INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600

AUTOMATIC GENERATION OF SDL SPECIFICATIONS FROM MSCs

Mohamed Musa Abdalla

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements for the
Degree of Master of Applied Science
at
Concordia University
Montreal, Quebec, Canada

November 1999

ABSTRACT

AUTOMATIC GENERATION OF SDL SPECIFICATIONS FROM MSCs

Mohamed Musa Abdalla

Software systems go through different phases during their life cycle. From user requirements to the deployment and maintenance, a software system goes through design, implementation and testing phases. The time for developing software systems is crucial. The goal of software teams is to shorten the development time and guarantee the quality of the end product. Formal Description Techniques (FDTs) has been used in the development cycle to provide clear, correct and unambiguous specifications throughout the phases in order to achieve this goal.

In order to speed up the development cycle and to guarantee the correctness of the design, we devise and implement an approach for generating SDL (Specification and Description Language) design specifications from requirement specifications given as a set of MSCs (Message Sequence Charts) and a target SDL architecture. Our approach handles MSC'96, except the parallel operators.

Our approach bridges the gap between requirements and design and guarantees the quality of the design. The generated SDL design specification is free of any design error, such as deadlocks or unspecified receptions, and conforms to the MSC requirements specification.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis supervisor, Professor F. Khendek, whose continued support, guidance, encouragement and constructive criticism have been of considerable help during the course of this work.

Words fall short when I attempt to express my love and gratitude to my parents, Musa and Zayinap, and to my wife, Fatma. This thesis would have never been completed without their love, encouragement and moral support.

I would like to acknowledge the financial support of the Ministry of Education in my country, Libya, and from Professor F. Khendek's FRDP Start-Up grant.

Lastly but not the least, I would like to thank all my friends at Concordia University and in my country for sharing with me a challenging experience.

To my parents, my wife and
my beloved daughter
Zayinap

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 SOFTWARE PROCESS

The software process goes through several phases. As shown in Figure 1.1, it starts with the requirement phase where the analysts have to interact with the user/consumer, and identify and recognize the user needs and clarify any ambiguity in the requirements. In this phase, formal or semi-formal documentation of the requirements in a high-level of abstraction is produced for the next phase. Furthermore, test cases will be generated from these requirements for testing the final product or the prototype. The output of the requirement phase feeds, as input, the design phase. In the design phase, the designers start the system development with an abstract specification of the design, (e.g., overall architecture). More design details will be gradually introduced into the specification. In the implementation phase, programmers implement the design specification provided by the designers in a target platform. Finally, before deployment, the product is tested against the requirements using the generated test cases.

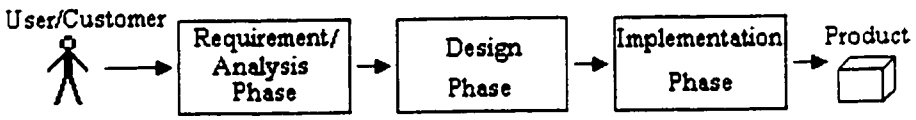


Figure 1.1. Software process.

Many software processes, such as Waterfall and Spiral models [1], have been developed to reduce the development time and to ensure the consistency among the software process phases.

1.2 TELECOMMUNICATIONS SOFTWARE

Nowadays, the global telecommunications network is the largest software system in the world. The development of telecommunications and distributed software systems is more complicated than general purpose software systems since it consists of independent components running in parallel with different speeds and environments, and developed by different teams.

Computers and software are playing major roles in telecommunications systems. These software systems have to be reliable and easy to maintain. The development of telecommunications systems requires powerful tools that can capture clear system specifications and help developers throughout the development cycle.

1.3 FORMAL DESCRIPTION TECHNIQUES (FDTs)

Using natural languages in the software process may lead into ambiguous and incomplete specifications. The increasing of size and complexity of the developed systems urged the need for more precise, expressive and unambiguous languages.

To overcome the ambiguity and incompleteness of the "informal" tools/documentation involved in the software process, stronger and more expressive techniques, based on mathematics, such as Z [13], SDL [8], MSC [10] and UML [14] were introduced to improve the quality of specification and design.

The main goal of FDTs is to produce formal specification that is unambiguous, clear and concise. FDTs are also intended to provide a foundation for analysis, simulation, validation and testing of specifications for correctness. Furthermore, using FDTs facilitates maintenance and reusability, as well as improves the quality of the product.

1.4 SDL AND MSC

SDL, Specification and Description Language, is a standard formal language that has been widely adopted for the description of telecommunications systems. The International Telecommunications Union (ITU) has standardized SDL in the Recommendation Z.100 [8]. The SDL language has graphical and textual representations. Several commercial tools support SDL and provide validation and simulation features for SDL specifications as well as code generation for the most popular implementation languages as C++ and Java and popular platforms. In the same way, MSC is a standard formal language widely adopted. It is used to capture the requirements of telecommunications systems. It has been standardized by ITU in its Recommendation Z.120 [10]. It also has graphical and textual representations.

SDL and MSC are compatible languages that complement each other. In addition to capturing the requirements, MSC is used to generate test cases for SDL specification validation or end product testing, as well as to visualize simulation traces.

1.5 GOAL OF THE THESIS

MSCs combined with SDL has been used by telecommunications software system developers to capture system requirements and design specifications, respectively. In a

typical software process, as shown in Figure 1.2, MSC is used to capture the system requirements at the requirement phase, while at the design phase, SDL is used to describe the design.

With the increasing complexity of telecommunications systems, manual translation from MSC specification (requirement phase) into SDL specification (design phase) will be time consuming and subject to human errors. In addition to syntax errors, semantic errors such as deadlocks may be produced by manual translation. Providing an automatic translation tool saves time and eases the engineers' task.

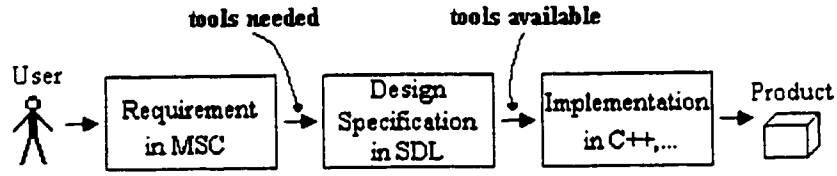


Figure 1.2. MSC and SDL in the software process.

In addition to the use of formal techniques, a full or partial automation of the software process will have a major improvement for telecommunications systems development, and will lead to better quality product. Several approaches have been presented to move automatically from an SDL design specification to an implementation. Many tools that generate source code from SDL specifications have been introduced such as ObjectGEODE [17] and SDT [18]. Our goal is to bridge the gap between the requirement phase and the design phase. In order to reach our goal, we take MSC as a requirement specification language, and SDL as the design specification language. Through two years of work, we have investigated the translation of MSC specifications to SDL specifications. In this document, we present an approach for automatic translation of MSC specifications to SDL specifications with a given target architecture.

More precisely, we have extended an existing approach introduced by G. Robert and F. Khendek [4] to cover MSC timers, coregions, inline expressions, message overtaking and HMSC operators. We also handle MSC specifications with multi-instances of the same process. In addition, we have addressed the inconsistency between MSC instance identifications and SDL instance identifications. Furthermore, we provide MSC semantic error detection algorithms, which detect deadlocks, process divergences, non-local choices, unspecified receptions and the implementability of a given MSC specification under the target SDL architecture.

1.6 ORGANIZATION OF THE THESIS

The rest of this document is structured as follows:

Chapter 2 gives an overview of the MSC language.

Chapter 3 introduces the SDL language.

Chapter 4 is devoted to the basic translation approach, which was introduced in [4]. It illustrates the translation of basic Message Sequence Charts (bMSCs) that contain instances and messages into SDL specifications. It also presents our extensions to the basic approach by including timers, instance creations, inline expressions, coregion as well as the consistency problem between MSC and SDL specifications and its verification.

Chapter 5 introduces our expansion of the basic approach to cover most of the MSC concepts. Actually, it covers HMSC and multi-instance specifications. We will also discuss the semantic problems, as non-

local choice and process divergence, which may result from the introduction of HMSC and our techniques to detect them. Later in this chapter, the implementability of MSC specifications under the target SDL architecture is also discussed and a verification algorithm is described.

Chapter 6 gives an overview of the MSC2SDL tool. MSC2SDL implements our translation approach. We also provide in this chapter real cases that have been conducted with MSC2SDL tool.

Chapter 7 summarizes the results of our research and provides suggestions for further research and investigations.

CHAPTER 2

MESSAGE SEQUENCE CHARTS (MSCs)

2.1 INTRODUCTION

MSC is a graphical language, which describes the system behavior in terms of sequences of message exchanges among the system entities and its environment. MSCs had been used for a long time before they were standardized by International Telecommunications Union (ITU) in Recommendation Z.120. Since then, MSCs have been widely adopted within many software methodologies and tools. MSCs gain vast popularity for their intuition for both engineers and designers as well as their well-defined semantics.

MSCs basically describe the message exchange between the system processes (instances), and abstract out data and internal computations. Communication is asynchronous. In the Z.120 recommendation, the system processes are called instances, and to avoid any conflict, this document follows the Z.120 notation.

Nowadays, there are many tools, (e.g., ObjectGEODE and SDT,) that support MSC to

- Capture system requirements
- Describe test cases
- Express system properties
- Visualize simulation traces

This chapter describes briefly the syntax and semantics of a subset of basic and high-level MSCs that we use throughout the report. For more details, reader can refer to [10] and [3].

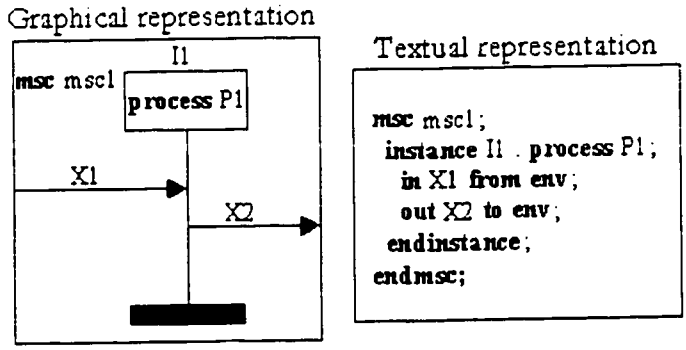


Figure 2.1. MSC graphical and textual representations.

The MSC has two representation forms: textual (MSC/PR) and graphical (MSC/GR). The mapping between the two representations is not linear because the textual representation does not have graphical information. Figure 2.1 illustrates both forms. From now on, this report will focus on MSC/GR for illustration purpose. In the latest revision (MSC'96), Recommendation Z.120 defines two main concepts: Basic Message Sequence Charts (bMSCs) and High-level Message Sequence Charts (HMSCs).

2.2 Basic Message Sequence Charts (bMSCs)

A MSC is composed essentially of a set of parallel process instances that exchange messages in an asynchronous mode. In addition to message exchanges, Z.120 allows a bMSC to contain conditions, which describe the state of a subset of processes in the MSC, actions, timers and instance instantiation and termination.

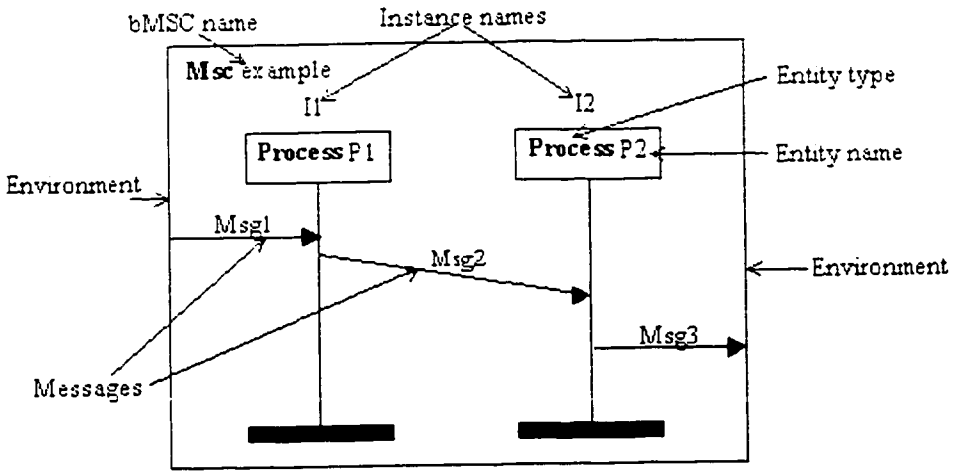


Figure 2.2. Essential bMSC constructs.

