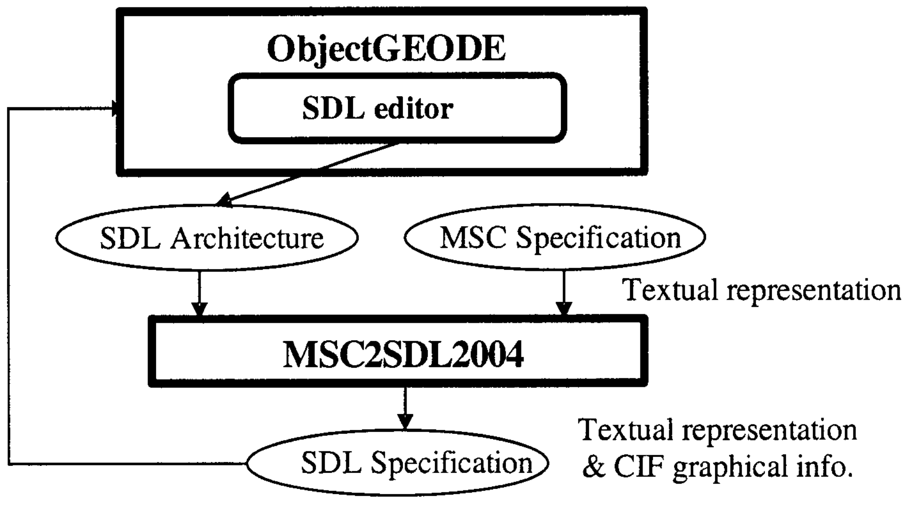


Figure 6.1: Interfacing MSC2SDL2004 with ObjectGeode

6.2 Architecture of the MSC2SDL tool

The tool consists of the following eight main modules: Dispatcher, BmscParser, HmscParser, bMSCProcessor, HMSCProcessor, SDLGeneratorbMSC, SDLGeneratorHMSC, and SDLFileGenerator. Figure 6.2 shows these modules in an architecture diagram.

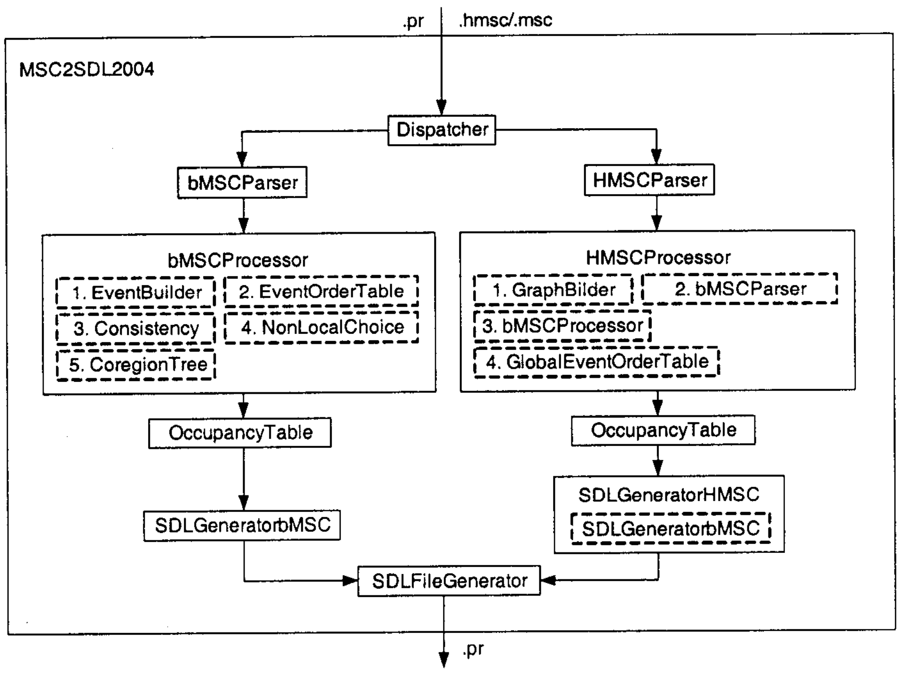


Figure 6.2: MSC2SDL2004 tool architecture

The inputs of the MSC2SDL2004 are an MSC files and an SDL architecture file.

The functionality description of the main modules and sub-modules are as the following:

(1) Dispatcher: distributes inputs into proper processing path according to different types of MSC.

(2) bMSCParser: checks the syntax of the input bMSCs.

(3) HMSCParser: checks the syntax of the input HMSCs.

(4) bMSCProcessor: analyzes bMSCs.

- EventBuilder: builds internal presentation of MSC events.

- EventOrderTable: builds Enhanced Event Order Table.
- Consistency: checks architecture consistency between the bMSCs and the given SDL architecture.
- NonLocalChoice: detects the existence of non-local choice in bMSCs.
- CoregionTree: builds coregion trees if there are any coregions in the MSC specifications.

(5) HmscProcessor: analyzes high level MSC.
- GraphBuilder: builds internal presentation of HMSC.
- GlobalEventOrderTable: merges individual Event Order Tables into a global one.

(6) OccupancyTable: builds Occupancy Tables.

(7) SDLGeneratorbMSC: generates SDL constructs for bMSCs.

(8) SDLGeneratorHMSC: traverses HMSC graph and calls SDLGeneratorbMSC to generate SDL constructs for a HMSC.

(9) SDLFileGenerator: generates SDL output file in the ObjectGeode format.

The architecture of MSC2SDL tool shown in Figure 6.2 also demonstrates the execution flow. First of all, the Dispatcher analyzes the input files.

(1) If the inputs are a SDL architecture file and a basic MSC file, the control flow goes to the bMSCParser, which checks the syntax of bMSCs. Then, the bMSCProcessor builds the internal presentation of MSC events and the Enhanced Event Order Table. Next, it does architecture consistency checking and Non-local

choice checking, then creates Coregion trees if there are coregions specified in the bMSC specification. The Occupancy module then builds the occupancy tables. Finally, the SDLGeneratorbMSC generates SDL specification and the SDLFileGenerator creates the output: a pr file that contains the process behaviors in SDL with its architecture.

(2) If the inputs are a SDL architecture file, a high level MSC file and a set of basic MSC files, the Dispatcher module sends them to the HMSCParser, which checks the syntax of HMSC. HMSCProcessor builds the internal presentation of HMSC as a graph, it then calls the bMSCParser and the bMSCProcessor to process each referenced bMSCs as we have described previously. Next, the HmscProcessor combines individual Event Order Tables into a global one then builds the occupancy tables. The SDLGeneratorHMSC then traverses the HMSC graph and calls the SDLGeneratorbMSC to generate SDL behavior for all processes from each referenced BMSC. Finally, the SDLFileGenerator creates the output: pr file, as the final step.

One of the important aspects for implementing our tool is about MSC-2000 parser. Since there is no commercial compiler for this latest MSC version, we have built our own parser that handles time and data in MSC2000. The building modules for bMSCParser and HMSCParser are lexical analyzer, pre-parser, syntactic analyzer, and semantics analyzer. Inputs to this parser are bMSC and/or HMSC textual files; the outputs are arrays of internal representation of MSC statements. The parser reads the input MSC files, and generates tokens that are specified in the MSC-2000 lexical rules. Then the Pre-parser is called to resolve names and variable types according to naming rules. The syntactic analyzer parses the token stream from the Pre-parser to check the syntax of the input file according to MSC-2000 grammar. Finally, Semantic analyzer detects semantic errors. Figure 6.3 shows the internal structure of the parser.

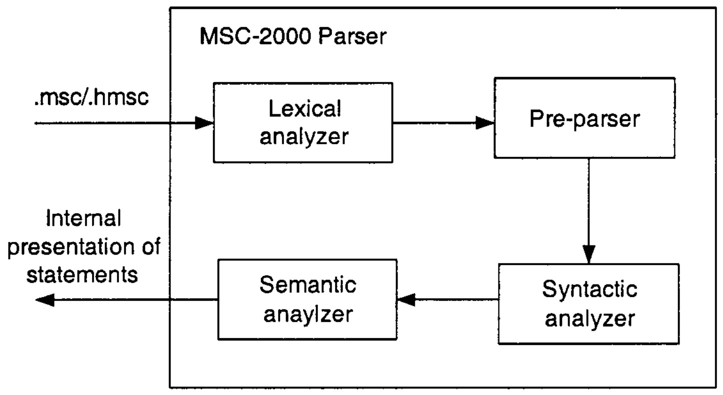


Figure 6.3: MSC-2000 parser

This parser is a reduced is based on a reduced version of MSC-2000. We have omitted MSC references in bMSCs, HMSC references in HMSC, instance decomposition, Gates, Data in instance creations, etc. The reduced grammar simplifies the design of the parser, and it is sufficient for the purpose of this study on the current stage.

6.3 Interface

Users run this tool by clicking the MSC2SDL2004 icon. The execution initiates a Graphic User Interface shown in Figure 6.4. The window contains various sections for users to use MSC2SDL tool easily.

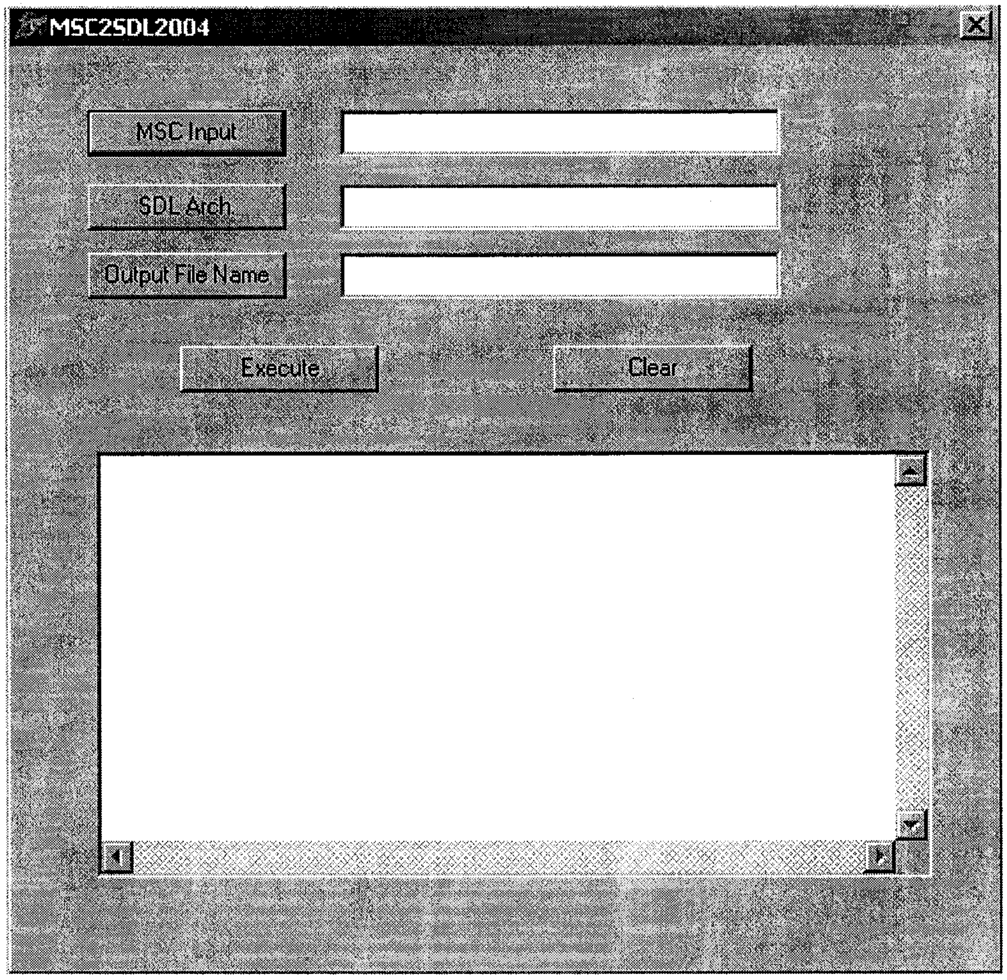


Figure 6.4: The tool interface

To run the application, the user needs to input MSC source files and SDL architecture files by clicking the "MSC Input" and "SDL Arch." buttons, a new window pops up each time to let the user to select MSC or SDL source file through a file browser. If the MSC input file is an HMSC, all referenced bMSC by this HMSC must be placed in the same folder as this HMSC. The default file name for generated SDL file is sdlout.pr and will be placed in the same folder as the selected SDL architecture file. User has the option to overwrite the default path and name. The "Execute" button initiates the translation. The processing information is shown to the user in the display area. The user can use the "Clear" button to reset the tool to its initial state after each execution.

6.4 ObjectGeode

ObjectGeode is a toolset dedicated to analysis, design, verification and validation through simulation, code generation and testing of real-time and distributed applications. ObjectGeode supports a coherent integration of complementary object-oriented and real-time approaches based on the SDL and MSC standards languages [12]. ObjectGeode provides graphical editors, a powerful simulator, a validation tool, and a C code generator. Our work adopts the SDL pr format of ObjectGeode. The required SDL target system architecture in our approach can be edited using ObjectGeode SDL editor. Furthermore, the generated SDL behavior can be viewed, edited, checked, and simulated using the ObjectGeode toolset. The detailed information of ObjectGeode can be found in the User Manuals and Tutorials [18].

6.5 Case Studies

We present in this section two applications.

6.5.1 Automatic Teller Machine (ATM)

The ATM system is used to for self serve Banking. It usually allows users to access their accounts and perform transactions such as balance inquiry, deposits, withdraw, etc. We have modeled a simple ATM system as a case study for the tool.