2.2.1 ENVIRONMENT

Within a bMSC, the system environment (env) is represented by a frame symbol that forms the boundary of the bMSC diagram, as shown in Figure 2.2. There is no assumed ordering among the environment events (sending/consuming). However, it is assumed that the environment behaves according to the bMSC specification.

2.2.2 INSTANCES

Instances, along with the environment, play the actor role in MSCs. They communicate by exchanging messages in one-to-one basis. Instances are depicted by vertical lines with time progressing downwards, as shown in Figure 2.2. A rectangle on the top of the vertical line denotes the instance head, which determines the start of the description of the instance within the bMSC, but does not describe the creation of the instance. At the other end of the vertical line, a filled rectangle donates the instance end. The instance end determines the end of the description of the instance within this bMSC, but does not describe the termination of the instance. Each instance has a unique name in the complete MSC specification. Within the instance heading an entity name, (e.g. process name,) may

be specified in addition to the instance name. In addition to exchange messages, instances can individually execute internal actions, use timers to express timing constraints, create instances and terminate themselves. Within each instance, events are totally ordered according to their positions from the start to the end symbols on the instance axis. Along each instance axis, the time is running from top to bottom, though, a proper time scale is not assumed.

2.2.3 MESSAGES

MSC instances communicate with each other through messages. These messages are represented by horizontal or slopping arrows associated with messages names, as shown in Figure 2.2. At least one of the ends should be attached to an instance axis. The tail of the message arrow denotes the sending event, while the head of the message arrow denotes the consumption event. For each message, its sending event precedes its consumption event in order. Nevertheless, MSC does not describe the actual receiving events.

2.2.4 ACTIONS

In addition to messages, an instance can include actions, which describe internal activities of the instance. An action may contain assignments, expressions and/or text separated by semicolons. It is represented by a rectangle containing informal text, as illustrated in Figure 2.3 where instance *I1* consumes message *i*, and executes internal assignment "j=i*i", then sends the result *j* to the environment.

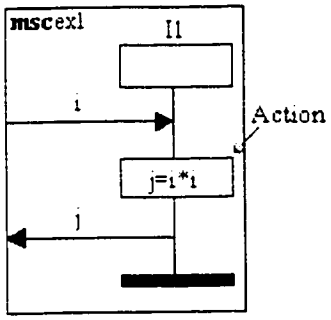


Figure 2.3. MSC action.

2.2.5 CONDITIONS

Conditions are used to emphasize important states within a bMSC. They are used for documentation purposes. A condition can be

- A global condition, which describes a global system state referring to all instances within bMSC: such as *condition1* in Figure 2.4. Global conditions can be used to restrict composition of bMSCs.
- A non-global condition, which describes a state referring to a subset of instances within bMSC: such as *condition2* in Figure 2.4.
- Or a local condition, which describes a private state referring to a single instance within bMSC: such as *condition3* in Figure 2.4.

Conditions are represented by hexagons covering the instances involved.

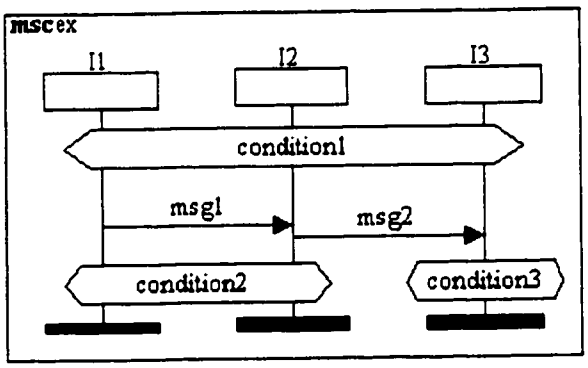


Figure 2.4. MSC conditions.

2.2.6 INSTANCE TERMINATIONS

An instance can terminate itself by executing a stop event. Execution of a stop event is permitted only as the last event in the description of an instance. The stop symbol is presented by a cross at the end of the instance axis: as shown in Figure 2.5.

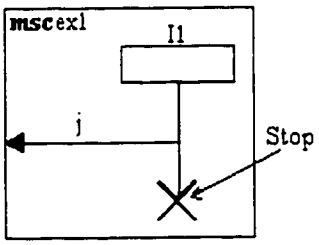


Figure 2.5. MSC instance termination.

2.2.7 INSTANCE CREATIONS

Instances can be created dynamically by another instances. As soon as the instance has been created, it runs, independently, in parallel to its parent. The created event is represented by a dashed arrow, see Figure 2.6. A created event arrow originates from a parent instance and points at the instance head of the child instance.

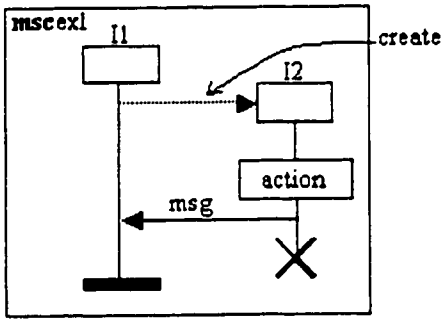


Figure 2.6. MSC instance creation.

2.2.8 INLINE EXPRESSIONS

Inline operator expressions define events composition inside bMSCs. The operators refer to parallel, alternative composition, iteration, exception and optional regions.

A parallel inline expression defines the parallel execution of bMSC. No ordering is preserved between events in different sections. In Figure 2.7.a, sending and reception of message *C* executes in parallel to the sending and reception of messages *B* and *D*.

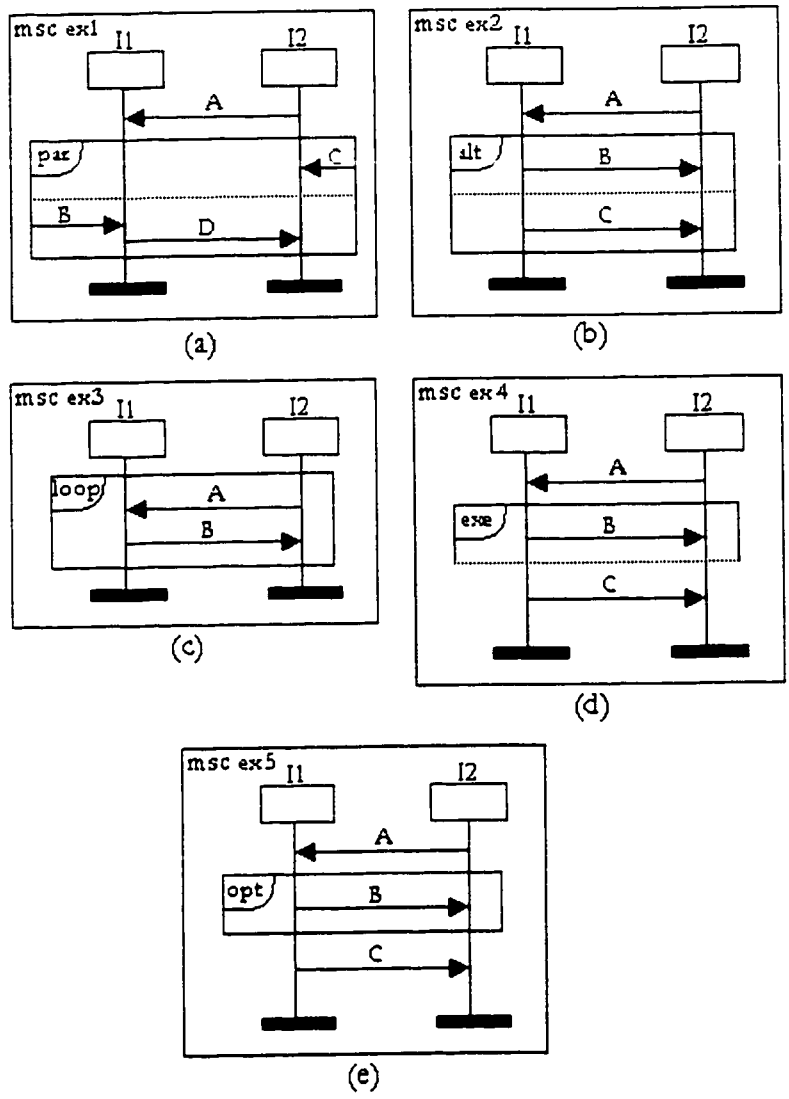


Figure 2.7. MSC inline expressions.

An alternative inline expression defines alternative executions of bMSC sections. Only one section will be executed for each execution trace. In Figure 2.7.b, after consuming message *A*, instance *I1* will send either message *B* or message *C* to instance *I2*.

An iteration inline expression defines iteration execution of bMSC section. Events in the iteration area will be executed many times (0-inifinite). In Figure 2.7.c, instances *I1* and *I2* will exchange messages *A* and *B* at least once.

An exception inline expression defines exceptional cases in bMSC. Either the specification in the exception area will be executed or the reset of MSC specification will be executed. In Figure 2.7.d, instance *I1* will send message *B* or message *C*, after consuming message *A*.

An optional inline expression defines an optional execution of bMSC section. Events in optional area may or may not be executed. In Figure 2.7.e, after consuming message *A*, instance *I1* may send message *B* before sending message *C*.

2.2.9 TIMER EVENTS

Timers may be defined within bMSC to express timing constraints (timer expiration and time supervision). There is no global time axis for MSCs. Along each instance axis, the time is running from top to bottom, though, a proper time scale is not assumed. Figure 2.8 shows time events that can be used within MSCs:

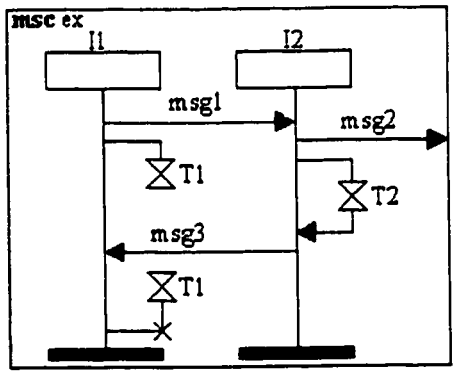


Figure 2.8. MSC timer events.

- Set event is represented by an hourglass connected with the instance axis by a (bent) line symbol. A timer set event denotes setting the timer.

- Reset event is represented by a cross connected with the instance axis by a (bent) line symbol. A reset timer event denotes resetting the timer.

- Time-out event is represented by an arrow, which is connected to the hourglass symbol, and the arrowhead pointed to the instance axis. A time-out event denotes expiration of the timer.

For each timer setting event, a corresponding time-out or/and timer reset has to be specified and has to follow it in order. However, corresponding timer events may be split among different bMSCs in cases where the whole scenario is obtained from the composition of several bMSCs

2.2.10 GENERAL ORDER

General Order relations describe orders between events which otherwise would not be ordered. They can prescribe the order relations, in time, between events within coregions. General Order is presented by connecting lines, Figure 2.9, between order events where the upper end of the line defines the preceding event.

2.2.11 COREGION

Events along each MSC instance are totally ordered. Coregions are introduced to relax the order in some parts of the instance axis. Within a coregion, the specified communication events are not ordered. However, the order between the corresponding timer events is preserved within coregions.

Furthermore, events can be ordered within coregions by prescribing an order relations between them in a form of general order relations. Coregions may be used to describe events, whose order is not specified yet and will be determined in the next phases, (e.g., design phase or implementation phase.) A coregion is represented by a dashed section of

an instance. In Figure 2.9, message *msg1* precedes both messages *msg2* and *msg4*, but there is no order relation between messages *msg2* and *msg4*.

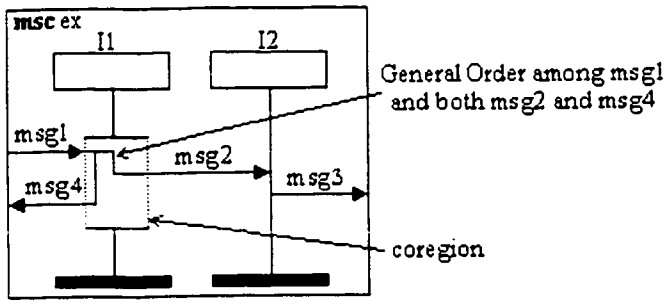


Figure 2.9. MSC coregion.

2.3 HIGH-LEVEL MESSAGE SEQUENCE CHARTS (HMSCs)

High-level Message Sequence Charts (HMSCs) give an overview of the complete system specification in terms of the composed bMSCs. HMSCs provide four operators to connect bMSCs to describe sequential, alternative, iterating and parallel execution of basic MSCs. In addition, HMSCs can describe a system in a hierarchical manner by combining HMSCs within a HMSC. Global conditions in HMSCs represent global system states.

Sequential HMSC operator defines the sequential execution of several bMSCs. The bMSCs will be executed one by one in the order that has been specified within HMSC. In Figure 2.10.a, bMSC *bMSC1* will be executed before bMSC *bMSC2*.

Alternative HMSC operator defines alternative executions of bMSCs. Only one of the alternative bMSCs will be executed for each execution trace. In Figure 2.10.b, either bMSC *bMSC1* or *bMSC2* will be executed.

Iteration HMSC operator defines iteration execution of bMSCs. In Figure 2.10.c, bMSC *bMSC* will be executed repeatedly.

Parallel HMSC operator defines the parallel execution of bMSCs or HMSCs. In Figure 2.10.d, HMSCs *HMSC1* and *HMSC2* will be executed simultaneously.

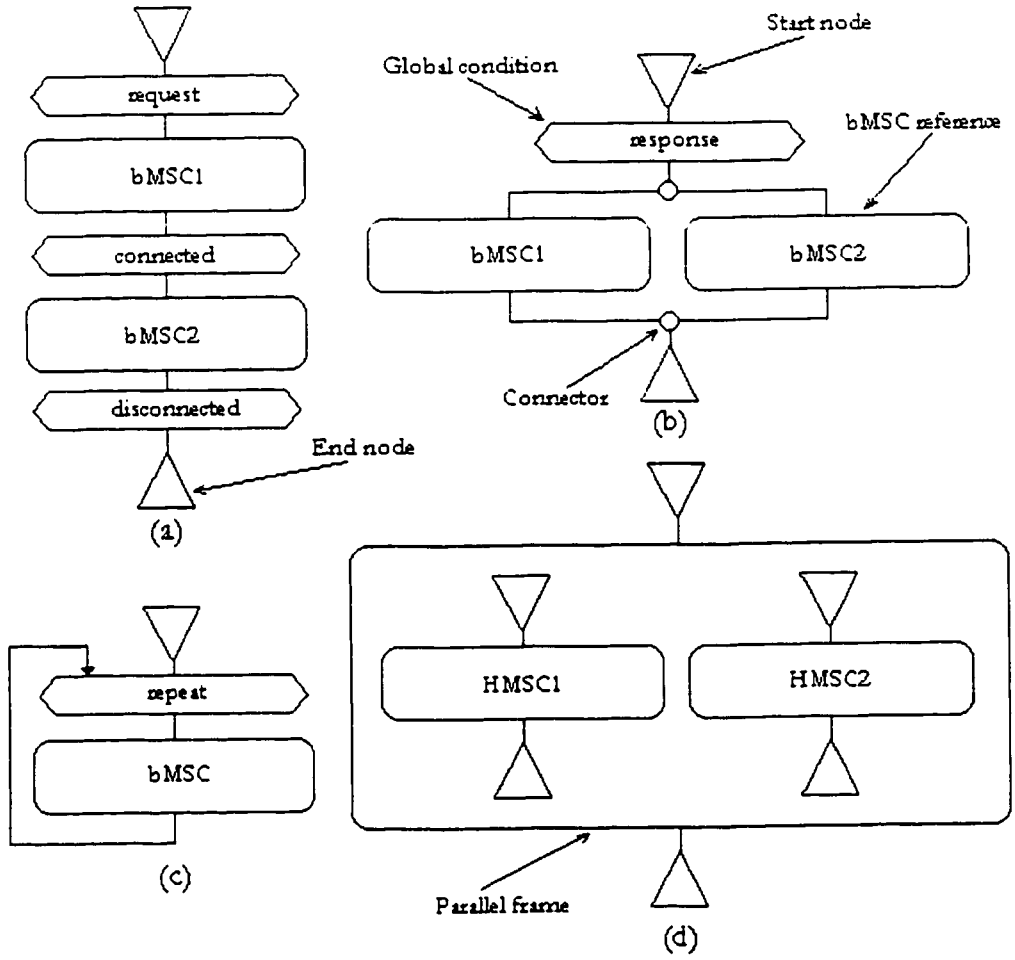


Figure 2.10.HMSC operators.

2.4 A MSC EXAMPLE

The given MSC, in Figures 2.11 and 2.12, demonstrates sequential and alternative bMSC compositions. The system is composed of two instances, *caller1* and *callee1*. Instance *caller1* tries to connect to instance *callee1*. In case, instance *caller1* does not receive a response from instance *callee1* during a certain period of time, it will try again for two

more times. If *caller1* does receive a response from *callee1*, they will be connected. Otherwise, *caller1* cancels the connection request and returns to the idle state.

The system overview is captured on the HMSC specification, as shown in Figure 2.11 the system begins its life by interpreting the start node. Initially, the system is in an idle state. Then, the system executes the bMSC *Request* scenario, and wait for the *callee1* response to execute bMSC *Connected* scenario or time-out signal to execute bMSC *Time-out* scenario.

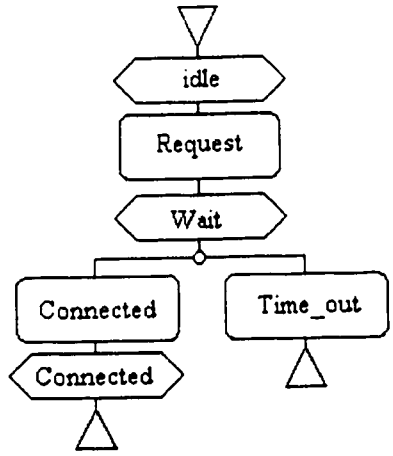


Figure 2.11. An example: HMSC specification.

Figure 2.12 shows the corresponding bMSC specifications for the HMSC in Figure 2.11.

In bMSC *Request*, instance *caller1* initializes the variable *N*, which holds the number of tries that will be made to connect to *callee1*. Next, *caller1* sends a request message (*request*) to *callee1* and sets up the timer (*T1*).

In bMSC *Connected*, instance *callee1* sends a response message (*response*) to instance *caller1*. As soon as *caller1* consumes message *response*, it resets the timer *T1*.

In bMSC *Time-out*, the timer *T1* is expired, consequently, instance *caller1* sends a cancel message (*cancel*) to instance *callee1* in order to terminate the previous request. Next, *caller1* checks the value of *N*. If *N* is greater than three, then the system will go to the idle

state. Otherwise, N will be incremented, a new request message will be sent, and the timer *T1* will be set up again.

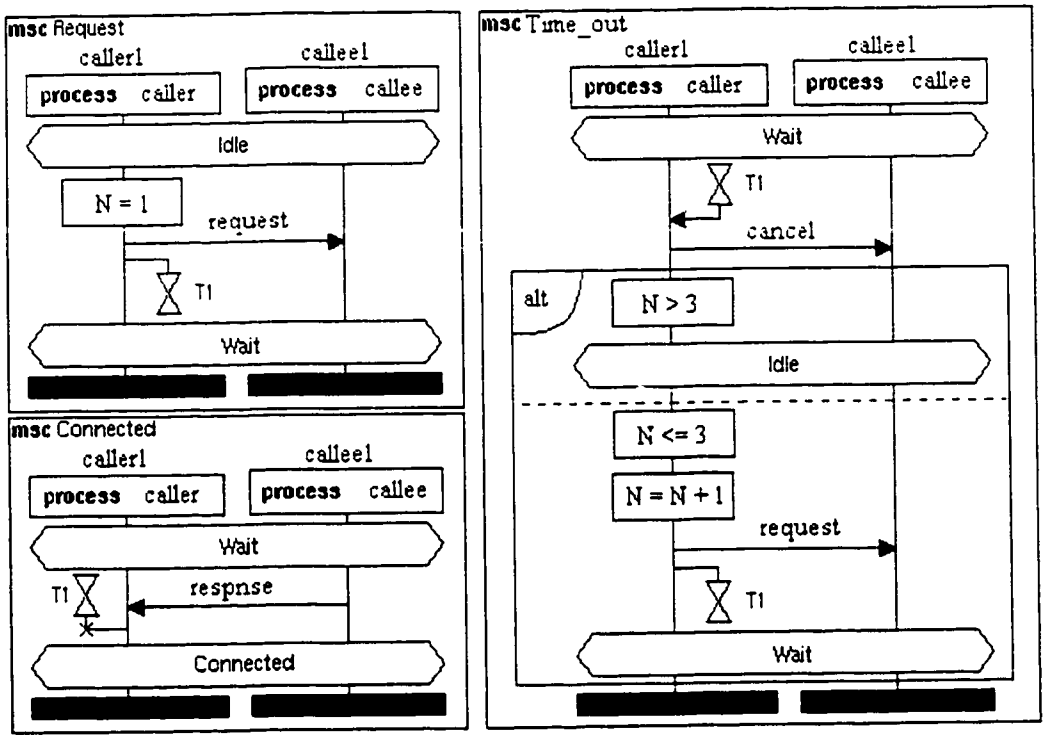


Figure 2.12.bMSC specifications correspond to HMSC in Figure 2.11.

In the later bMSC, the choice between the alternatives of the inline expression *alt* is non-deterministic because the comparison is informal and is specified within an MSC action. However, the newest revision of MSC, MSC2000, has solved this problem by providing guards, which look like conditions, that control the flow of execution.

CHAPTER 3

SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

This chapter briefly describes the syntax and semantics of a subset of SDL used throughout the report. For more details, the reader can refer to [8] and [11].

3.1 INTRODUCTION

SDL is a formal language used to specify and describe systems, especially telecommunications systems. It is mainly concerned with the specification of the behavior, structure and data. SDL was developed by ITU, and standardized for the first time in the ITU Recommendation Z.100, 1976. Since then, many releases had been issued, however, the language matured only in 1988 release, called SDL-88. To cope with new software engineering techniques such as Object Oriented programming, the ITU has released SDL-92 in 1992 and SDL-96 in 1996. While SDL is intended to be used for telecommunications systems, it may be used in all kind of real time systems. During the last decade, it has gained popularity because of the following reasons:

- It is an international standard with committed support from ITU.
- It is supported by powerful commercial tools, such as ObjectGEODE and SDT, that simulate, verify and generate code from the SDL specifications.
- It has a clear structured graphical representation.
- Its ability to view the systems at several abstract levels.

SDL has two concrete representations: graphical (SDL/GR) and textual (SDL/PR), which share the same abstract syntax. Henceforth, this report will concentrate on SDL/GR for illustration purpose. For further details, the reader can refer to [8].

An SDL system specification is a formal model that defines the relevant properties of an existing or planned system in the real world. The world, within SDL, is divided into two parts: the system and its environment, and they communicate using discrete signals. Everything outside the system is considered as part of the environment. A complete SDL system specification essentially consists of two main parts:

- **Architecture**, which describes the communication among the system entities
- **Behavior**, which describes the behavior of each entity in the system

3.2 SDL ARCHITECTURE

An SDL architecture defines the communication among the system entities and with the outside world, namely the environment. An SDL architecture can be typically divided into different levels of hierarchy. The highest level shows only the major components (top-level blocks) of a system, whereas the lowest level shows the details (behavior) of the system. Figure 3.1 shows three levels of an SDL system specification.

The highest level shows the system specification of the SDL system *S1*. The system consists of two blocks (*B1* and *B2*) that communicate with each other through channel *Ch2*. Block *B2* communicates with the environment through channel *Ch1*.

The intermediate level, in this Figure, is represented by the specifications of blocks *B1* and *B2*, for illustration, the Figure shows only *B1* specification. Block *B1* consists of two processes *P1* and *P2* that communicate through the signal route *Sr2*. Process *P1*

communicates with the block environment through signal route *Sr1*, which is connected to channel *Ch2* at the block border. Process *P2* sends signal (*Z*) to the block environment through signal route *Sr3*, which is connected to the channel *Ch2* at the block border.