MSC specification

The MSC specification of the system is shown in Figure 6.5, which has time requirements, an inline expression and various HMSC operators. The system consists of two actors: the ATM machine and the Bank. The system interacts with the user represented as the environment of the system. For illustration purpose, we have specified two ATM functions as well as the user logon process. The HMSC specification references some bMSCs. For example, in bMSC SATRTUP, the user supplies the card and the ATM machine initializes a counter and sets up a timer for the purpose of controlling the user's login process. If the user does not enters the password to the ATM machine before the timer expires, the ATM returns the card to the user and informs the user timeout, which is bMSC TRY_AGAIN. Then the system returns to its initial state. Otherwise, the ATM machine requires the Bank to verify the password and updates the counter. This scenario is represented by bMSC PROCESS_PIN; and a time constraint is specified that requires the time duration from sending signal Verity from the ATM machine to its reception by the Bank should be within (0,2). If the password is incorrect, the Bank informs the ATM machine and the ATM machine informs the user to reenter its password (bMSC REENTER_PIN). The user has three chances to give the correct password. Otherwise, the card gets swallowed by the ATM machine (bMSC SWOLLOW_CARD) and the system returns to its initial state. If the password is correct according to the verification by the Bank (bMSC PIN_OK), the user is given three options.

(1) Get balance of his account (bMSC GET_BALANCE).

The user inquires its account balance; the ATM machine relays this inquiry to the Bank. The Bank gets the account balance then the ATM relays it to the user. The system gets back the option state the user to perform the next transaction. A relative time constraint is specified for the ATM machine that requires the time duration from sending signal Get_Balance to the Bank and the reception of signal Balance from the Bank should be within [1,5].

(2) Withdraw allowable amount of money (bMSC WITHDRAW).

The user enters an amount to withdraw; the ATM machine then sends this inquiry to the Bank. If this amount is not approved (less than the account balance, for example), the Bank informs the ATM machine and the ATM machine relays this information to the user. Otherwise, the Bank informs the ATM machine to disburse money to the user. Then the ATM machine inquires the account balance to the Bank. When the information is sent back, the ATM machine sends it the user. The system gets back the state status the user to perform the next transaction. A relative time constraint is specified for the alt inline expression that requires the time duration from reception of signal Ent_Amount by the ATM machine to the end of the expression (either the transaction is approved and the ATM disburse the money then prints the transaction record, or otherwise the ATM machine informs the user the withdraw amount is not allowed) should be within (3, 10).

(3) End the transaction (bMSC CANCEL_TRANS).

If the user decides to end the transaction, the ATM machine informs the Bank and returns the card to the user. The system returns to its initial states.

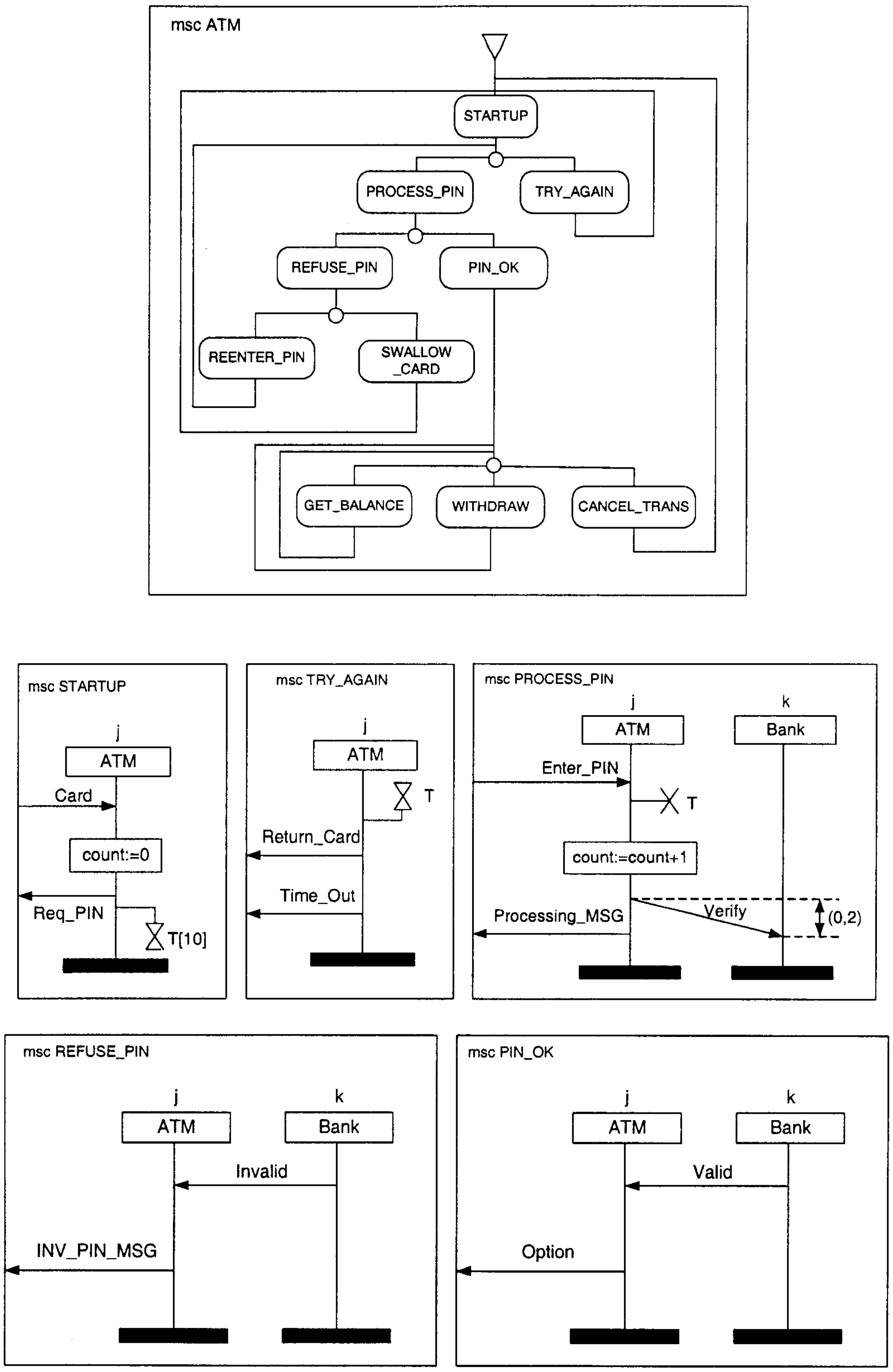


Figure 6.5: The MSC specification for the ATM system

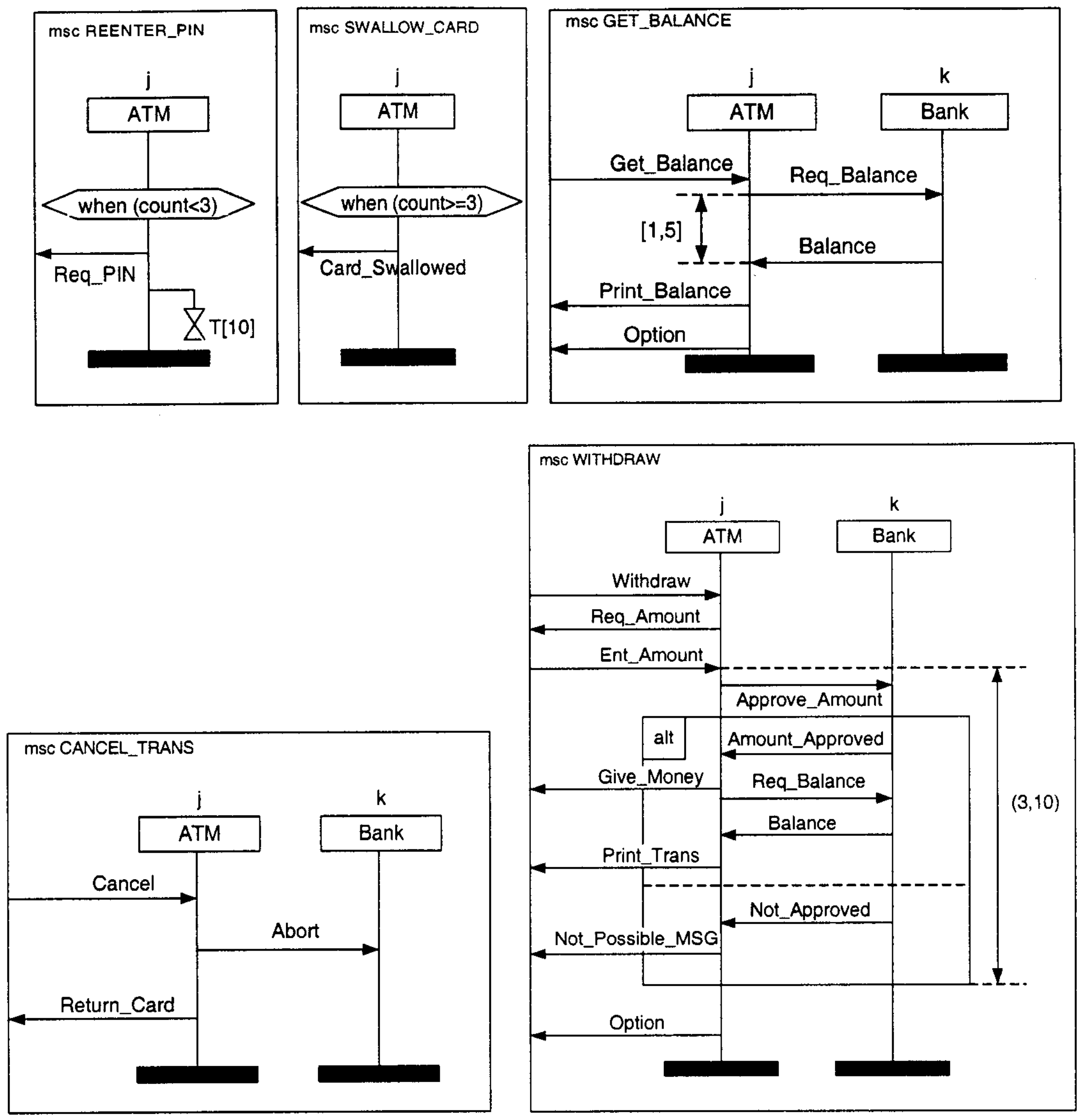


Figure 6.5: The MSC specification for the ATM system (continued)

SDL architecture

Figure 6.6 shows the SDL architecture of the ATM system. In the system level, it consists of ATM_block and a signal channel connecting to the environment. Inside ATM_Block, there are two signal channels sr1 and sr2. Channel sr2 connect process ATM to process Bank, and channel sr1 connects process ATM to the block environment with signal channel ch.

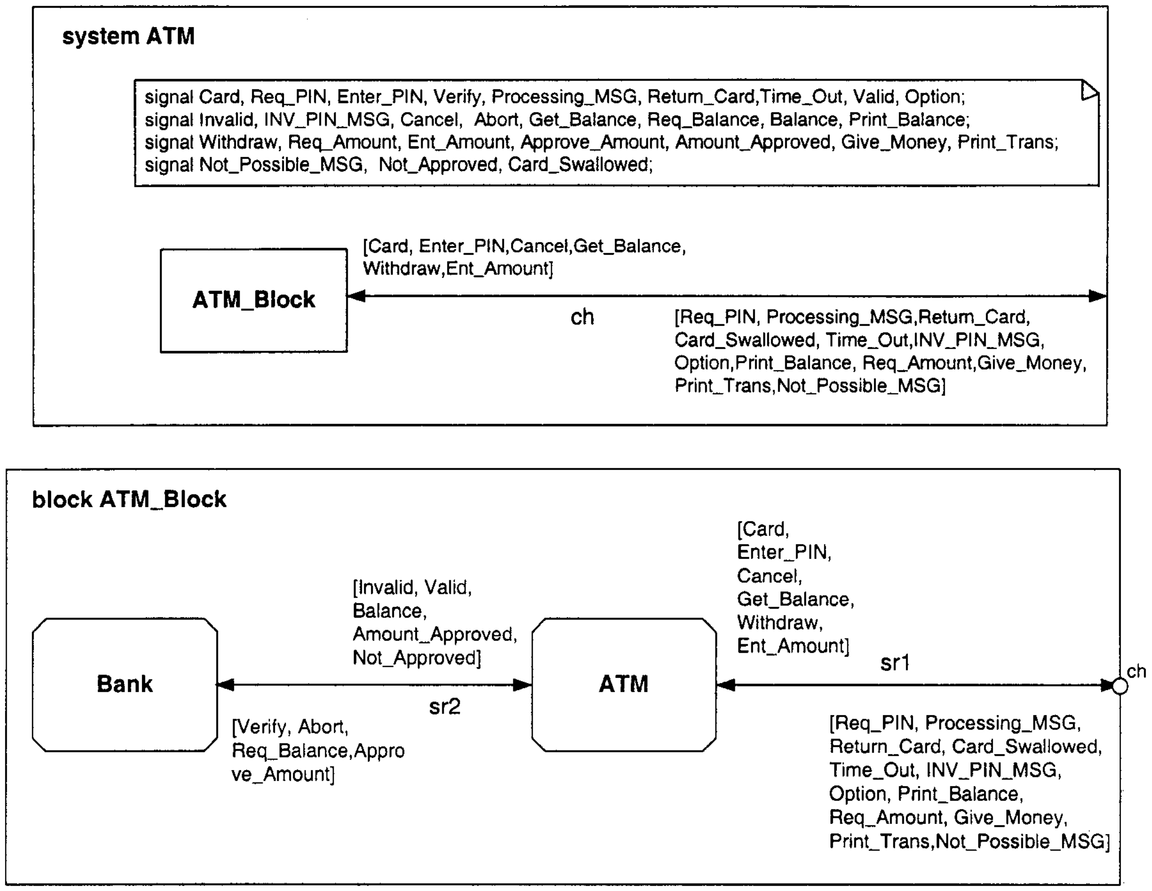

system ATM
signal Card, Req_PIN, Enter_PIN, Verify, Processing_MSG, Return_Card,Time_Out, Valid, Option;
signal Invalid, INV_PIN_MSG, Cancel, Abort, Get_Balance, Req_Balance, Balance, Print_Balance;
signal Withdraw, Req_Amount, Ent_Amount, Approve_Amount, Amount_Approved, Give_Money, Print_Trans;
signal Not_Possible_MSG, Not_Approved, Card_Swallowed;
ATM_Block
[Card, Enter_PIN,Cancel,Get_Balance, Withdraw,Ent_Amount]
ch
[Req_PIN, Processing_MSG,Return_Card, Card_Swallowed, Time_Out,INV_PIN_MSG, Option,Print_Balance, Req_Amount,Give_Money, Print_Trans,Not_Possible_MSG]
block ATM_Block
Bank
[Invalid, Valid, Balance, Amount_Approved, Not_Approved]
sr2
[Verify, Abort, Req_Balance,Approve_Amount]
ATM
[Card, Enter_PIN, Cancel, Get_Balance, Withdraw, Ent_Amount]
sr1
ch
[Req_PIN, Processing_MSG, Return_Card, Card_Swallowed, Time_Out, INV_PIN_MSG, Option, Print_Balance, Req_Amount, Give_Money, Print_Trans,Not_Possible_MSG]