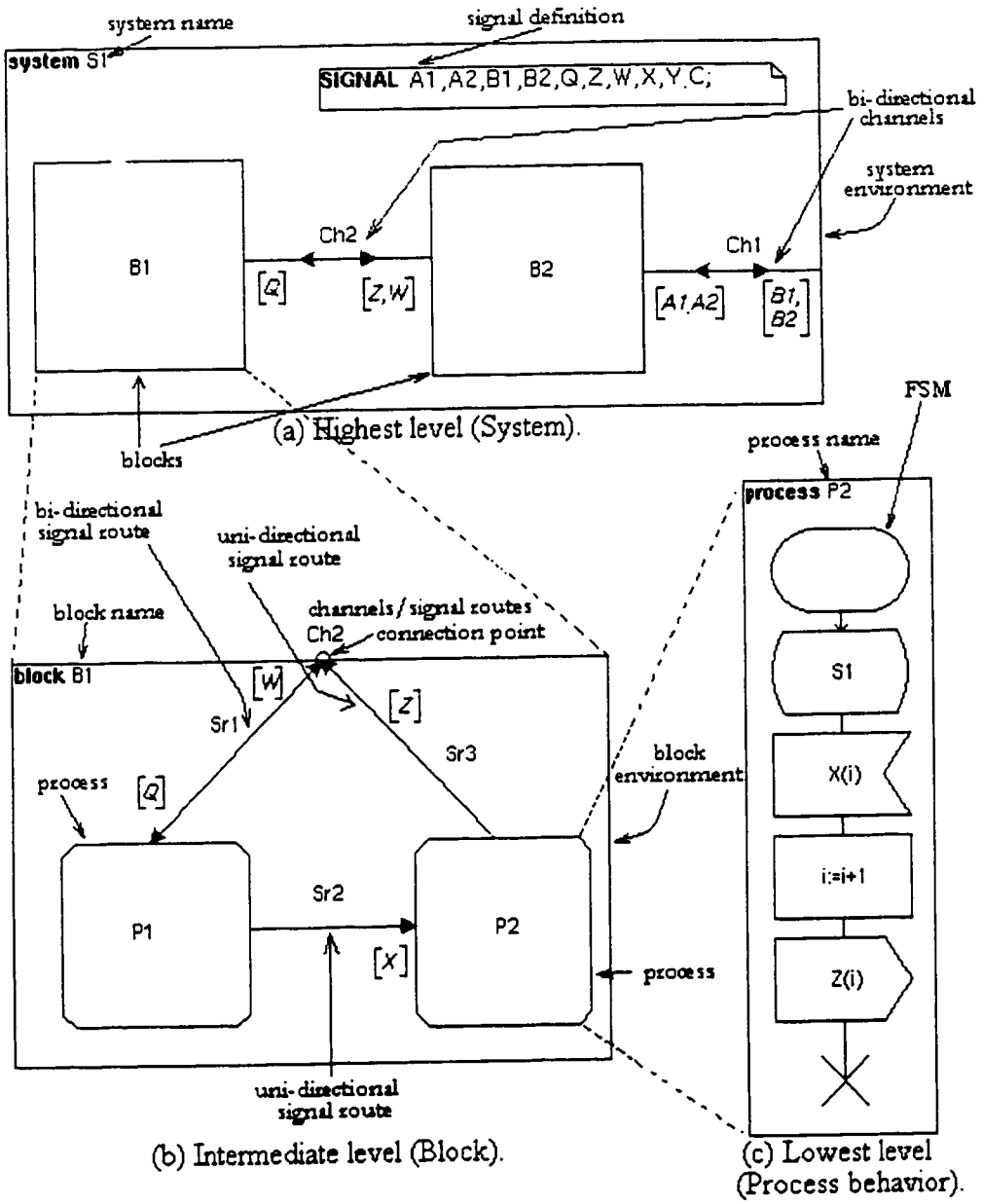


Figure 3.1. SDL system hierarchy.

The lowest level is represented by the process behaviors. The Figure shows the behavior of process *P2*. Process *P2* has only one state *S1*, and after interpreting the start node, it consumes signal *X*, which is sent by *P1* according to the block specification. Next, *P2*

increments the value of the received parameter *i*, and sends it within the signal *Z* to the other block *B2* through signal route *Sr3*.

3.2.1 ENVIRONMENT

The environment has the ability to exchange signals with the SDL system. It is expected that the environment behaves in an SDL-manner and that it does not send signals to the system in a fully random fashion. The environment is represented by the boundary of the SDL system.

3.2.2 THE SYSTEM

The system may compose of a set of nested blocks communicating with each other and with the environment through channels. They communicate by sending/receiving signals (referred to as messages in MSCs). Blocks may contain recursively sub-blocks or contain processes. Processes communicate with each other and with the block environment through signal routes that convey signals. Channels and signal routes are infinite FIFO (First In First Out) queues and convey signals in one or two directions. These channels and signal routes exchange their signals at block borders. The only difference between channels and signal routes is that channels can postpone signals for a non-deterministic period (called delayed channels), while signal routes deliver signals instantaneously.

3.2.3 BLOCKS

Blocks are used to provide multi-level hierarchy within the SDL system. Blocks are components either of a system or of higher-order enclosing blocks. Lower level blocks only contain one or more processes. Blocks communicate with each other via channels.

They also communicate with the system environment or higher-order block environment via channels, which connect to signal routes at the border of blocks.

3.2.4 PROCESSES

Processes define the behavior of the SDL system. A process behavior is described as an Extended Finite State Machine (EFSM). The dynamic system, during lifetime, may have multi-instances of a process, called process instances[1]. The number of instances of a process that are created at the system startup can be specified between parentheses in the process definition. Furthermore, the maximum number of instances that can be existing together throughout the system execution can be specified. For example, the following process definition, PROCESS *P1*(2,5), allows the system to create two instance of *P1* at the startup, and no more than five instances can be exist together. The instances may communicate with each other and with the environment via signal routes. There is no hierarchy associated with instances, and they run in parallel with equal rights. Some instances may create others, but once created, the new instances run in parallel with the creating ones.

Each instance has a unique FIFO queue called (input queue) that preserves the order of the received signals. However, it does not guarantee the order of signals that sent via different channels/signal routes. Furthermore, each instance has a unique identification address called (pid). Signals travel associated with their sender pids. Instances can be created immediately by the system at the start of the execution, or by other instances during the system run time.

[1] A process defines the static behavior of the SDL system, while process instance defines the dynamic behavior of the SDL system.

3.2.5 COMMUNICATION BETWEEN INSTANCES

Each instance is assigned a unique address during the system startup. This address can be used by other instances to send signals. However, an instance, when created, does not have any information about other instances except about the following:

- Itself by using the pre-defined expression *self*.
- Its parent, the instance that created it, by using the pre-defined expression *parent*, if does not created by the system.
- Its last child which has been created by it, by using the pre-defined expression *offspring*.
- The sender of the last consumed signal by using the pre-defined expression *sender*.

At creation time, instances usually require to handshake with each other to build a database of their world/system. An instance can send signals using explicit or implicit addressing. Still, the specifier has to take on his account the following SDL rules:

- If a signal cannot find a unique path for delivery, an arbitrary path out of the possible ones is selected.
- If the destination process has more than one instance, an arbitrary instance is selected.
- Signals that do not find a destination instance are discarded.

3.2.5.1 EXPLICIT ADDRESSING

An instance can send signals to a unique instance by providing the exact pid of the destination instance. Since instance pids are known only during execution time, explicit addressing can be used just in the following cases:

1 An instance can explicitly address itself by using the predefined expression *itself*.

2 An instance can explicitly address its parent by using the predefined expression *parent*.

3 An instance can explicitly address its latest child by using the predefined expression *offspring*, or other children, which their pids had been saved previously.

4 An instance can address explicitly the sender instance of last consumed signal by using the predefined expression *sender*, or other instances which their pids had been saved previously. In this case, the consumed signal should be uniquely identified, otherwise, the instance may communicate with undesired instance.

3.2.5.2 IMPLICIT ADDRESSING

In the case where the pid of the destination instance is not known, SDL instances have to address their destination implicitly using one of the following methods:

1 An instance can send signals to the target instance by specifying the path, which connects the sender instance with the receiver instance. The path is composed of a list of signal routes and channels. To ensure the delivery of the signal, the path should be unique.

2 An instance can send signals to the target instance by specifying the SDL process name of the target instance. In this case, the target process should have only one instance to ensure the delivery of the signal.

3 An instance can send signals to the target instance without specifying the target address. The signal should be unique, and it is used only by the two parties.

3.3 SDL BEHAVIOR

An SDL system behavior is composed of a set of process behaviors. Each process describes a part of the system behavior, as an Extended Finite State Machine, as shown in Figure 3.2. An SDL process is either in a state or in transition between two states. A transition is triggered by the consumption of a signal.

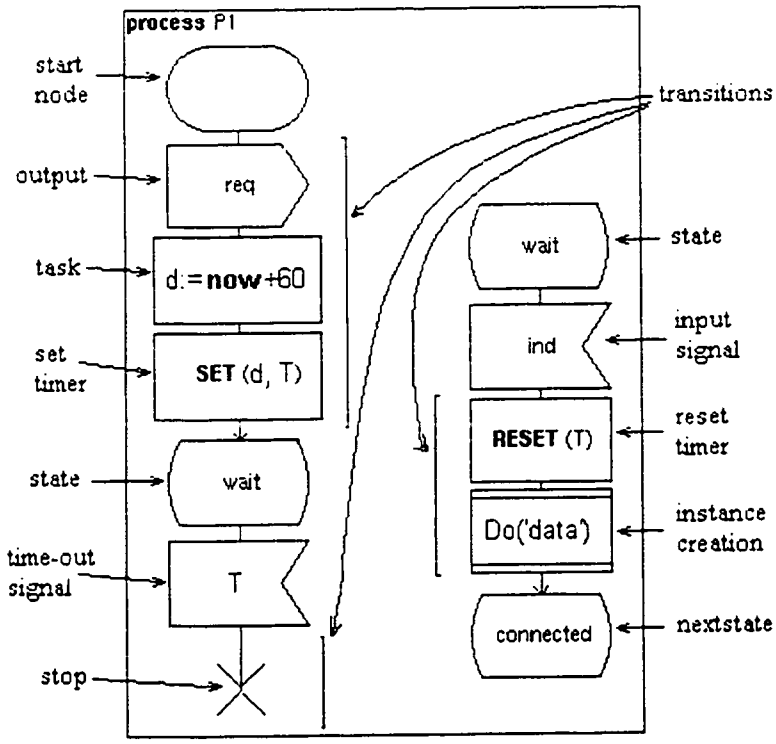


Figure 3.2. SDL behavior.

3.3.1 STATE

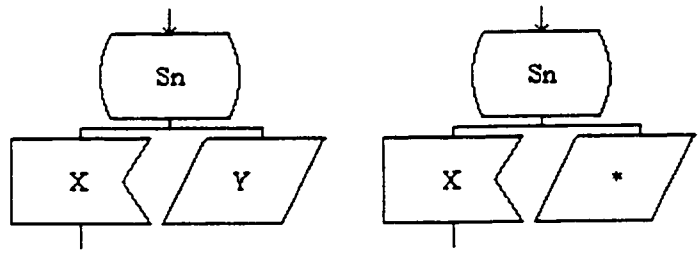
States are stable points in the behavior of a process. The transition from state to state is triggered by consuming a received signal from the instance input queue. A spontaneous transition can be used to progress the SDL behavior without consuming a signal. It is expressed by using the keyword *none* in an SDL input construct. While spontaneous transitions do not depend on the contents of the input queue, their activation is non-deterministic and makes the state unstable. No priority is assumed between transitions activated by signal consumption and spontaneous transitions. To provide a simple clear specification, the same state can be shown in different places, in the process behavior, with different inputs. Each process has a single start node and as soon as a process instance is created, it begins its lifetime by interpreting the start node.

3.3.2 INPUT

An instance cannot consume a signal until it has been received in its input queue. If the input queue is empty, the instance remains in the current state until one of the specified input signals has been arrived in the input queue. Unspecified input signals, which are in the head of the input queue, are discarded. The consumed signals will be removed from the input queue. The pid of the sender can be discovered after the consumption of the signal by using the pre-defined expression *sender*.

3.3.3 SAVE

If a signal in the head of the input queue is not specified by the current state, it will be discarded without triggering the state, i.e. the instance remains in the same state. To protect the signal for later consumption, SDL provide a save construct. Any signal that included in the save construct will stay in the input queue, and the next signal in the input queue will be used. In Figure 3.3.a, for instance, signal *Y* will be protected while looking for signal *X* in the input queue. In case of protecting all signals that may arrive before the specified signal in the current state, one save construct can be used with an asterisk, as shown in Figure 3.3.b.



(a) specific save. (b) wild save.

Figure 3.3. SDL save.

3.3.4 TRANSITIONS

During transition to the next state, instances can perform activities such as sending signals, setting timers, creating instances, etc.

3.3.4.1 OUTPUT

An instance can send signals to other instances, environment or itself during transition between states. The signal can be send implicitly (Figure 3.4.a), via a particular path (Figure 3.4.b) by providing the list of signal routes/channels of the desired path, or to a process entity (Figure 3.4.c) by providing the process name. Furthermore, the signal can be send explicitly (Figure 3.4.d) to the destination instance by providing the destination instance pid.

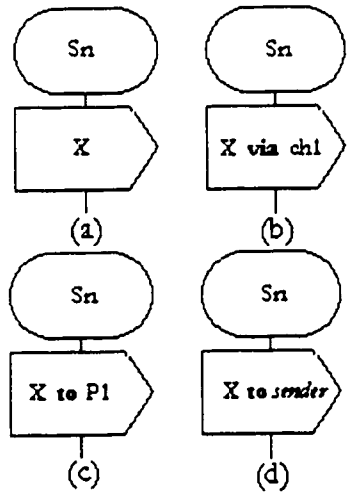


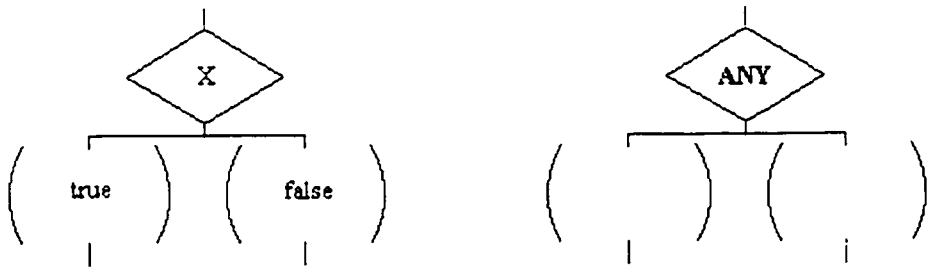
Figure 3.4. SDL output.

3.3.4.2 TASK

In addition to output, a transition can include tasks. Tasks represent process internal activities. These tasks may contain informal text or assignment expressions.

3.3.4.3 DECISION

Using a decision, a transition may split into two or more branches depending on the result of the decision expression, (e.g., Figure 3.5). Each result may lead to a separate state. The pre-defined expression *any* can be used to represent a non-deterministic choice, as shown in Figure 3.5.b.



(a) deteministic choice. (b) non-deterministic choice.

Figure 3.5. SDL decision.

3.3.4.4 STOP

An instance can only be terminated by itself. In process definition, it may have many transition branches that lead to a stop node. After the stop node, the instance ceases to exist.

3.3.4.5 CREATING NEW INSTANCES

Instances may create other instances that present in the same block. However, as soon as an instance is created, it runs in parallel to, and independently from, the creating instance.

3.3.4.6 TIMERS

Timers, in SDL, are independent processes. They are created by the SDL system as soon as the owning instances are created. They run in parallel with the owning instances and with equal rights, until the owning instances stop. Initially, timers are inactive. When a timer is activated, it is set to a given time and supervises the current time. As soon as the

timer expired, a timer signal is put in the input queue of the owning instance. A timer becomes inactive when the timer signal is consumed, or reset by the owning instance.

3.4 DATA PART IN SDL

In our investigation, we have not focused on data types since MSC so far does not have a clear semantics for data types. However, our approach declares all generated SDL timers, pid variables and some action variables. In this section, we give a brief description of SDL data types.

SDL uses the concept of abstract data types. Data types are called *sorts* in SDL. As in programming languages, SDL provides the user with some pre-defined sorts, and allows him to create new sorts. 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 a sub-level of it. SDL signals, Timers and variables should be declared before they can be used. SDL signal names should be declared at the system level by using the keyword *SIGNAL* (e.g., *SIGNAL* x, y, z;). SDL timers are declared within the owning process scope by using the keyword *TIMER* (e.g., *TIMER* T1;). SDL variables are declared within the process scope by using the keyword *DCL* (e.g., *DCL* x, y *integer*;)

Pre-defined sort	Example
Boolean	true, false
Character	'A', '$', ' ', '9'
Charstring	'Hi', 'SDL'
Integer	-3, 0, 902
Natural	0, 12
Real	-2.305, 0, 34.84010
Pid (Process Identification)	null
Duration	-4, 0, 1
Time	-1, 0, 103

Table 3.1. SDL pre-defined sorts.

3.5 AN SDL SPECIFICATION EXAMPLE

For illustration purpose, we will consider here the SDL system *S* shown in Figure 3.6. It consists of one block *Blk*. There is no communication between the system and the environment. Block *Blk* consists of two processes *P1* and *P2* which exchange signals *X* and *Y* through signal route *Sr*. From the shown process behaviors, the two processes exchange signals *X* and *Y* one hundred times.

SDL process *P1* begins its life by initializing the loop counter (*N*) and transits to the next state *St1*. Since there is no signal specified in order to be consumed, *P1* waits until the spontaneous input (*NONE*) is activated. Next, *P1* sends the signal *X* to process *P2*, and moves to the next state *Wait*. *P1* remains in the state *Wait* until signal *Y* is received in its input queue. As soon as signal *Y* arrives in the input queue of *P1*, *P1* consumes it and checks the value of the counter *N*. If *N* is greater than 100, then process *P1* will terminate. Otherwise, *P1* increments the counter by one, and returns to the state *St1*.

On the other hand, SDL process *P2* begins its life by initializing the loop counter (*N*) and transits to the next state *Wait*. *P2* remains in the state *Wait* until signal *X* is received in its input queue. As soon as signal *X* is arrived in the input queue of *P2*, *P2* consumes it and sends signal *Y* to *P1*. Then, *P2* checks the value of the counter *N*. If *N* is greater than 100, then process *P2* will terminate. Otherwise, *P2* increments the counter by one, and return to the state *Wait*.

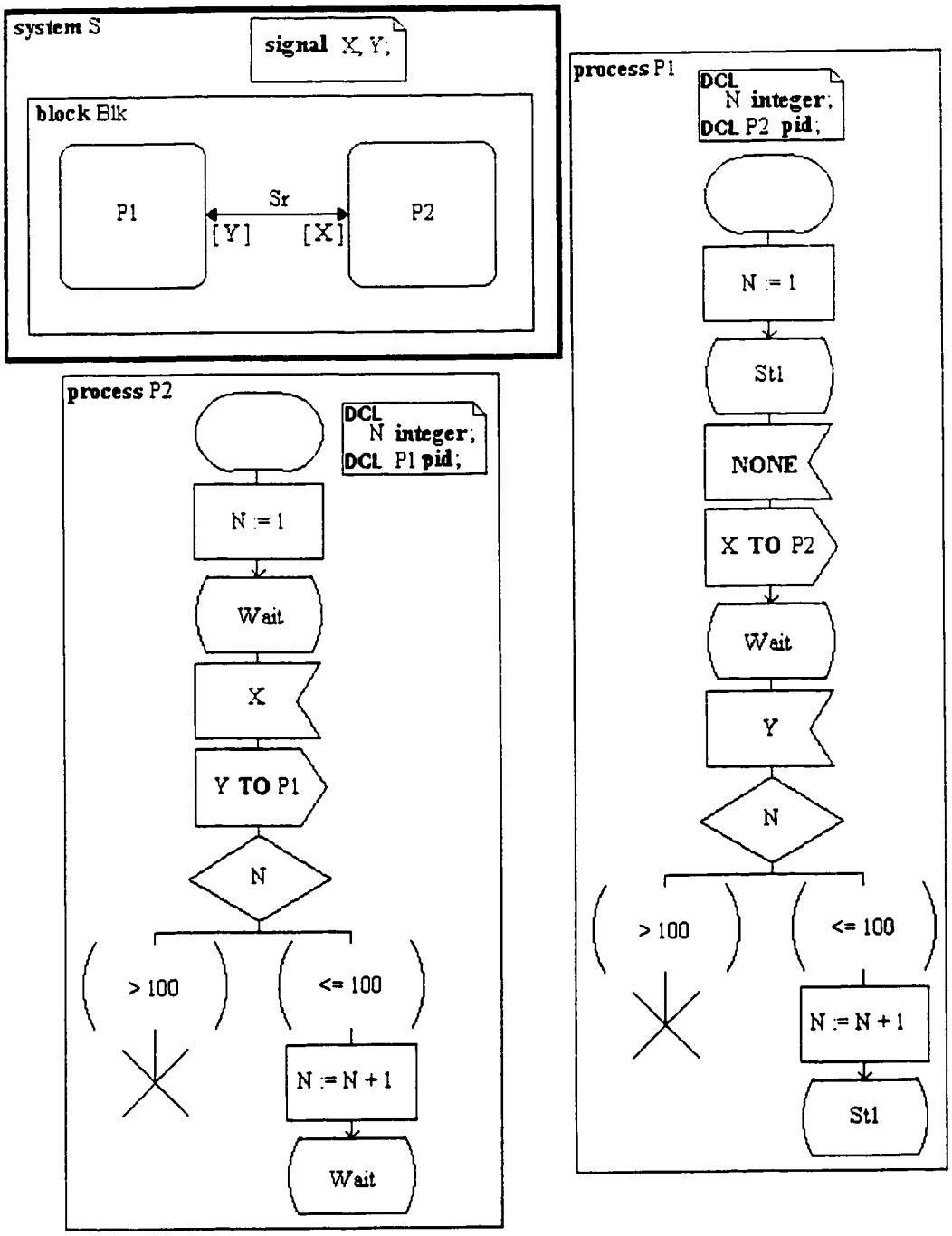


Figure 3.6. An SDL example.

CHAPTER 4

GENERATION OF SDL BEHAVIOR FROM A bMSC

4.1 MOTIVATIONS

The rapid growth of the volume of software being produced and the almost exponential increase in the complexity of problems being solved with software have urged the demand for systematic software development techniques. Due to the increase of the complexity of telecommunications systems in recent years, systems specifications are more difficult to capture, and their manual translations have become complex and error prone.

Furthermore, competition in the telecommunications market has resulted in more pressure on software system developers to respond quickly to the growing market demands. In order to meet these demands, developers may need several weeks to months depending on user requirements. This translates into time-to-market going far beyond consumer expectations. One of the causes of this situation is the lack of synthesis tools that support transition between software development phases, especially between the requirement and design phases.

Our particular interest is in a systematic translation from behavioral requirements to design with a given and fixed architecture. For the purpose of this research, we have

taken MSC as a requirement specification language and SDL as a design specification language. Our goal is to provide a systematic approach to generate SDL specification from MSC specification.

MSC and SDL have been widely adopted within the telecommunications industry, and currently many tools support both languages. Each tool provides a complete package of documentation/specification languages used throughout the software process. However, users are required to go manually from one phase of the software process to the next phase. For instance, ObjectGEODE, which we have used in our approach, does not have any mechanism for translating MSC to SDL. Consequently, users have to manually translate their MSC specification, requirement phase, into SDL specification, design phase. This manual translation is a repeated effort and is time consuming. Since MSC and SDL are compatible languages, a synthesis tool can be implemented. On the other hand, most of these tools can transform SDL specifications into C++ or Java code, implementation phase.

4.2 THE BASIC APPROACH

The basic approach, which we have implemented and extended, was introduced by Robert and Khendek in [4]. The approach translates a bMSC specification with a given SDL architecture into SDL process behaviors. Essentially, the bMSC consists of instances that exchange messages. The approach ensures, by construction, consistency between the SDL specification and the bMSC specification. The generated SDL specification requires non-additional validation against the requirements, in this case the given bMSC specification. The approach also builds two tables to prevent any deadlock

that may be generated during translation. In the following subsections, the basic approach will be introduced using an example with one bMSC specification (Figure 4.1) and two different SDL architectures (Figure 4.2).

4.2.1 CONSISTENCY

In the software process, the requirement specification has to be validated against the user requirement. This approach assumes that the MSC specification has been validated against the user requirement and is used as a reference for the semantic consistency of the SDL specification. Since the MSC and SDL specifications of a given system are often developed independently from each other, the SDL specification has to be validated against the MSC specification to ensure consistency between the requirement phase and design phase. An SDL specification is semantically consistent with respect to a bMSC if and only if the set of traces defined by the bMSC is included in the set of traces of the SDL specification. Since the approach synthesizes SDL from bMSC, the approach ensures, by construction, consistency between the SDL specification and the bMSC specification, and no further validation against the requirement specification is required. Furthermore, the approach guarantees that the generated SDL specification is free of deadlocks (i.e. state where all the queues in the SDL system are empty and there is no possible progress for any process).

To make sure that the given bMSC specification can be executed under the given SDL architecture, the architectural consistency between the bMSC and the SDL architecture is verified. The architectural consistency is defined as follows:

- All processes described in the given bMSC are present in the given SDL architecture.