Figure 6.6: The SDL architecture for the ATM system

Non-local choice

MSC2SDL2004 tool has detected a possible non-local choice problem in the given MSC specification. It is at the alternative operator connecting bMSC PROCESS_PIN and TRY_AGAIN. Since PROCESS_PIN is controlled by the environment and TRY_AGAIN is controlled by ATM. According to our non-local choice detecting criteria, this is considered as a non-local choice. However, in fact the scenario is controlled by the environment and a timer, and the choice is deterministic. We can actually classify this type of scenarios and handle it properly in the future development our approach and tool.

SDL Generation

One assumption we have made in applying this case study is that the environment is fully cooperative with the system. That is, the environment sends signals to the system according to the requirements from the system and never sends signals in a random order. This gives clearer generated process behaviors in SDL without so many signal saves that may be caused by the system architecture. However, another possible assumption about the environment is that the environment contains more than one process that behaves in a random order. In this case, if the signals travel multiple channels, the system need to generate more signals saves since it has no idea about the sequence of signals sent from the environment. The tool can also be configured to adopt the second assumption.

The generated process behaviors are shown in Figure 6.7 and Figure 6.8. The process behavior for ATM is split into two pages. The user defined variable "count" and timer T have been declared in the SDL process. There are also some generated time variables that are used to translate the time requirements specified in the MSC specification. Process Bank also has one time variable declared for the time requirement. Moreover, there are actually some redundant states with spontaneous transition in its behavior, such as state S0. This problem is caused by our translation algorithm, which generates a state for the process when each referenced bMSC is processed even though the bMSC contains no behavior for the process. Here, Bank does not appear in bMSC STARTUP, and our algorithm still gives a state to the process. This problem can be solved with one more step of removing redundant states after SDL generation. Whenever there is a state with only a spontaneous transition to the next state, the state can be removed. Consequently, the nondeterministic behavior of state S8 also disappears after state S0 is removed. Finally, since ATM and Bank are in the same block, they address each other their process names. ATM sends signals to the environment using implicit addressing.

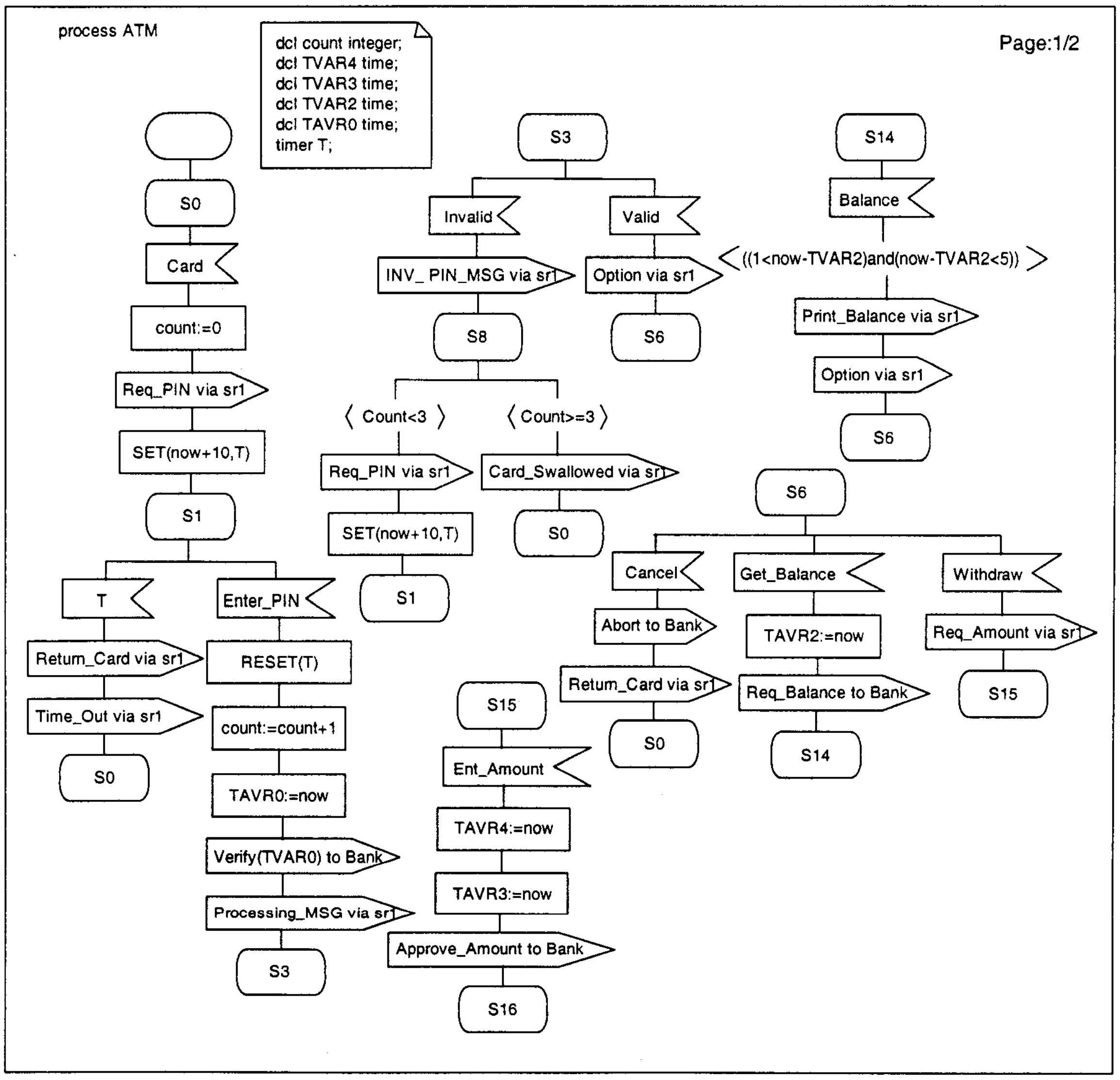


Figure 6.7: Process behavior for ATM

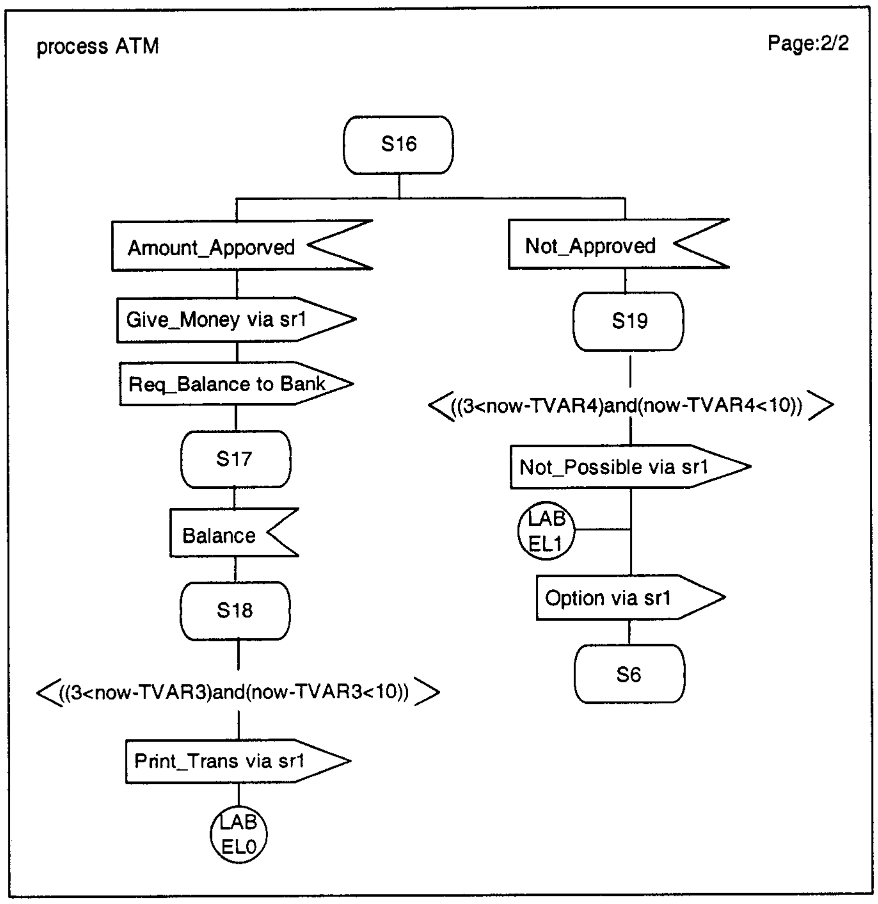


Figure 6.7: Process behavior for ATM (continued)

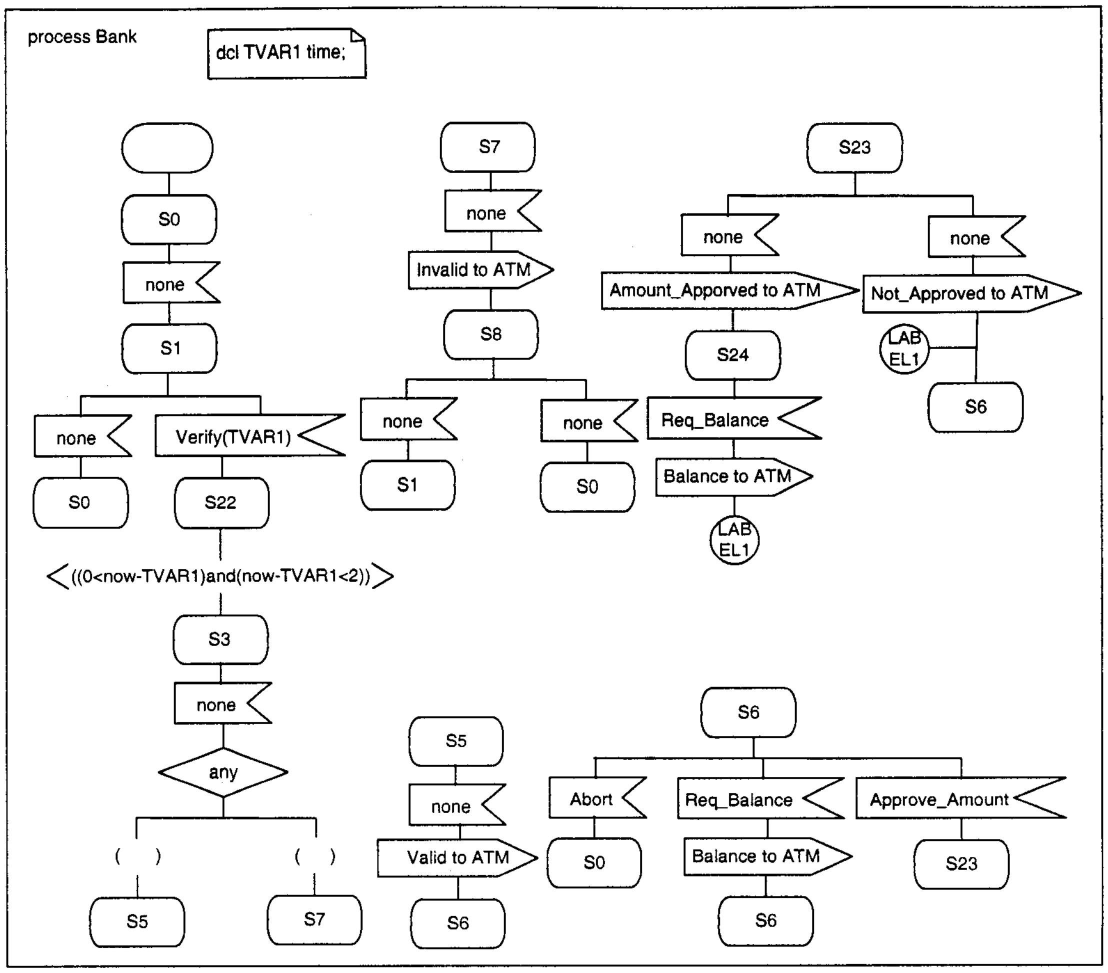


Figure 6.8: Process behavior for Bank

6.5.2 Shuttle system

The shuttle system is rail-based transport system described in [17]. The system consists of a railway network, shuttles, and orders for shuttles to accomplish. After a shuttle obtains an order, it delivers the order to its destination. Then, a shuttle may get a payment or pay a charge depending on whether or not it meets the specified deadline.