- There is a path connecting the sending and the receiving processes in the given SDL architecture for each message described in the given bMSC. While SDL specifications allow implicit and explicit routes, the approach accepts only explicit routes.

Since bMSCs define only partial behaviors of the SDL system behavior, the SDL architecture may consist of more processes and routes than the required bMSC architecture.

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 explicitly specified. 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 processes interleaving. On the other hand, SDL instances implicitly discard signals, which are in the front of their input queues, and are not expected at the current state. These discarded signals, which may be required in the next states, may lead to a deadlock. In Figure 4.1, instance *I2* has to consume specified messages in the following order: *a*, *b* then *c*. However, the actual arrival order of messages to *I2* input queue may be different from the consumption order, depending on the target architecture and processes interleaving. For the target SDL architecture given in Figure 4.2.a, the possible message interleavings for *I2* are:

- *a*, *b* then *c*
- *a*, *c* then *b*
- *b*, *a* then *c*

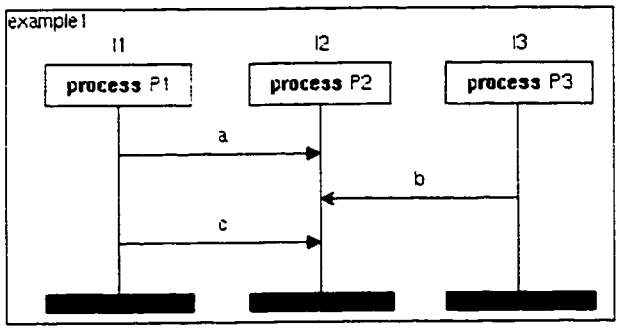


Figure 4.1. bMSC specification used to illustrate the basic approach.

According to the SDL specification, there is no problem with the first order since signals *a*, *b* and *c*, arrive according to the consuming order. The other two lead to deadlocks. In the second order (a, c then b), after *I2* consumes signal *a*, signal *c* is discarded since *I2* is expecting signal *b* not *c*. Consequently, *I2* will never reach completion. In the last order, (b, a then c), signal *b* is discarded since *I2* is expecting to consume *a*. After consuming *a*, signal *c* is discarded and *I2* will wait for the previously discarded signal *b*. More deadlock scenarios can arise, if the SDL architecture allows signals *a* and *b* to travel through different paths as in Figure 4.2.b.

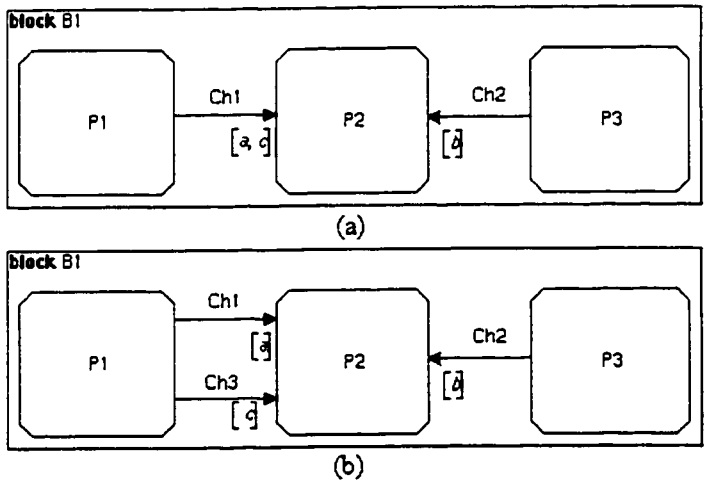


Figure 4.2. Target SDL architectures used to illustrate the basic approach.

To prevent deadlocks, the approach adds an SDL save construct for each signal that may arrive in the input queue earlier than expected. The need for a save construct can be determined by checking the order relation of each specified input event against the order of all successive input events for the same instance. The order relation between each pair of events, according to ITU Recommendation Z.120, can be determined by the following rules:

- Events are totally ordered for each instance axis
- The output event of a message precedes the corresponding input event

The approach builds a table, called Event Order Table, which maintains the order relations in time between input/output events represented by rows and columns. The table cells are filled by a logic value (true), if the row event precedes, in time, the corresponding column event. First, all input/output events in the given bMSC will be assigned a unique sequential number, as in Figure 4.3. Then, the cells of each row are marked if the row event precedes the corresponding column event.

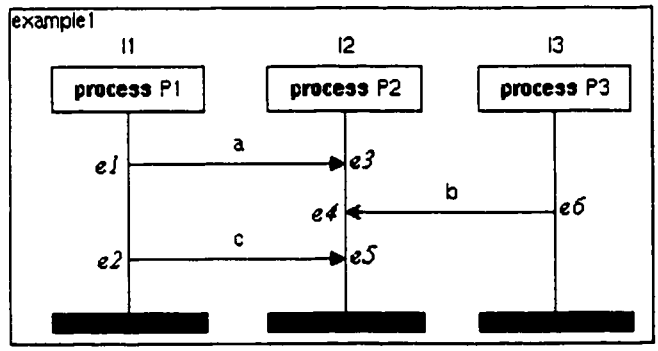


Figure 4.3. Numbering input and output events.

Table 4.1 shows the Event Order Table corresponding to the bMSC in Figure 4.3. For instance, event *e6* is an output event. By applying the first rule, there are no events below event *e6* on *I3* axis, so no events will be put in the event row. Since *e6* is an output event,

the second rule will be applied, *e4* is the corresponding input event. The approach puts *e4* in row *e6*. Furthermore, *e4* is above *e5* on *I2* axis, and consequently, *e5* will be put in row *e6*.

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

Table 4.1. Event Order Table corresponds to bMSC specification in Figure 4.3.

4.2.3 OCCUPANCY TABLE

In addition to building an Event Order Table, the approach builds Occupancy Tables. The Occupancy Table maintains the order relations between the input signal of the corresponding SDL process. The number of tables is equal to the number of MSC processes that have input events. The rows of each table represent only the input events of the corresponding MSC process. The columns of each table represent the incoming routes that convey signals from other processes. The table is filled with all input signals that may exist in the input queue when the process is ready to consume the row event by applying the following expression on all input events in the same instance axis:

$$\text{NOT } (e_r \ll e_c) \text{ AND NOT } (e_c \ll e_s)$$

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.

$\ll$ denotes order relations (precedes).

($e_x \ll e_y$) mean e_x precedes e_y in time. This relation can be extracted from the Event Order Table where e_x represents the row event and e_y represents the column event. For instance, from Table 4.1, ($e_1 \ll e_3$)=T and ($e_1 \ll e_6$)=F.

Table 4.2 shows the Occupancy Table corresponding to the bMSC specification in Figure 4.3 and SDL architecture in Figure 4.2.a. Since processes *P1* and *P3* do not have input events, there is no corresponding Occupancy Table. Process *P2* has three input events *e3*, *e4* and *e5*, and two incoming routes *Ch1* and *Ch2*. For example, when *P2* is ready to consume *b* (row *e4*), the approach checks the input messages *a*, *b* and *c*.

For message *a*
NOT ($e_r \ll e_c$) AND NOT ($e_c \ll e_s$)
NOT ($e_3 \ll e_4$) AND NOT ($e_4 \ll e_1$)
from the Event Order Table in Table 4.1
NOT T AND NOT F $\Rightarrow$ F AND T $\Rightarrow$F
Then, message *a* is not included in the Occupancy Table row of event *e4*.

For message *b*
NOT ($e_r \ll e_c$) AND NOT ($e_c \ll e_s$)
NOT ($e_4 \ll e_4$) AND NOT ($e_4 \ll e_6$)
from the Event Order Table in Table 4.1
NOT F AND NOT F$\Rightarrow$ T AND T $\Rightarrow$ T
Then, message *b* is included in the Occupancy Table row of event *e4* at the appropriate column *Ch2*.

For message *c*
NOT ($e_r \ll e_c$) AND NOT ($e_c \ll e_s$)
NOT ($e_5 \ll e_4$) AND NOT ($e_4 \ll e_2$)
from the Event Order Table in Table 4.1
NOT F AND NOT F$\Rightarrow$ T AND T $\Rightarrow$ T
Then, message *c* is included in the Occupancy Table row of event *e4* at the appropriate column *Ch1*.

input events for *P2*	input message	Route *Ch1*	Route *Ch2*
e3	*a*	*a, c*	*b*
e4	*b*	*c*	*b*
e5	*c*	*c*	

Table 4.2. Process *P2* Occupancy Table for SDL architecture in Figure 4.2.a.

Table 4.3 shows the Occupancy Table corresponding to the bMSC specification in Figure 4.3 and SDL architecture in Figure 4.2.b. Since processes *P1* and *P3* do not have input events, there is no corresponding Occupancy Table. Process *P2* has three input events *e3*, *e4* and *e5*, and three incoming routes *Ch1*, *Ch2* and *Ch3*.

input events for *P2*	input message	Route *Ch1*	Route *Ch2*	Route *Ch3*
e3	*a*	*a*	*b*	*c*
e4	*b*		*b*	*c*
e5	*c*			*c*

Table 4.3. Process *P2* Occupancy Table for SDL architecture in Figure 4.2.b.

4.2.4 TRANSLATION

After building the necessary tables, the approach begins translating MSC constructs to SDL constructs on a one-to-one basis. An SDL process progresses from a state to another state by consuming an input signal. The approach inserts a new SDL state before each MSC input event. Consequently, the approach adds an SDL save construct for each message in the corresponding Occupancy Table row, except for messages that are sent by the same instance and travel on the same route of the input message. Figure 4.4 shows the generated SDL process behaviors from the given MSC specification in Figure 4.1 and the target SDL architecture in Figure 4.2.a.

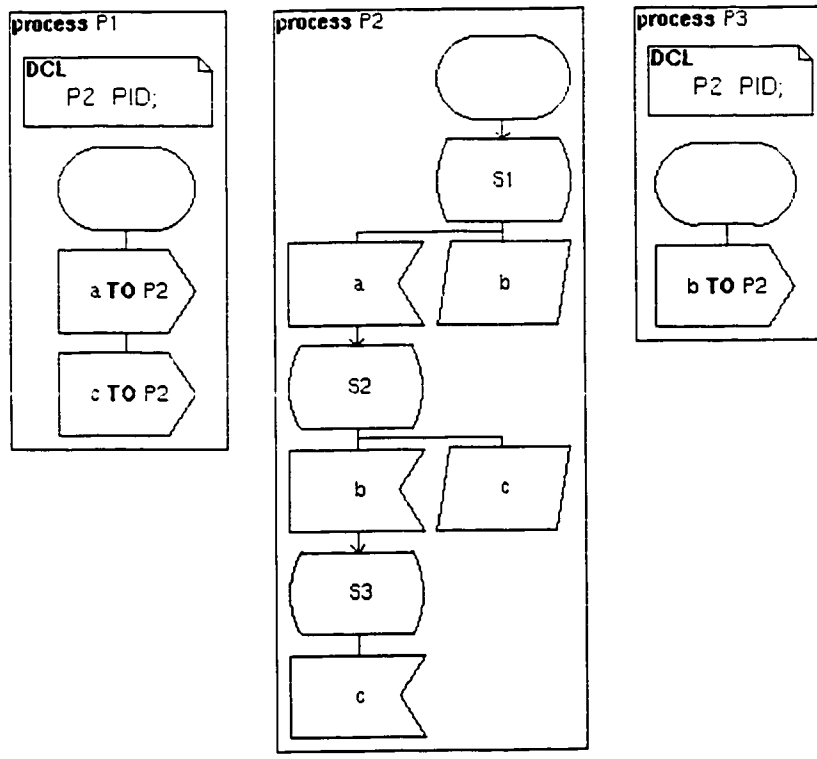


Figure 4.4. Generated process behaviors for SDL architecture in Figure 4.2.a.