MSC specification

We have simplified the shuttle system and modeled it with MSC. The MSC specification is shown in Figure 6.9, which has time requirements, an inline expression various HMSC operators, and MSC states that helps to achieve the concurrency between the shuttle and the broker agent.

The system consists of three actors: the shuttle, the broker agent and the bank. The broker agent generates and announces orders for the shuttle to accomplish (bMSC ANNOUNCE_ORDER). The shuttle has the choice of biding for an order (bMSC MAKE_OFFER) or not doing so (bMSC NO_OFFER). Even if the shuttle makes an offer, the broker agent still can reject the offer if it considers the offer is proper for the order it has announced (bMSC NO_ORDER). Moreover, if the shuttle gets the order, it will deliver then report to the broker agent (bMSC GET_ORDER_DELIEVR). If the order deadline has passed, the shuttle will pay a penalty. Otherwise, the shuttle gets paid for completing the order. The bank is taking charge of paying or charging the shuttle.

In order for the broker agent and shuttle act independently, we have decomposed the both shuttle and the broker agent into two components. The shuttle consists of process Shuttle and ShuttleExt. The Shuttle bids and takes orders only, and the ShuttleExt delivers the orders. The broker agent consists of the BrokerAgent and BrokerAgentExt. The BrokerAgent generates, announces and assigns to the shuttle only, and the BrokerAgentExt monitors the result of each delivered order by the shuttle. In this way, the shuttle may take more than more one order at any time and the orders are delivered sequentially. Moreover, each actor goes back its original states by using MSC set conditions after one round of operation is completed. For illustration purpose, one assumption we have made in the specification is that, the deadline of each order is the absolute time of 10 times of the order ID x. The order ID x is incremented by 1 each time a new order is generated. The completion time of each order is measured and compared with its deadline. The result is used to decide whether the shuttle should get paid or should pay a penalty.

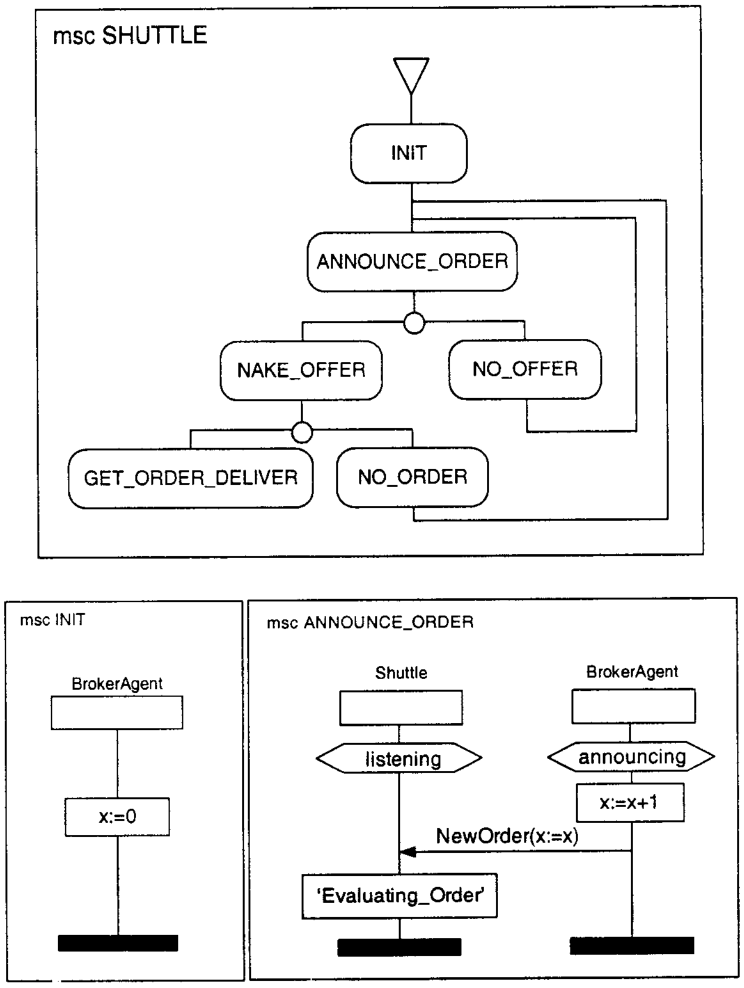
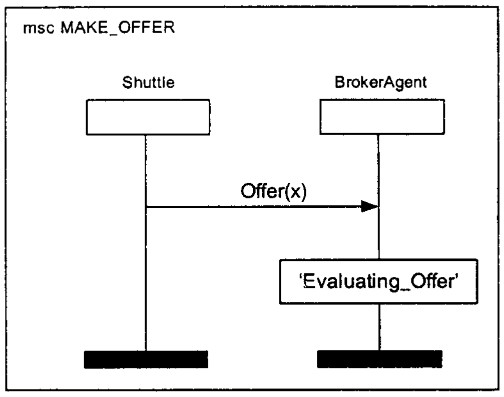
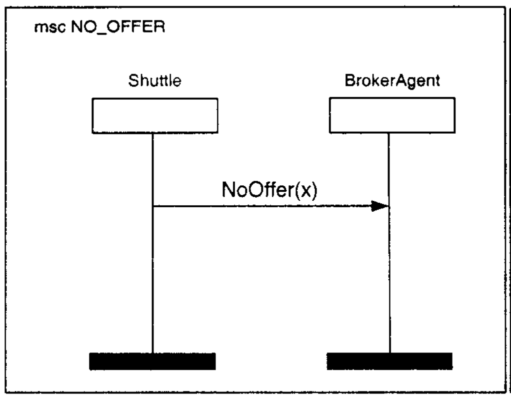


Figure 6.9: The MSC specification for the Shuttle system





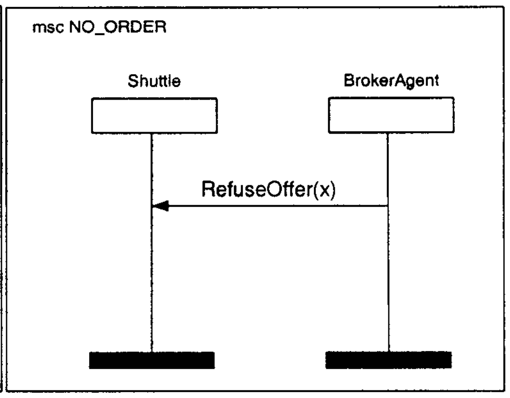


Figure 6.9: The MSC specification for the Shuttle system (continued)

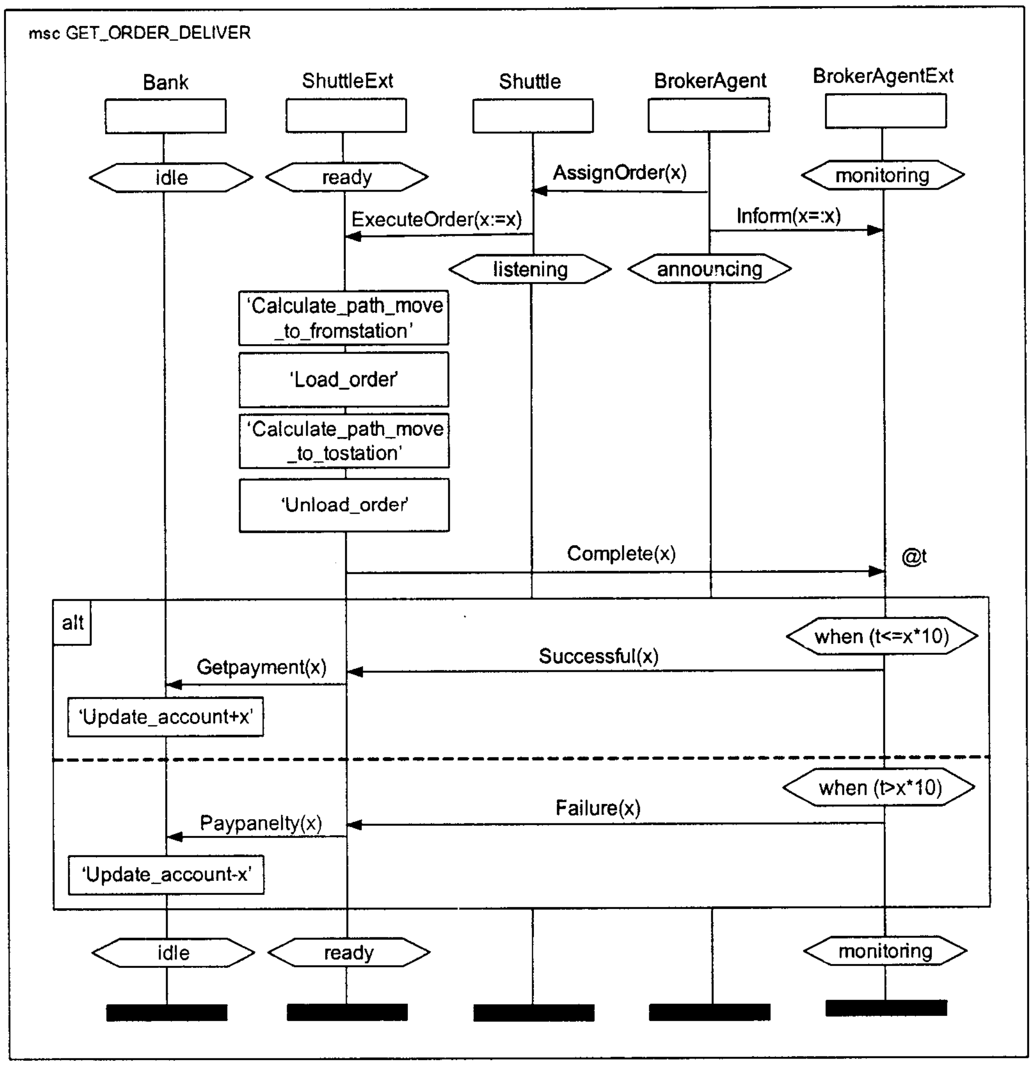


Figure 6.9: The MSC specification for the Shuttle system (continued)

SDL architecture

Figure 6.10 shows the SDL architecture of the shuttle system. In the system level, it consists of three blocks, namely the Bank-Block, the Shuttle_Block and the Broker_Block. These blocks are connected with two signal channels ch1 and ch2. In the Bank_Block is the process Bank, which connects to ch1 through the signal sr4. In the Shuttle_Block, the shuttle has been decomposed into two processes. One is Shuttle and the other ShuttleExt. They are connected with signal channel sr5. ShuttleExt connects to ch1 and ch2 through signal channel sr8 and sr2 respectively. Shuttle connects to ch1 through signal channel sr8. In the Broker_Block, the broker agent has been decomposed into the BrokerAgent and the BrokerAgentExt processes. They are connected through signal channel sr6, and they connect to ch1 through signal channel sr3 and sr7 respectively.

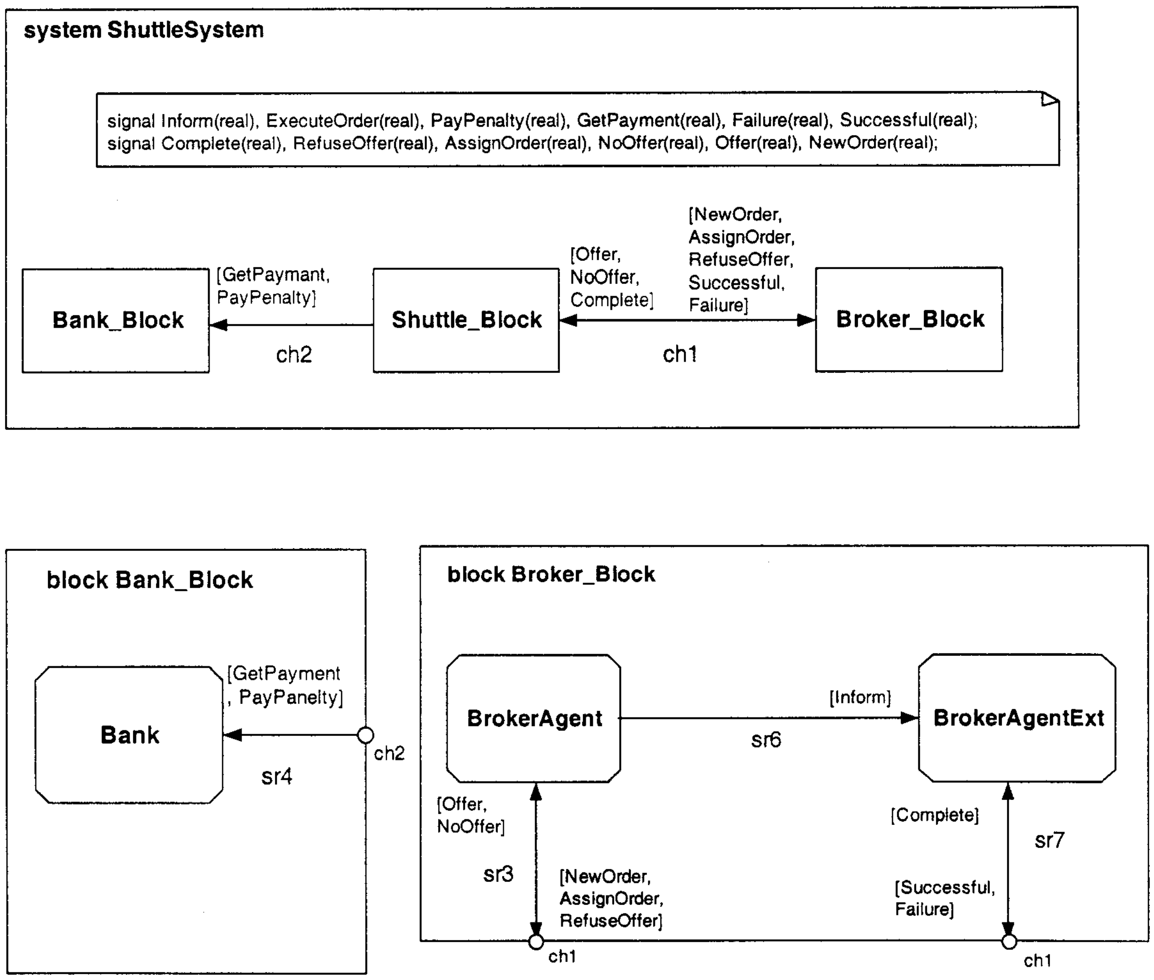

system ShuttleSystem
signal Inform(real), ExecuteOrder(real), PayPenalty(real), GetPayment(real), Failure(real), Successful(real);
signal Complete(real), RefuseOffer(real), AssignOrder(real), NoOffer(real), Offer(real), NewOrder(real);
Bank_Block
[GetPaymant, PayPenalty]
ch2
Shuttle_Block
[Offer, NoOffer, Complete]
[NewOrder, AssignOrder, RefuseOffer, Successful, Failure]
ch1
Broker_Block
block Bank_Block
Bank
[GetPayment , PayPanelty]
sr4
ch2
block Broker_Block
BrokerAgent
[Inform]
sr6
BrokerAgentExt
[Offer, NoOffer]
sr3
[NewOrder, AssignOrder, RefuseOffer]
[Complete]
sr7
[Successful, Failure]
ch1
ch1

Figure 6.10: The SDL architecture for the Shuttle system

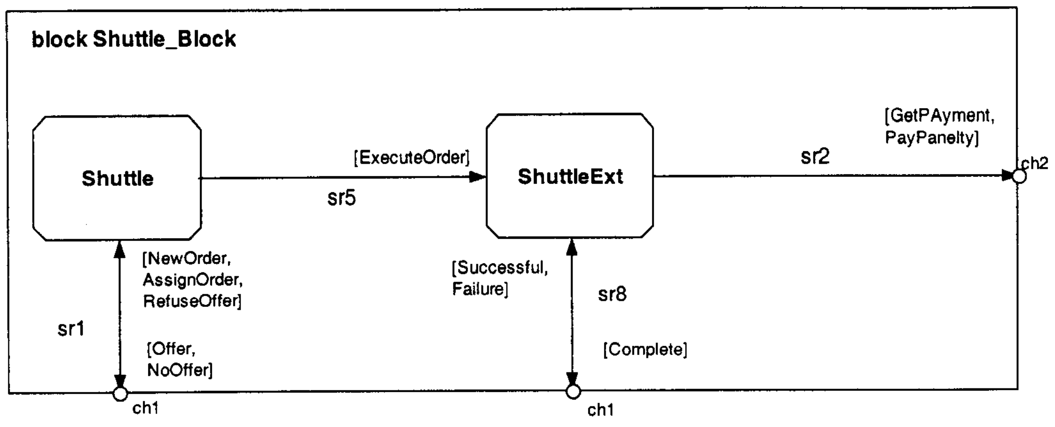


Figure 6.10: The SDL architecture for the shuttle system (continued)

SDL Generation

The generated process behaviors are shown from Figure 6.11 and Figure 6.15. In the MSC specification, no instance has process type. The tool checks the consistency between MSC specification and given SDL architecture by promoting all instances in the MSC specification as process types. Then they match the processes defined in the SDL architecture. There are 5 user defined variables, namely x's in instance Shuttle, ShuttleExt, BrokerAgent and BrokerAgentExt, and t in BrokerAgentExt. They are all declared in corresponding SDL processes. Since there are three blocks in the system, when two processes from different blocks pass massages, implicit addressing is used. However, explicit addressing is used for two processes in the same block.

Moreover, we can see that there are still redundant states in the process behaviors as those in the previous case study; and they can be removed with a refine step after SDL generation. We have also found some nondeterministic transitions, which come from the MSC specification. For example, state "ready" and "S2" in ShuttleExt both have two possible transitions and they are nondeterministic. ShuttleExt does not appear in any referenced bMSCs other than GET_ORDER_DELIVER. If we take the assumption that all processes should go with the control flow specified by an HMSC, then absent processes in some bMSCs are not controlled when at the alternative operators and our SDL generating algorithm generates nondeterministic behaviors for all process in the system. Therefore, this problem can be seen as a type of non-local choice. Furthermore, if we look at this case study carefully, we found that the referenced bMSC that process ShuttleExt only acts at the very end of the HMSC specification. Therefore, ShuttleExt does not really need to follow the control flow before GET_ORDER_DELIVER, which creates the nondeterministic behaviors. As a matter of fact, in this case, the nonderministic behaviors can be also be removed after SDL is generated. We see that in state "ready", ShuttleExt has a spontaneous transition to state S1, and only spontaneous transitions lead ShuttleExt from S1 to monitoring, therefore, this kind of "redundant" loop can also be detected and removed. Then the behavior of ShuttleExt starts with the SDL start construct, followed directly by state "monitoring". The same problem happens to process BrokerAgentExt and Bank.