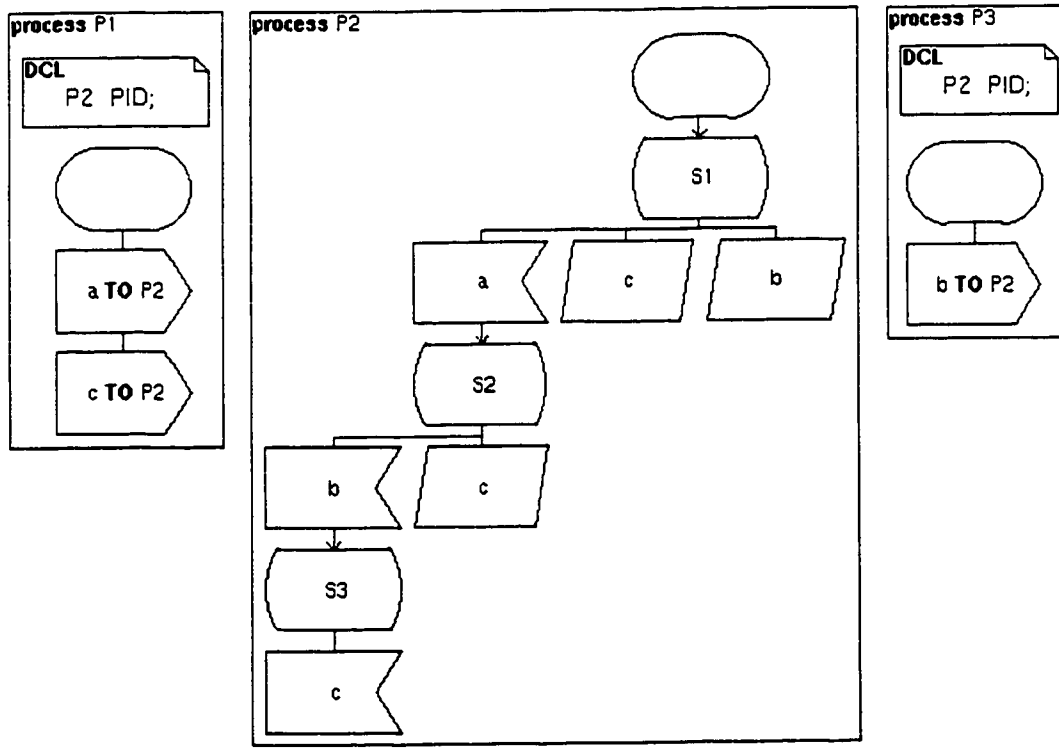


Figure 4.5. Generated process behaviors for SDL architecture in Figure 4.2.b.

Figure 4.5 shows the generated SDL process behaviors from the given MSC specification in Figure 4.1 and the target SDL architecture in Figure 4.2.b. The only difference between the two generated system behaviors is that in Figure 4.5 process *P2* saves signal *c* while waiting for signal *a* because signals *a* and *c* are coming through different routes.

4.2.5 ALGORITHM

In this section, we describe the steps that should be followed to apply the basic approach as has been described in [4].

1- **Check the architectural consistency between the given SDL and MSC.**
   - For each process described in the MSC, there is a corresponding process type in the SDL architecture.
   - Each message described in the MSC is enumerated in the SDL architecture by a route connecting the sending process and the receiving one.

2- **Number each input/output event uniquely.**

3- **Build the Event Order Table.**
   - For each input/output event, the corresponding column of all input/output events below the current row event is marked.
   - For each output event, the column of the corresponding input event is marked. Furthermore, all events that are marked in the corresponding input event row are marked.

4- **Fill the Occupancy Tables**

```
for each instance Pi in the MSC diagram
   for each out event e_s, sending message m to instance Pj
      find the related input event e_r in instance Pj
      for each in event e_k, in instance Pj
         if NOT (e_k ≪ e_s) AND NOT (e_r ≪ e_k)
            add message m to the appropriate receiving queue
         end
      end
   end
end
```

5-**Generate the SDL code**

```
for each instance Pi in the MSC diagram
    for each event ej
        if the event is an out generate a SDL output
        else if the event is an in of message m
            generate a SDL input of message m
            for each receive queue of Pi (except the queue to which m belongs)
                generate a SDL save for all the messages in the queue
            end
        end
    end
end
```

4.3 EXTENSIONS TO THE BASIC APPROACH

In this section, we illustrate our contributions to the basic approach. We have extended the bMSC approach to include MSC timers, instance creations, inline expressions and coregions. In addition, we discuss inconsistency problems between the MSC specification and the SDL specification.

4.3.1 SDL PROCESS INSTANCE IDENTIFICATION AND ADDRESSING

Messages exchange between instances is explicitly illustrated in the MSC graphical representation. The sender and the receiver of any message are clearly expressed without using any addressing or identification scheme. The sender is identified by attaching the tail of the message to the sender axis, as shown in Figure 4.6.a. At the other end, the message head points to the receiver axis. In SDL, process instances have to communicate with each other using pid (process identification) addresses. The pid addresses are dynamically assigned during the system startup time, so on every system startup, different pids may be assigned to SDL process instances. Initially, instances do not know each other, and they have to handshake with each other in order to gather information

(pid) about other existing parties. Inconsistency between MSC and SDL may appear at initialization part of the specification, if an output signal can be sent to different processes or process instances without knowing the pid of the destination instance. We use the scenario in Figure 4.6.a to illustrate this problem. In this MSC specification, it is explicitly specified that instance *I1* has to send message *X* to instance *I2*. However, we cannot express it under the given SDL architecture given in Figure 4.6.b, since the signal can be received by undesired instance *I3*, *I4* or *Env* instead of instance *I2*. To solve this problem, SDL users have to assign a unique signal between instances to handshake, and split multi-instances into different blocks.

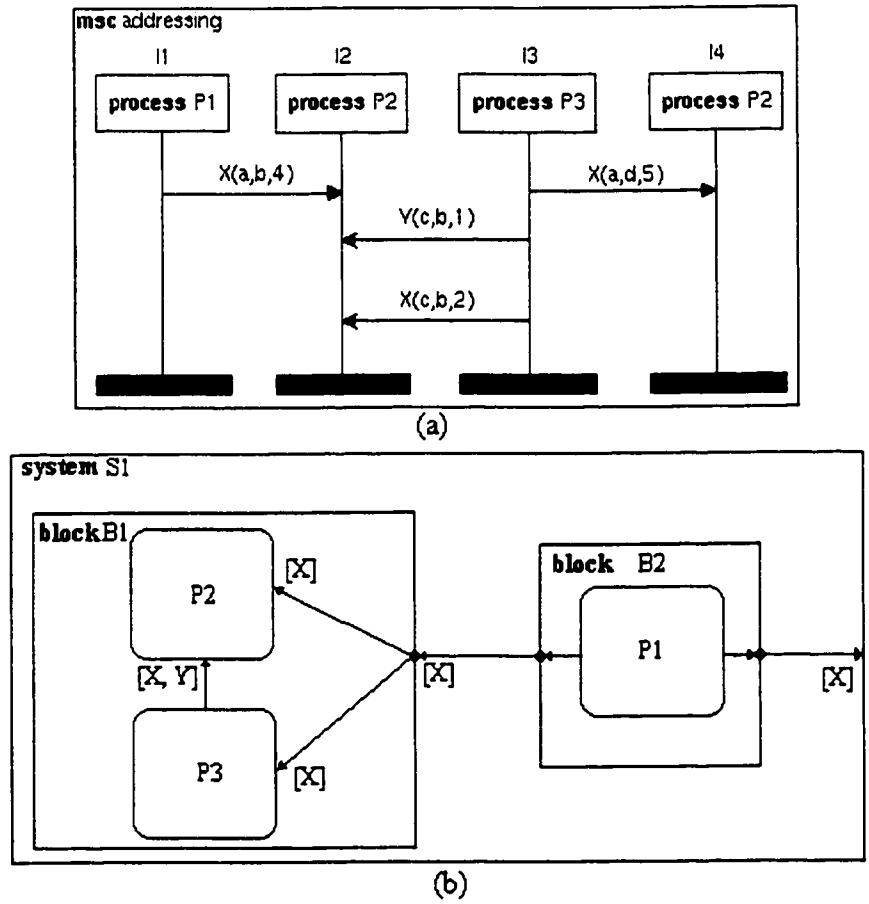


Figure 4.6. Indistinguishable signals.

A similar situation may arise at the receiving side, when two or more instances send the same signal to one instance. This situation is illustrated by the MSC shown in Figure 4.6.a. In fact, while instance *I2* is waiting for signal *X* from instance *I1*, *I2* may consume signal *X* sent by instance *I3*, which may be the first one to arrive into the input queue of *I2*. According to SDL, *I2* cannot distinguish between the two X signals, since the sender pid can only be known after consuming the signal (which can be the wrong signal). This scenario may not be typical, however, it does illustrate the usefulness of a new construct in SDL for the modeling of distributed systems. It will allow for checking the sender pid or another condition on signals in the input queue without consuming them, and without having a side effect as in Promela [5].

In order to use explicit addressing, our approach creates SDL pid variables to save the pids of the other instances that communicate with it. After each input signal, the approach saves the pid of the sender instance into an SDL variable if the pid will be used later, as shown in Figure 4.11 and Figure 4.15. Therefore, if an instance receives a signal where there is no output specified in the MSC specification to the same instance, the sender pid will not be saved. Moreover, the child pid is saved into an SDL variable as soon as the child is created, if the parent instance is required, by the MSC specification, to send messages to its child as shown in Figure 4.15.

4.3.2 INSTANCE CREATION

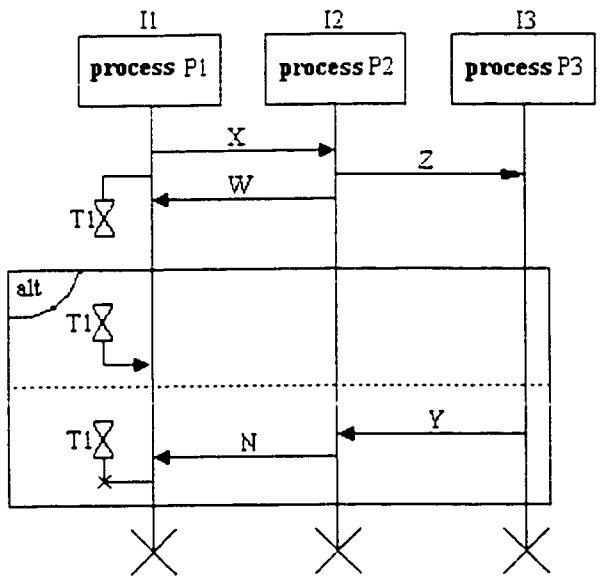
MSC create event defines an order relation between the created instance input/output events and all input/output events that are above the create event on the creator instance. In other words, events that precede a create event also precede all events on the child

instance. The instance creation modifies the Event Order Table by adding more order relations. For illustration, see the example given in section 4.3.8.

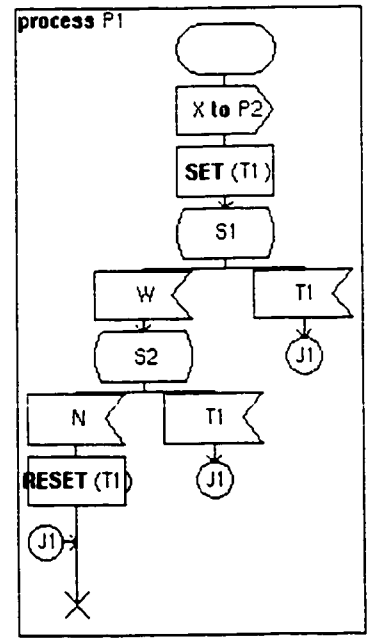
4.3.3 TIMERS

SDL timers are independent processes that run simultaneously with their owning instances. Timers, which are activated by a set signal, will stay activated as long as the current time does not exceed the setting time. When the current time reaches the setting time, the timer immediately sends a time-out signal to the input queue of the owning instance. If the time-out signal is discarded from the input queue, the timer will send another time-out signal. Timers become inactive upon the consumption of their time-out signal. Timers may also be deactivated by receiving a reset signal. This action (reset timer), will remove corresponding timer signals from the input queue of the owning instance. Consequently, in accordance to SDL after setting a timer, the owning instance has to expect a time-out signal (as shown in Figure 4.7) at the successive states until resetting the corresponding timer or consuming a corresponding time-out signal. On the other hand, the approach neglects the time-out expectation if the given bMSC specification does not specify the time-out event.

Furthermore, MSC timers may express a time delay, as shown in Figure 4.8.a. In this case, the approach issues an SDL input signal, with a new state, corresponding to the MSC time-out event. As shown in Figure 4.8.b, after setting timer *T2*, process *P1* will wait at state *S2* until timer expiration in order to progress.

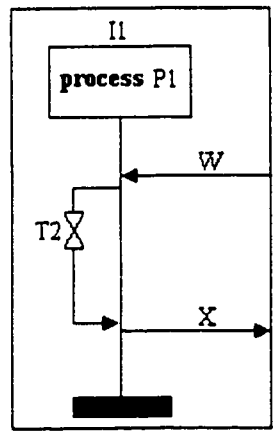


(a) bMSC specification.