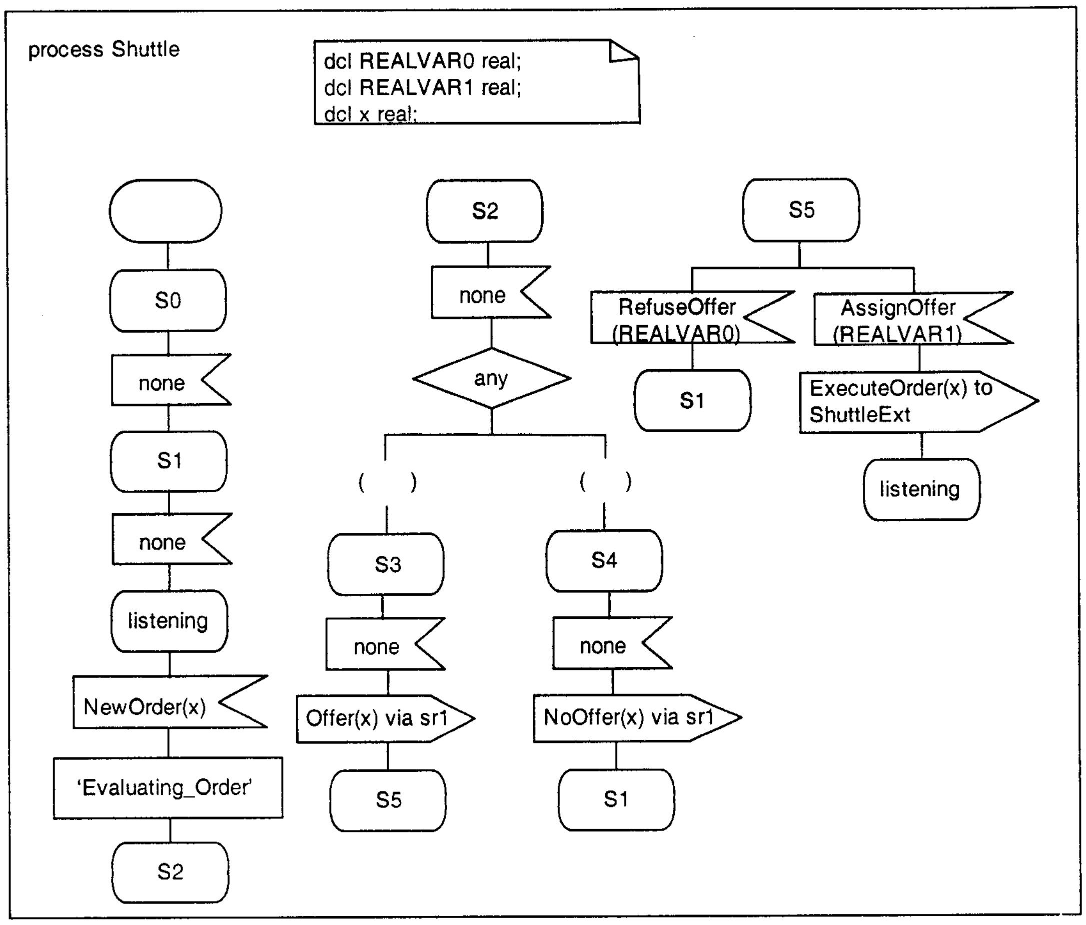


Figure 6.11: Process behavior for Shuttle

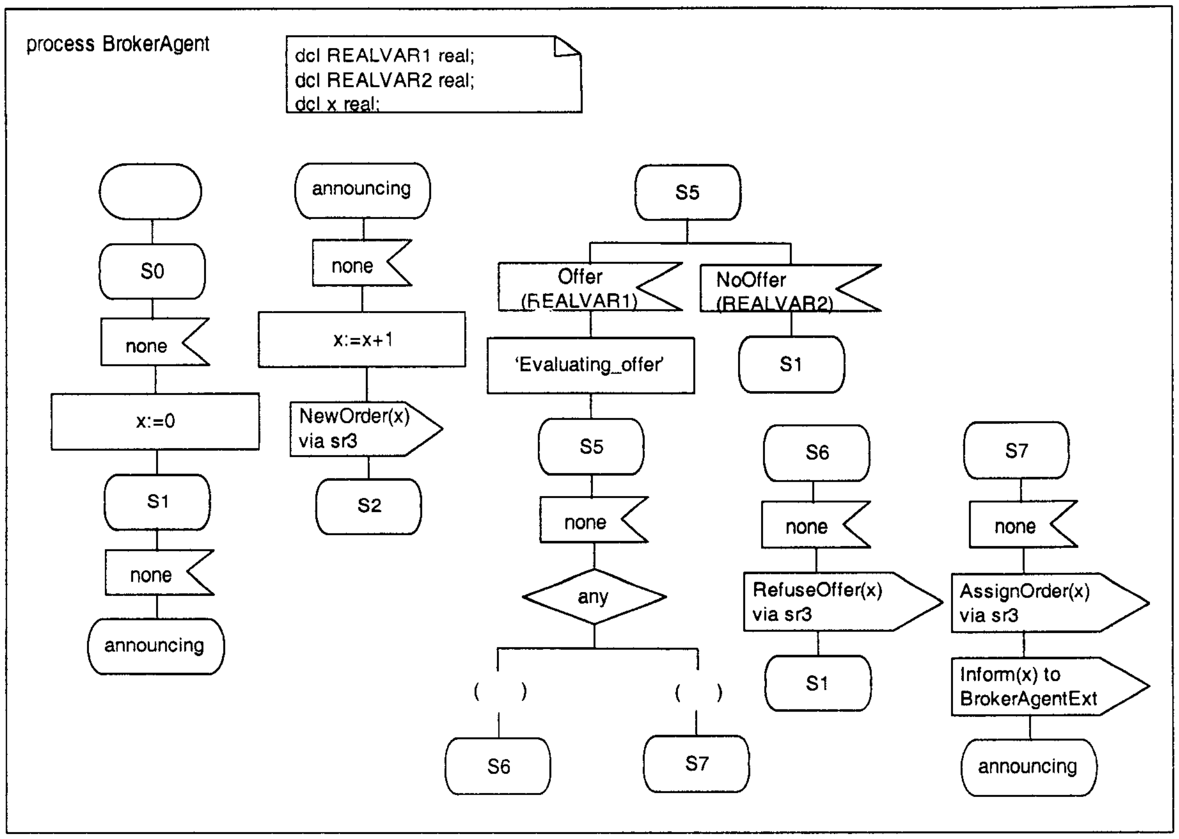


Figure 6.12: Process behavior for BrokerAgent

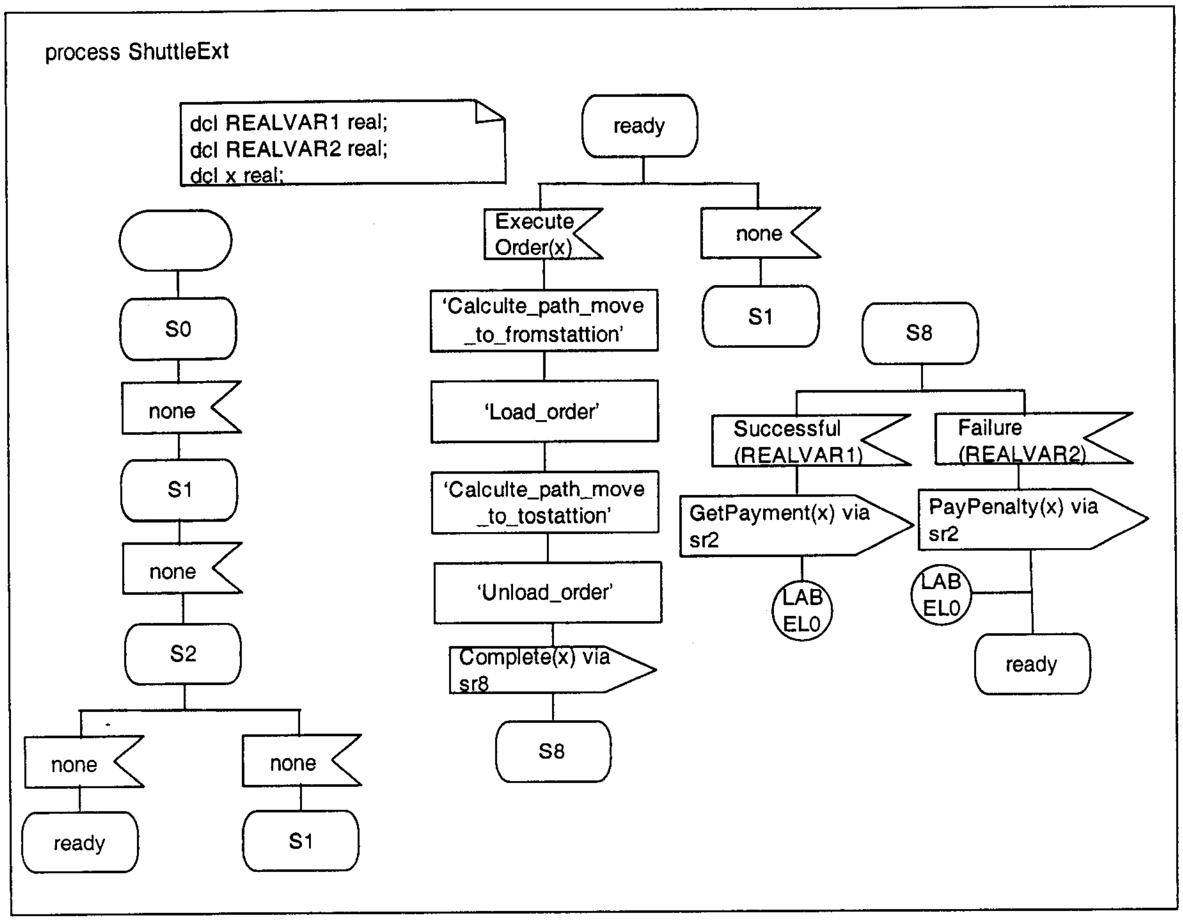


Figure 6.13: Process behavior for ShuttleExt

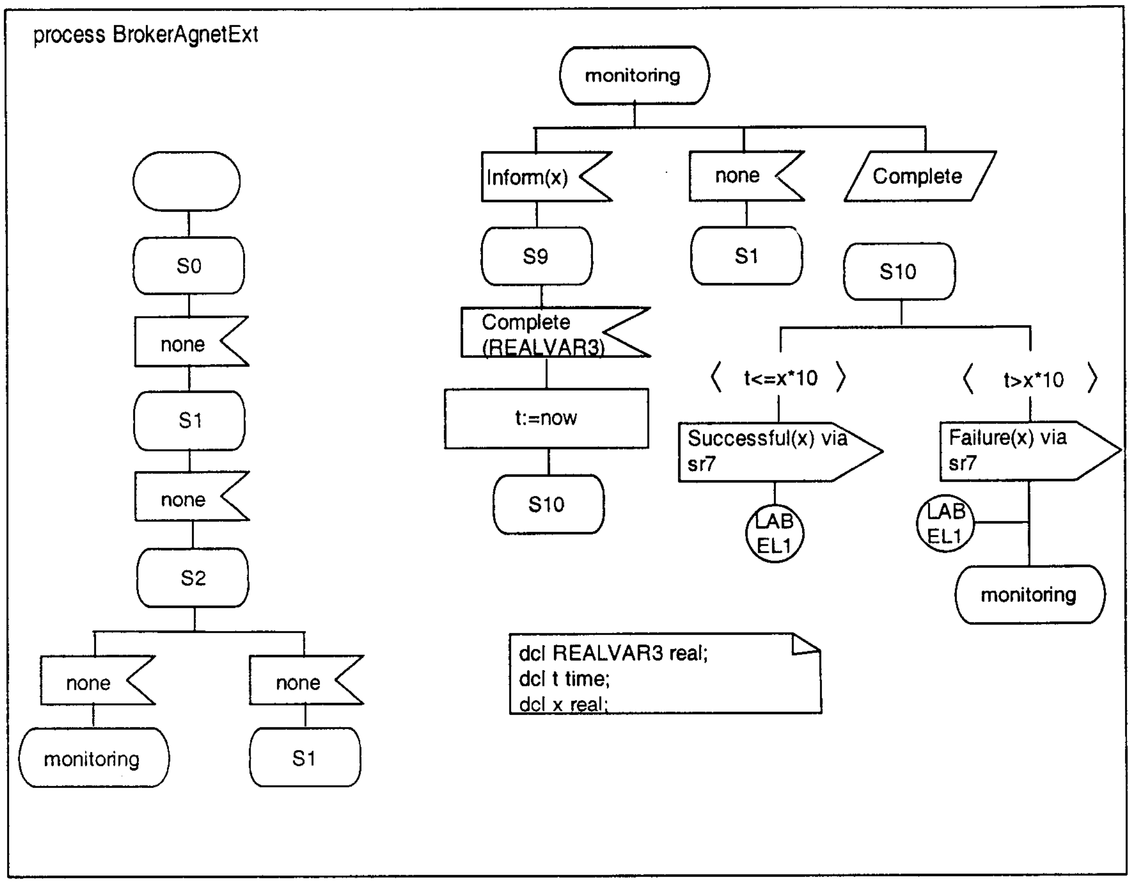


Figure 6.14: Process behavior for BrokerAgentExt

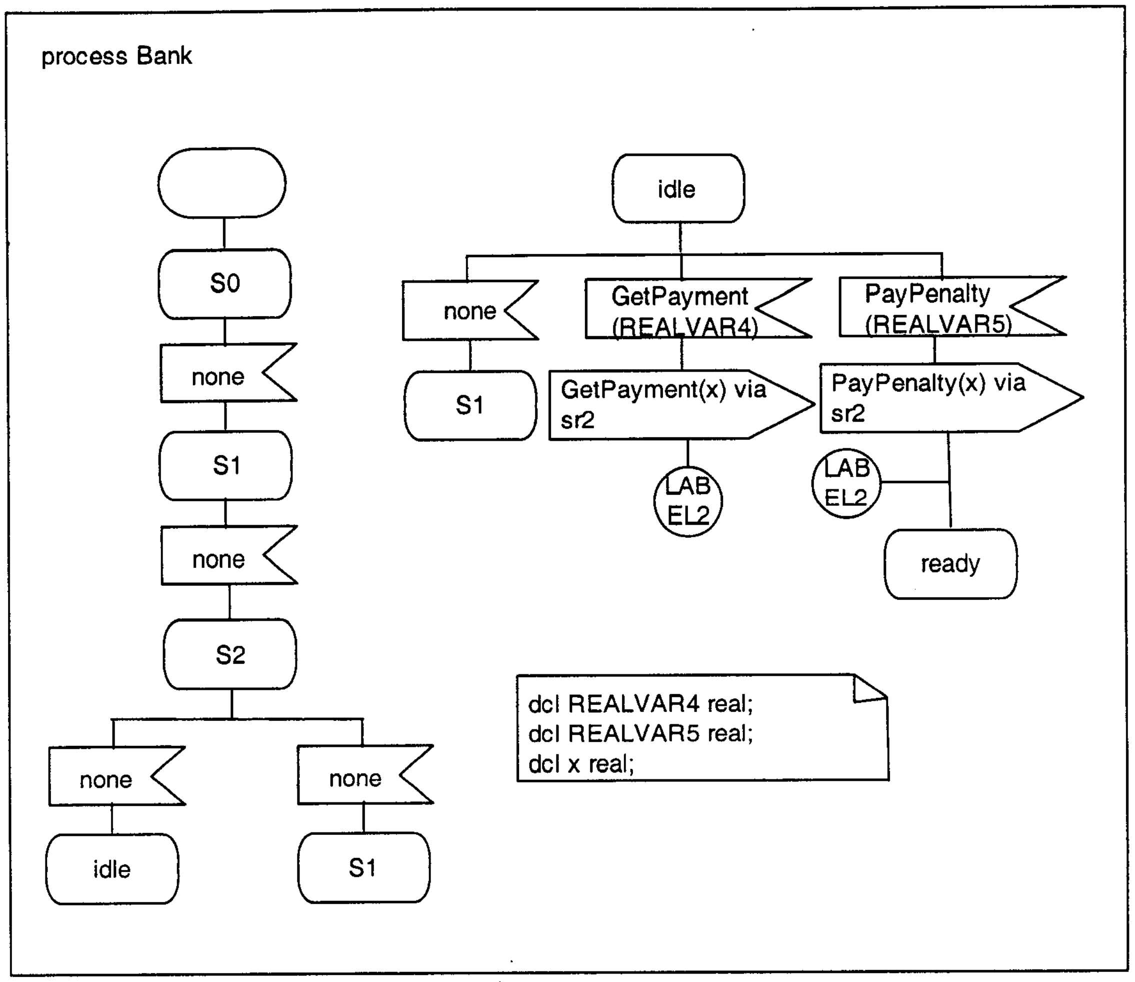


Figure 6.15: Process behavior for Bank

6.5.3 Discussion

Following the two case studies, we have seen two problems in our approach. First, our SDL generating algorithm may generate "redundant" state transitions for some HMSC specifications. Second, it may also generate "redundant" loop transitions for some HMSC specifications. We can classify them as the following. The "redundant" state transitions are those states transit to the next state spontaneously with no other "actions" than an input of **none**. A "redundant" loop may contain many "redundant" state transitions that form the loop with no other "actions". Both of them can be detected and removed. As a matter of fact, this is to minimize states for nondeterministic automata.

The fact that the "redundant" loop in process ShuttleExt in Shuttle System is redundant is due to the property of the MSC specification that the process appears only in one pat. of the HMSC. This fact also removes the potential non-local choice property. In the case that the process appears in more than one path and it is not aware of the decision at any alternative operator, non-local choice should be detected before generating SDL.

6.6 Strength and limitations of the Tool

The newly built tool MSC2SDL2004 includes the functionalities of the previous tool, and it adds functionalities to handle the translation of real-time system specifications. It also handles inline expressions (except **exc** and **par** operator) as well. The strengths and limitations as follows:

Strengths:

(1) The tool handles time constraints broadly and robustly.

(2) The tool checks various non-local choice scenarios in inline expressions.

(3) In an MSC specification, an instance is allowed not to have a process name. In this case, the instance name can be used as corresponding process name.

Limitations:

(1) The tool does not allow declarations in bMSCs, thus it requires an mscdocument to be included in the input MSC files.

(2) Time constraints associated with orderable events other than input/output is not implemented in this version.

(3) Message relay for conveying time constraints information is not implemented in this version.

(4) In bMSCs, no nested inline expression is allowed.

(5) This version does not handle multi-instances for one process.

(6) The Non-local choice detecting mechanism for HMSC is not fully implemented in this version.

Chapter 7

Conclusions and Future Work

7.1 Contributions

In this thesis we have developed on an existing framework for generating SDL design specifications from MSC specifications. Our contribution consists mainly in handling the time constructs of MSC-2000 for the translation to SDL.

For translation purpose, we have redesigned the Event Order Table. The Event Order Table keeps track of event order between each pair of events; and also records real time requirements specified in the MSC. Moreover, new order relation between a pair of events has been introduced to indicate events in different execution traces, so that we can handle MSC inline expressions and HMSC alternative operators efficiently. Signal extension has also been introduced in order to handle relative time requirements between events in different instances. Variablemap has been used to organize time variables and signal extensions for reference during the SDL generation phase.

We have defined the mapping from timed MSC to SDL, and devised an SDL generation algorithm. This algorithm handles all types of real time requirements that can be specified with MSC, such as time offsets, absolute time constraints and measurements, relative time constraints and measurements, timed inline expressions, and time constraints specified for referenced bMSCs in HMSCs. The SDL generation algorithm from HMSC has also been redesigned, so that it conforms to the MSC-2000 standard and capable of handling more complex specifications than the previous approach.

Our work includes the study of some semantic issues. We have devised a non-local choice detection algorithm for MSC inline expressions that contain MSC guarding conditions, message control and system architecture. For HMSC, the previous non-local choice detection algorithm has been extended to detect non-local choices caused by system architecture. Time related implementability of MSCs is addressed in our approach in order to make sure that time requirements in an MSC specification can be implemented in SDL. We also discussed translation related problems such as MSC conditions, and SDL procedures.

Finally, we have rebuilt the MSC2SDL2004 tool with our newly developed approach and experimented with case studies. The result has proved the feasibility of automatic synthesis of SDL design specifications from timed MSC requirement specifications.

Our work is the first one so far that addresses translation of timed MSC to SDL. It handles real-time requirements and addresses the related thorny issues.

7.2 Future work

Our approach is very promising for telecommunication software development; since it provides a systematic and automatic tool not only to make the transition from requirements to design much easier, but assure the quality of the product. Our approach and the tool can be further enhanced using an incremental development approach.

MSC structural concepts

Our approach so far handles a subset of MSC structural concepts. We believe more work can be done, such as allowing MSC references in bMSCs, HMSC references in HMSC, etc. Parallel operator of inline expression and HMSC are also an interesting extension. Furthermore, SDL-2000 has introduced exception and exception handler; MSC exc inline expression may be handled using these concepts.

External Data types

Currently our approach only allows MSC specifications to use limited data types and manipulations, such as integer, real, and their simple operations. However, complex systems need more advanced data types and operations that may be defined in other languages, such as C or ASN.1. Handing imported data definitions and functions in MSC will make our approach more powerful for complex specifications.

Multi-instance

The previous work handles multi-instances in untimed MSCs. We believe that more study should be done in order to handle this problem in our newly devised approach.

Non-local choice in HMSC

The non-local choice problem in HMSC is rather complex. It is very important to detect this problem before translating MSC into SDL. A more complete detecting approach may be devised and integrated into this work, so that our tool becomes more powerful in helping users to detect such kind of semantic problems.

Bibliography

[1] E. Rudolph, J. Grabowski and P. Graubmann, Tutorial on Message Sequence Charts (MSC'96), Proceedings of FORTE/PSTV'96 Conference, Kaiserslautern, Germany, 1996.

[2] G. Robert, F. Khendek and P. Grogono, Deriving an SDL Specification with a Given Architecture from a Set of MSCs. in A. Cavalli and A. Sarma (eds.), SDL97: Time for Testing - SDL, MSC and Trends, Proceedings of the eighth SDL Forum, Evry, France, Sept. 22 - 26, 1997.

[3] F. Khendek, G. Robert and G. Butler and P. Grogono, Implementability of Message Sequence Charts, Proceedings of the first SDL Forum Society Workshop on SDL and MSC, Berlin, Germany, June 29 - July 1, 1998.

[4] ITU-T, Specification and Description Language (SDL), International Telecommunications Union, Telecommunications Standards Sector (ITU-T), Recommendation Z.100, 1999.

[5] ITU-T, Message Sequence Chart (MSC), International Telecommunications Union, Telecommunications Standards Sector (ITU-T), Recommendation Z.120, 1999.

[6] T. Zheng, F. Khendek, Time Consistency of MSC-2000 Specifications, Computer Networks, Vol. 42, No. 3, 2003.

[7] M. M. Musa, Automatic Generation of SDL Specifications from MSCs, M.A.Sc Thesis, Concordia University, Montreal, Quebec, Canada, 1999.

[8] R. Detcher, I. Meiri, J. Pearl, Temporal constraint networks, Artificial Intelligence 49 (1991) 61-95.

[9] N. Mansurov and D.Zhukov, Automatic Synthesis of SDL models in Use Case Methodology, SDL'99: The Next Millennium, Proceeding of the ninth SDL Forum, Montreal, Quebec, Canada, June 21 –25, 1999.

[10] W. Dulz, S. Gruhl, L. Kerber and M. Söllner, Early Performance Prediction of SDL/MSC-specified Systems by Automated Synthetic Code Generation, SDL'99: The Next Millennium, Proceeding of the ninth SDL Forum, Montreal, Quebec, Canada, June 21 –25, 1999.

[11] J. Persson, MSC Transformations, MASC thesis, Department of Communication Systems, Lund Institute of Technology Instructor, Sweden, 1998.

[12] ObjectGeode, Telelogic, Toulouse, France, 2001.

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

[14] Formal Methods Group, University of Toronto, www.cs.toronto.edu/fm.

[15] J. Ellsberger, D. Hogrefe, A. Aarma, SDL Formal Object-oriented Language for Communicating Systems, Prentice Hall, 1997.

[16] ITU, Recommendation X.680 - Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation 2002.

[17] Software Engineering Group, Shuttle System Case Study, Version 1.0, University of Paderborn, 2004, http://wwwcs.upb.de/cs/ag-schaefer/CaseStudies/ShuttleSystem/.

[18] Telelogic, ObjectGeode version 4.0, Tutorial, 1999.

[19] S. Graf, Expression of time and duration constraints in SDL, Telecommunications and beyond: The Broader Applicability of SDL and MSC, Third International Workshop, SAM 2002, Aberystwyth, UK, June 24-26, 2002. Revised Papers.

[20] TAU SDL Suite, Telelogic, Sweden, 2004.

[21] ITU-T, The Tree and Tabular Combined Notation Version 3 (TTCN -3): Core language, International Telecommunications Union, Telecommunications Standards Sector (ITU-T), Recommendation Z.140, 2003.

Appendix A: The Complete Algorithm for bMSC

(1) **Check the architectural consistency between given SDL and MSC**

The following two steps ensure the MSC and given SDL architecture are consistent before generating process behavior.

- *For each process described in the MSC, there is a corresponding process type in the SDL architecture (If any MSC instance has no defined process type, the instance name is promoted as a process type name).*
- *Each message described in the MSC is enumerated in the SDL architecture by a route connecting the sending process and the receiving process.*

(2) **Number each input/output event uniquely**

Each event in the MSC specification is uniquely numbered, so that they can be distinguished in the translation process.

- *Each input/output event is assigned a unique number.*

(3) **Build the Event Order Table**

This algorithm builds the Enhanced Event Order Table. It takes three steps. First, time and order information are filled for each pair or individual events. For each event, check all time constraints and measurements, in which it involves, fill time information into the corresponding table cells. Then order information is obtained and filled using temporal order rules. Order specified by coregion and instance creations is also taken into account as well. Moreover, events in different operands of an alt or exc inline expression and events in opt inline expressions are identified and their orders are marked as described in the previous section; Second, the table is updated according to the transitive and reflective properties among events. Finally, for each inline expression that specifies with time requirements, using the order information to determine the staring or ending events of each operand. Fill the time requirements in the corresponding cells. If the starting event or ending events cannot be determined, that the MSC is not implementable is declared.

```
For each instance i in the bMSC
 For each input/output event e on the axis of instance i
   If absolute time constraints exist
     Insert it into the cell (e, e)
   Endif
   If relative time constraints with other events exist
     For each time constraint with event $e_i$
       Insert the time constraint information into the cell ($e_i$, e)
  Endfor
   Endif
   If e is in a coregion
     For each event $e_i$ with in the coregion
```

```
If the order between e and $e_i$ is not specified or e precedes $e_i$ by a general order
specification
            Mark cell (e, $e_i$) with 'T'
          Endif
      Endfor
   Endif
   If e is in an alt or exc inline expression
      For each $e_i$ in the same inline expression
          If $e_i$ is in a different operand
            Mark cell (e, $e_i$) with "E"
          Endif
          If $e_i$ is in same operand and following e
            Mark cell (e, $e_i$) with "T"
          Endif
      Endfor
   Endif
   If e is in an opt inline expression
      Mark cell (e, e) with "O"
   Endif
   For each event $e_i$ that is in the same instance and follows e
      Mark cell (e, $e_i$) with "T"
   Endfor
   If a create event $e_k$ exists following e
```

```
For each event $e_i$ in the created instance
            Mark cell $(e, e_i)$ with "T"
         Endfor
      Endif
      If e is an output event and its corresponding input event is $e_i$
            Mark cell $(e, e_i)$ with "T"
      Endif
  Endfor
Endfor
Repeat
For each event row e
  For each cell $(e, e_i)$ marked with 'T'
      Look for the event row $e_i$.
      For each cell $(e_i, e_k)$ marked with 'T'
         If cell $(e, e_k)$ is not marked with "T"
            Mark it with 'T'
            The Event Order Table has been changed.
         Endif
      Endfor
  Endfor
Endfor
Until there is no change in the Event Order Table entries
For each cell $(e, e_i)$ marked with "E"
```

```
 Mark the cell ($e_i, e$) with "E"
Endfor

For each inline expression ex
 For each requirement specified for ex
   If the time requirement is specifies with **start before** or **start after** event $e_k$
     If the inline expression is an alt
       For each alt operand i
         Look up the Event Order Table
           If the start event can be decided as $e_i$
             Insert the time information into cell ($e_i, e_k$) or ($e_k, e_i$)
           Else set NOT_IMPLEMENTABLE flag
           Endif
       Endfor
     Else Look up the Event Order Table
       If the start event can be decided as $e_i$
         Insert the time information into cell ($e_i, e_k$) or ($e_k, e_i$)
       Else set NOT_IMPLEMENTABLE flag
       Endif
     Endif
   Endif
   If any time constraints specified as **end before** or **end after** $e_k$
     If the inline expression is an alt
```

```
            For each alt operand i
                Look up the Event Order Table
                    If the end event can be decided as $e_i$
                        Insert the time information into cell ($e_i$, $e_k$) or ($e_k$, $e_i$)
                    Else set NOT_IMPLEMENTABLE flag
                    Endif
            Endfor
    Else Look up the Event Order Table
            If the end event can be decided as $e_i$
                Insert the time information into cell ($e_i$, $e_k$) or ($e_k$, $e_i$)
            Else set NOT_IMPLEMENTABLE flag
            Endif
        Endif
    Endif
    If time interval exists following the end of the inline expression
        If time interval specified with relative time
            If the inline expression is an alt
                For each alt operand i
                    Look up the Event Order Table
                    If the start and end event can be decided as ei and $e_k$
                        Insert the time information into cell ($e_i$, $e_k$)
                    Else set NOT_IMPLEMENTABLE flag
                    Endif
                    Endfor
                Else Look up Event Order Table
                    If the start and end event can be decided as $e_i$ and $e_k$
                        Insert the time information into cell ($e_i$, $e_k$)
                Else set NOT_IMPLEMENTABLE flag
                Endif
            Endif
            Else (time interval specified with absolute time)
                If it is a singular time
                    For every event $e_i$ in this inline expression
                        Insert the information into cell ($e_i$, $e_i$)
                    Endfor
                Else (bounded time)
                    Look up Event Order Table
                    If the start and end event can be decided as $e_i$ and $e_k$
                        Insert the time information into cell ($e_i$, $e_k$) and ($e_k$, $e_i$)
                    Else FLAG NOT_IMPLEMENTABLE
                    Endif
                Endif
            Endif
        Endif
    Endfor
Endfor
```

(4) **Check time related implementability**

This algorithm checks time related under-specifications and errors. The two events involved in any relative time constraint or measurement must be ordered. Moreover, there should be no conflict between any two absolute time constraints associated with the two ordered events.

```
For each Event Order Table cell with relative time constraint
  If the cell is not marked with "T"
     Terminate the translating process
  Endif
Endfor
For each pair of events $e_i$ and $e_k$ with absolute time constraints
  If ($e_i$ proceeds $e_k$) and (time value of $e_i$ is greater than that of $e_k$)
     Terminate the translating process
  Endif
Endfor
```

(5) **Check non_local choice**

This algorithm detects non-local choice. There are three steps. First, for each participating instances, a set is formed, in which operand numbers whose guarding conditions with **true** values are contained. If all sets contains only one common operand, non-local choice does not exists is declared. Otherwise, step 2 follows. The controller of the inline expression is going to be decided. If all operands have one common controller instance, then the controller for the inline expression exists. If the controller instance does not exist, non-local choice exists is declared. Otherwise, check whether the set for the controller instance is a subset of all other sets. If this is not the case, non-local choice exists is declared. Otherwise, if the inline expression is an alt inline expression, non-local choice does not exist. If the inline expression is an opt or loop, step 3 follows. In step 3, the channels that control messages travel are checked. If all control messages travel the same channel, non-local choice does not exist. Otherwise, declare that non-local choice exists.

```
For each inline expression
 For each participating instance
 Evaluate all guarding condition for each operand (if no guarding condition for
 one operand, it is considered having a value true)
     Form a set with all the operand No. with true guarding condition
 Endfor
 If each set contains only one element
   If all elements from all sets are common
     Declare non-local choice does not exist
   Else
     Declare non-local choice exists
   Endif
   Exit
 Else
   Find controller instance for this inline expression
```

```
If no controller instance found
    Declare non-local choice exists
    Exit
Else
    If the controller instance is not the instance whose set $s_i$ has the least
    elements
        Declare non-local choice exists
        Exit
Else
    If for all other sets $s_1$, $s_2$ ... $s_n$, not (($s_i \subseteq s_1$) and ($s_i \subseteq s_2$) and... and ($s_i \subseteq$
    $s_n$))
        Declare non-local choice exists
        Exit
Else
  If the inline expression is alt
    Declare non-local choice does not exist
    Exit
Else
    If not all control messages travel the same channel
        Declare non-local choice exists
            Exit
Else
    Declare non-local choice does not exists
```

```
      Endif
    Endif
  Endif
      Endif
        Endif
    Endif
```

- **Subroutine:** Find_the_controller_instance_for_an_inline_expression

```
For each instance i
  Controller_i = true
  For each operand
    If the first event does not precedes any other events in all instances in this
operand
      Controller_i = =false;
    Endif
  Endfor
  If Controller_i = = true
    Instance i is the controller instance
  Exit
Endfor
```

(6) **Create Coregion Tree**

This algorithm creates a coregion tree for each coregion in a bMSC that manifests all possible execution traces. It is same as the one in the previous work.

```
For each instance i in the bMSC
 If coregions exist
   For each coregion area C on i axis
     Extract events that precede all other events in C or have equal footing
     If only one event found
     Create the tree header node and put this event in the header
     CreateChildren (rest of events in C) /* a recursive function that
       extracts events that precede all others given or have equal footing and
       create child nodes*/
     Else
       Create an empty tree header node.
       CreateChildren (all events in C). /* a recursive function that extracts
       events that are precede all other given or have equal  footing and
       create child nodes*/
   Endif
  Endfor
 Endif
Endfor
```

(7) **Fill the Occupancy Tables**

This algorithm builds an occupancy table for each instance that has input events, indicating upon each signal consumption, the possible signals that can be in the process's input queue. The same condition is used as described in the existing approach, except that since we are handling inline expressions now, two messages passing in different operands of an inline expression won't enter each other's table row. In the following presentation, EΦEs indicates event E and Es are exclusive in the Enhanced Event Order Table.

```
For each instance i in the bMSC with message input
  For each input event $e_c$
    If $e_c$ is in a loop inline expression
      Unfold the loop once and build a new Event Order Table with unfolded
      loop events
    For each input event $e_r$ and its corresponding output event $e_s$
      If (NOT ($e_r << e_c$) AND NOT ($e_c << e_s$)) AND (NOT ($e_r \Phi e_c$) AND
      NOT ($e_c \Phi e_s$))
        Add message $e_r$ to row $e_c$ of the Occupancy Table
      Endif
    Endfor
  Endfor
  If a coregion exists in instance i
    Get the corresponding Coregion Tree
    Distribute the saved messages of the coregion events among the tree node.
  Endif
Endfor
```

Φ denotes the order relation (exclusive)

(8) **Build Variablemaps**

A Variablemap is generated for each time constraint or measurement. The Frontevent is always present, which is either the event associated with an absolute time constraint or measurement, or the first event of a relative time constraint or measurement. The Backevent is the second event in a relative time constraint or measurement. The FrontVariable is generated to record the occurrence time of the first event. If a signal extension is needed, then the Carryoutevent and Carryinevent are filled. The CarryVariable is also generated. The UserVariable is filled if exists. A StampVariable is generated when the second event of a relative time constraint is both the Carryinevent and the Backevent.

```
For each time constraint or time measurement in the Event Order Table
  Build a Variablemap
  Fill out the Frontevent
  Fill out the Backevent as needed
  Generate the time variables as needed
  Find the Carryoutevent and the Carryinevent as needed
Endfor
```

(9) **Generate the SDL code**

This algorithm generates process behaviors in SDL according to the types of MSC events and the mapping design for each type. First of all, absolute time constraint values are changed if a time offset for the MSC exists. Then, MSC events on each instance are scanned sequentially to generate SDL behaviors. There is a handler for each type of MSC events. These handlers generate necessary translate MSC events into correct SDL designs. A set of Flags are manipulated for each event to indicate any time constraints and time measurements related to it.

```
For each instance i in BMSC
 Generate an SDL start node
 If time offset exists
   Change all absolute time constraint values with addition of the time offset
 Endif
 For each event e on I instance
   Call Set_Flags
   Case type of event e
   Input event:
     Call Input_handler
   Output event:
     Call output_handler
   Start of coregion:
     Call Coregion_handler
   Start of alt inline expression:
     Call alt_handler
```

```
        Start of opt inline expression:
            Call opt_handler
        Start of exc inline expression:
            Call opt_handler
        Start of loop inline expression:
            Call opt_handler
        Set timer event:
            Call Set_timer_handler
        Time out event:
            Call Time_out_handler
        Otherwise:
            Call Otherwise_handler
        Endcase
    Endfor
Endfor
```

- **<u>Subroutine</u>:** Set_Flags

```
If e has a relative time constraints with preceding events in another instance
    Set flag R_P_REMOTE
Endif
If e has a relative time constraints with preceding events in same instance
    Set flag R_P_LOCAL;
Endif
```

```
If e has a relative time constraint or a time measurement with succeeding events
in another instance
  Set flag R_F_REMOTE
Endif
If e has a relative time constraint or a time measurement with succeeding events
in the same instance
  Set flag R_F_LOCAL
Endif
If E has a absolute time constraint
  Set flag ABS
Endif
If E has a time measurement as the second event
  Set flag MEA
Endif
```

- **Subroutine:** Input_handler

```
Generate an SDL state construct if needed
If the corresponding signal has been extended
  Generate an SDL time variable definition
Endif
Generate an SDL input construct
If (ABS) OR (R_P_LOCAL)
```

```
Generate an SDL conditional transition construct with the time constraints as the
condition
Endif
If there are messages assigned to be saved
  Generate an SDL save construct for each one
Endif
If a timer has been activated
  Generate an alterative SDL input construct for the time signal
Endif
If a message will be sent later to the sender
  Generate an SDL pid variable definition
  Generate an SDL task construct to Save the sender pid into the pid variable
Endif
If (R_P_REMOTE)
  Generate an SDL time variable definition


  Generate an SDL task construct to save now into the time variable
  Generate an SDL nextstate construct
  Generate an SDL transition construct with the time constraints as the condition
Endif
If (MEA)
  If now has been saved
```

```
Generate an SDL task construct to calculate time measurement use the value in
time variable
 Else
    Generate an SDL task construct to calculate time measurement with **now**
 Endif
Endif
If ((R_F_REMOTE) OR (R_F_LOCAL)) AND (**now** has not been saved)
 Generate an SDL time variable definition
 Generate an SDL task construct to save **now** into the time variable
Endif
```

- **<u>Subroutine</u>:** Output_handler

```
If (ABS) OR (R_P_LOCAL) OR (R_P_REMOTE)
 Generate an SDL nextstate construct
 Generate an SDL input construct with input **none**
Generate an SDL conditional transition construct with these time constraints as
the condition
Endif
If (MEA)
 Generate an SDL task construct to calculate time measurement with **now**
Endif
If the pid address of the destination is known
 Use the pid variable to send the signal
```

```
Else
  Use the name of the destination process or find a channel to send the signal
Endif
Generate an SDL output construct
If (R_F_REMOTE) OR (R_F_LOCAL)
  Generate an SDL time variable definition
  Generate an SDL task to save now into the time variable
Endif
```

- **Subroutine**: Coregion_handler

```
Rebuild the coregion tree
Translate the rebuilt coregion tree
Go to the end of the coregion
```

- **Subroutine**: Rebuild_coregion_tree

```
From root of the coregion tree
For all children of each node e
  If there are events other than input event or time out event AND there is more
  than one event
      Create an input event node input none and take this new node as child of e
      Take all events among E's children other than input events or time out events
      and their subtrees as subtrees of the new node
  Endif
Endfor
```

- **<u>Subroutine</u>**: Translate_the_rebuilt_coregion _tree

```
From root of the rebuilt coregion tree
For each level of the tree
  For each node
    If the node is an event
      Call Set_flags
      Case the type of event e
      Input event:
Call input handler (save necessary messages which specified in the coregion tree)
If the input event with an input none and there are more than one nodes as its
children
          Generate an SDL non-deterministic decision construct
        Endif
      Output event:
        Call output handler
      Set timer event:
        Call set timer handler
      Otherwise:
        Call otherwise handler
    Endif
  Endfor
Endfor
```

- **Subroutine**: Set_timer_handler

```
    Generate an SDL timer definition

    Generate an SDL set timer construct
```

- **Subroutine**: Time_out_handler

```
    Generate an SDL input construct
```

- **Subroutine**: Otherwise_handler

  This handler is to handle events having a type that is not enumerated.

- **Subroutine**: Alt_handler

```
Generate an SDL state construct T

Name a label N

For each of the operand S in alt inline expression

  If guards for S exists

      Generate an SDL conditional transition decision with guards as condition.

  Else

      Flag NO_GUARDS

  Endif

  For each of the event e in S

      Case the type of event e

      Input event:

          Call Set_Flags

          Call Input handler
```

```
        Output event:
            If NO_GUARDS and E is the first event in S
                Generate an SDL input construct with input **none**
            Endif
            Call Set_Flags
            Call Output_handler
        Start of coregion:
            Call Coregion_handler
        Set timer event:
            If NO_GUARDS and E is the first event in S
                Generate an SDL input construct with input **none**
            Endif
            Call Set_timer_handler
        Time out event:
            Call Time_out_handler
        Otherwise:
            Call Otherwise_handler
        Endcase
    Endfor
    If S is not the last operand in alt inline expression
        Generate an SDL join construct with label N
    Else
    Generate an SDL connection construct with label N
  Endif
Endfor
```

- **<u>Subroutine</u>**: Opt_handler

```
Generate an SDL state construct T
Name a label N
For the opt
  If guards for S exists
    Generate an SDL conditional transition decision with guards as condition.
  Else
    Flag NO_GUARDS
  Endif
  For each of the event e in S
    Case the type of event e
    Input event:
      Call Set_Flags
      Call Input handler
    Output event:
      If NO_GUARDS and E is the first event in S
        Generate an SDL input construct with input none
      Endif
      Call Set_Flags
      Output_handler
      Start of coregion:
        Call Coregion_handler
      Set timer event:
        If NO_GUARDS and E is the first event in S
          Generate an SDL input construct with input **none**
        Endif
        Call Set_timer_handler
      Time out event:
        Call Time_out_handler
      Otherwise:
        Call Otherwise_handler
      Endcase
  Endfor
Endfor
Generate an SDL join construct with label N
Endfor
(Generate an SDL connection construct in the other branch with label N, when
process the events following the opt inline expression)
```

- **<u>Subroutine</u>**: Exc_handler

```
Generate an SDL state construct
For the exc
  If guards for S exists
      Generate an SDL conditional transition decision with guards as condition.
```

```
Else
    Flag NO_GUARDS
Endif
For each of the event e in S
    Case (E type) of event e
    Input event:
        Call Set_Flags
        Flag NO_STATE
        Call Input handler
    Output event:
        If NO_GUARDS and E is the first event in S
            Generate an SDL input construct with input none
        Endif
        Call Set_Flags
        Output_handler
    Start of coregion:
        Call Coregion_handler
    Set timer event:
        If NO_GUARDS and E is the first event in S
            Generate an SDL input construct with input none
        Endif
        Call Set_timer_handler
    Time out event:
            Call Time_out_handler
         Otherwise:
            Call Otherwise_handler
         Endcase
      Endfor
   Endfor
   Generate an SDL stop construct
   (Continue translation following state T for the rest of exc inline expression)
```

- **Subroutine**: Loop_handler

```
Name two labels N1, N2.
Generate an SDL variable definition
Generate an SDL task construct (for the upperbound)
Generate an SDL variable definition
Generate an SDL task construct (for the lowerbound)
Generate an SDL state construct
Generate an SDL input construct with input none
Generate an SDL decision construct
Generate an SDL join construct with label N1 follow the true answer construct
Following the false answer construct
Generate an SDL variable definition
Generate an SDL task construct (initialize the loop counter)
Generate an SDL state construct s
Following the state s
```

```
{
Generate an SDL conditional transition construct taking loop boundaries and
guard (if exist) as the condition
  Generate an SDL connection construct with label N2
     For each event e in the loop
        Case the type of event e
        Input event:
           Call Set_Flags
           Call Input handler
        Output event:
           Call Set_Flags
           Output_handler
        Start of coregion:
           Call Coregion_handler
        Set timer event:
           Call Set_timer_handler
        Time out event:
           Call Time_out_handler
        Otherwise:
           Call Otherwise_handler
        Endcase
     Endfor
  Generate an SDL task construct to increment counter
  Generate an SDL nextstate construct
}
{
Generate an SDL conditional transition construct taking loop upper boundary
and counter as the condition
  Generate an SDL connection construct with label N1
}
{
Generate an SDL conditional transition construct taking loop boundaries, counter
and guard (if exist) as the condition
  Generate an SDL decision construct with condition any
  Generate an SDL join with construct Label N1 following one answer construct
  Generate an SDL join with construct Label N2 following another answer
construct
}
{
Generate an SDL conditional transition construct taking loop boundaries, counter
and guard (if exist) as the condition
  Generate an SDL join construct with label N1
}
```

Appendix B: The Complete Algorithm for HMSC

(1) **Check the architectural consistency between given SDL and MSC**

The following two steps ensure the HMSC and given SDL architecture are consistent before generating process behavior.

- For each process in each referenced bMSC, there is a corresponding process type in the SDL architecture.

- For each message described in each referenced bMSCs is enumerated in the SDL architecture by a route connecting the sending process and the receiving process.

(2) **Number each input/output event uniquely**

Each event in the HMSC specification is uniquely numbered, so that they can be distinguished in the translation process.

- Each input/output event is assigned a unique number.

(3) **Build the Event Order Table**

- Build individual Event Order Tables for all referenced bMSCs.

- Calculate all pair shortest path for HMSC graph

- Combine individual Event Order Tables into a Global Event Order Table according to the relationship among the bMSCs.

(4) **Create Coregion Trees**

Same as described for bMSC

(5) **Build Occupancy Tables**

Same as described for bMSC, except that the inline operators alt, opt are replaced by the corresponding HMSC operators. When loop operators exist, unfold the loop and build an Event Order Table with the unfolded loop events.

(6) **Build Variable Maps**

Same as described for bMSC.

(7) **Generate SDL code**

This algorithm generates process behaviors in SDL from an HMSC specification. The first step is to reserve state names for each HMSC node. Then, the HMSC graph is first search manner traversed and process behavior is generated from all referenced bMSCs.

```
Reserve state name for each HMSC node
Generate SDL from HMSC
```

- **Subroutine**: *Reserve_statename*

```
For each connect
    Search for local condition in connected bMSCs
 If found
    Use this name as the state name for this instance
 Else
    Reserve a state name
```

```
      Endif
    Endfor
  For each Label which represents a BSMC
    Generate a state name for this BMSC
  Endfor
```

- **Subroutine**: *Generate_SDL*

```
Generate an SDL start construct
Traverse the graph from the start node (BFS):
{
If there is only adjacent node
   If the node is a guarding condition
        Generate an SDL nextstate construct
        Generate an SDL state construct
        Generate an SDL enabling transition construct

   Else if the node is a setting condition
    Generate an SDL nextstate construct
    Generate an SDL endstate construct (if needed)
    Generate an SDL state construct

   Else if the node is a connect
        Generate a state using its reserved state name
        Continue to traverse the graph

      Else if the node is an bMSC
        Generate an SDL nextstate construct using the its state name
        Generate an SDL endstate construct to end above state (if needed)
        Generate an SDL state construct
        Call BMSC SDL generating procedure

      Else if the node is the end node
        Generate an SDL end construct

    Else (there are more than one adjacent node)
      If the instance is the controller instance
        Generate an input "none"
        Generate a decision construct with "any"
        Generate an empty answer construct
      Else
        Continue to traverse the graph
      Endif
    Endif
    Continue to traverse the graph
}
```