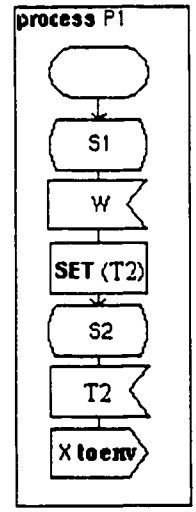


(b) Process P1 SDL behavior.

Figure 4.7. Timer events in bMSC.



(a) bMSC specification.



(b) Process P1 SDL behavior.

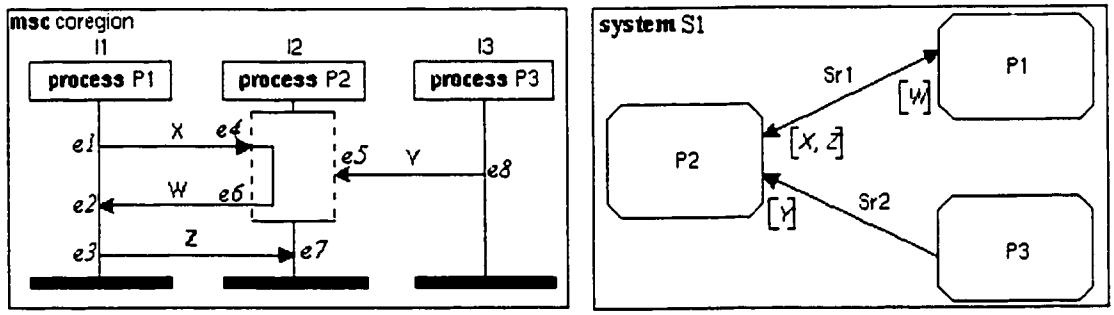
Figure 4.8. Time delay specification.

4.3.4 INLINE EXPRESSIONS

Because of the semantic similarities between inline expression operators and HMSC operators, we will discuss them in section 5.1.

4.3.5 COREGION

The order between events on the axis of an instance is preserved. To relax the order between events, coregion has been introduced to bMSC. Events inside coregion are unordered except if an order relation has been explicitly specified. In addition, the order of related timer events are preserved. For instance, events *e4* and *e6* in Figure 4.9 are ordered inside the coregion, but there is no order relations between event *e5* and both *e4* and *e6*.



(a) bMSC specification. (b) SDL architecture.

Figure 4.9. A coregion example: bMSC specification and target SDL architecture.

The approach completes the Event Order Table, as shown in Table 4.4, with all possible order relations caused by coregion in addition to other relations outside the coregion, while preserving any stated order relations in the form of MSC general order.

	e1	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*
e1		T	T	T	T	T	T	
e2			T				T	
e3							T	
e4		T	T		T	T	T	
e5		T	T	T		T	T	
e6		T	T		T		T	
e7								
e8		T	T	T	T	T	T	

Table 4.4. Event Order Table for the coregion example in Figure 4.9.

While events on an MSC instance axis create only one executable trace, events in coregions create many exceutable traces. Consequently, coregion input events may have different messages to save on each trace. To translate MSC coregion to SDL, the approach extracts all possible traces in a tree, as shown in Figure 4.10, called Coregion Tree. After creating the Occupancy Table (Table 4.5), the approach distributes the saved messages of coregion input events into the Coregion Tree.

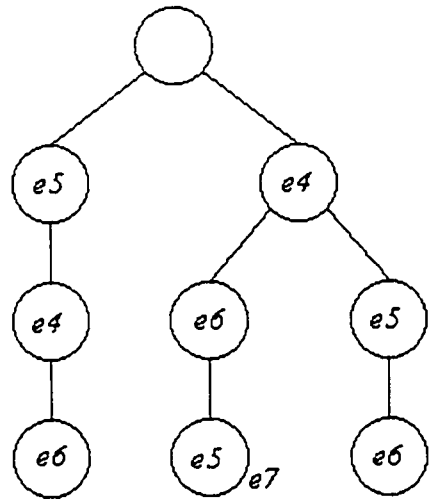


Figure 4.10.Coregion Tree for example in Figure 4.9.

To illustrate this, message *Y* (*e5*) has two saved messages *X* (*e4*) and *Z* (*e7*), as shown in Table 4.5. However, since *e4* is an alternative node to *e5* or precedes *e5* in the Coregion Tree, as shown in Figure 4.10, no save construct is needed for message *X* (*e4*). On the other hand, message *Z* (*e7*) is saved in the *e5* middle node in the Coregion Tree, while no save construct is made for message *Z* on the left and right nodes. This is because message *Z* (*e7*) cannot be expected before sending message *W* (*e6*), which follows *e5* left and right nodes, as shown in Figure 4.10.

input events for *P1*	input message	Route *Sr1*
e2	*W*	

input events for *P2*	input message	Route *Sr1*	Route *Sr2*
e4	*X*		*Y*
e5	*Y*	*X, Z*	
e7	*Z*		

Table 4.5. Occupancy Tables for the coregion example in Figure 4.9.

The generated SDL behavior of process *P2* is shown in Figure 4.11.

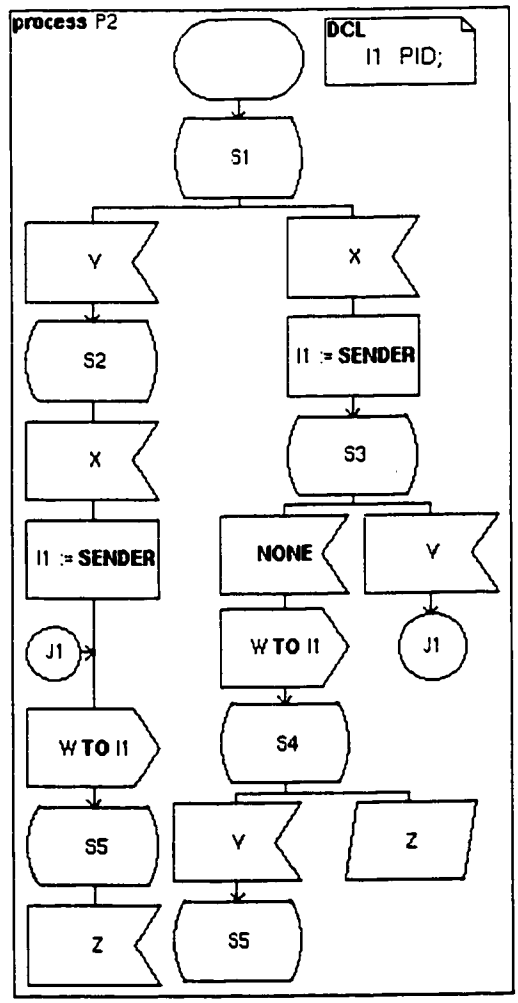


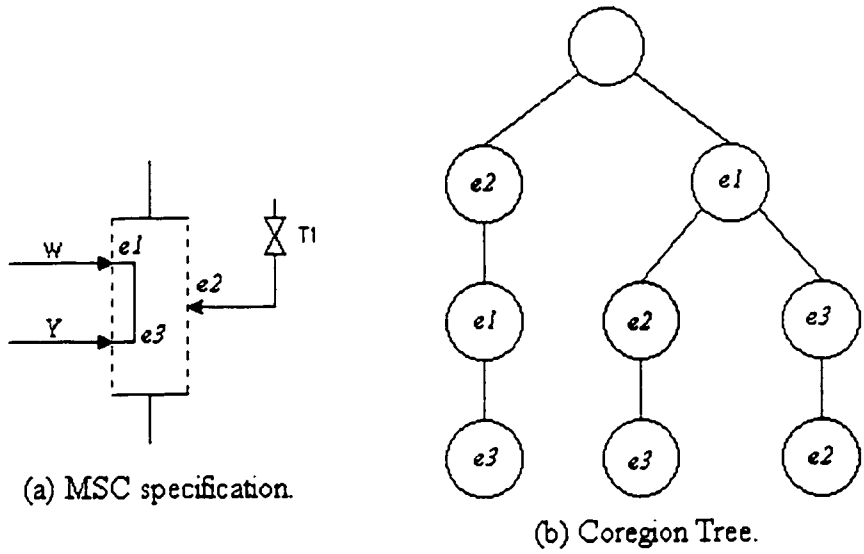
Figure 4.11.Generated SDL behavior for process P2 in Figure 4.9.

TIME-OUT WITHIN COREGION

MSC semantics for timer events are straightforward in that the setting event of a timer always precedes its corresponding reset and/or time-out event. The time-out, in MSC, can be used in the following two scenarios:

- To bypass some instance events. As an example: in the telephone systems, instead of waiting for a call confirm event from a non-responding callee forever, the caller can set the timer to bypass this event after the timer is expired (receiving a time-out event).

- To delay instance for a certain time. This happens by setting the timer with a desired delay duration and waiting for the timer expiration (time-out event arrival).



(a) MSC specification. (b) Coregion Tree.

Figure 4.12. Time-out event in coregion.

However, time-out events inside coregion, regardless of the corresponding setting event position, produce an ambiguous behavior (Figure 4.12) for the following reasons:

- If the timer has expired (especially if the timer has been set before coregion), the system cannot identify which time-out branch to take. This is shown in the above figure where there are three places (shadow nodes) that represent a time-out event (*e2*).

- MSC ITU-Recommendation only preserves the order of the timer events inside coregion. Including time-out event in coregion provides only one goal

which is to delay the instance exit from coregion. In other words, the instance will not leave the coregion at least until the timer is expired which can be reached by specifying the time-out event outside the coregion. To illustrate this, if we take the first level of the above tree, we find that the instance has to receive the message *X* (*e1*) or time-out message (*e2*). If we assume that *X* is not yet in the input queue and timer is expired, than the instance will proceed to the right branch of the tree where it will wait for message *X* again. As such, the instance cannot bypass message *X*. Further, the time-out event is distributed evenly throughout the tree, therefore, the instance has to wait for the time-out.

4.3.6 MESSAGE OVERTAKING

Message overtaking, as shown in Figure 4.13.a, can be described as two messages sent from one instance to another instance, but the receiving instance will consume them in the opposite order of the sending order. In the case where both messages are conveyed through the same route, as shown in in Figure 4.13.b, the basic approach does not issue a save construct for the earlier message. Consequenctly, a deadlock will arise in the generated behavior. To solve this problem, our approach checks for the existence of overtaking messages in the given bMSC specification, and makes a save construct for the earlier message for later consumption as shown in Figure 4.13.c.

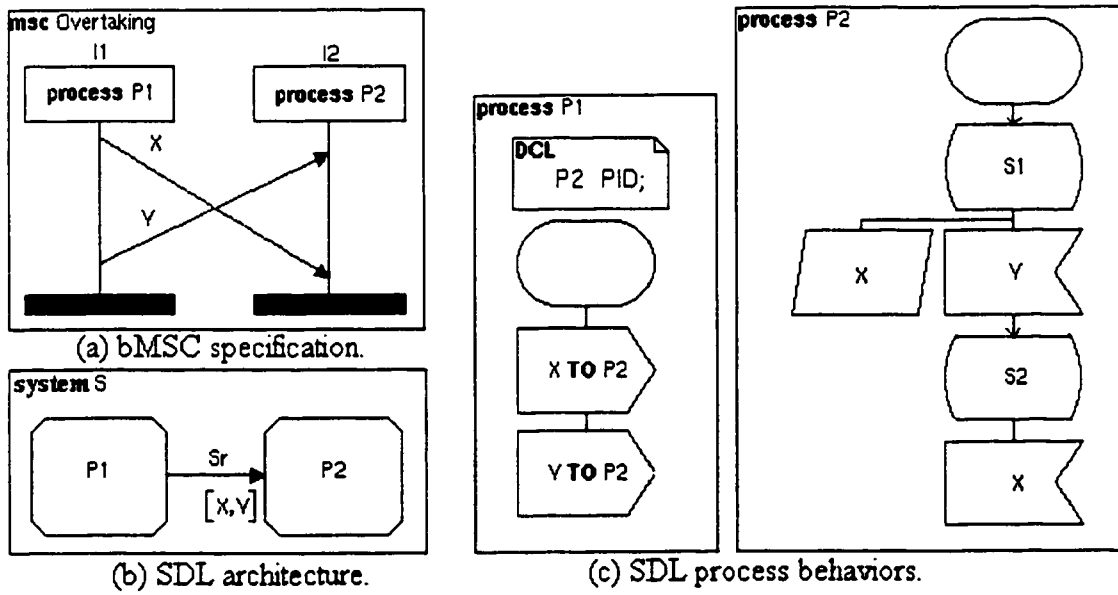


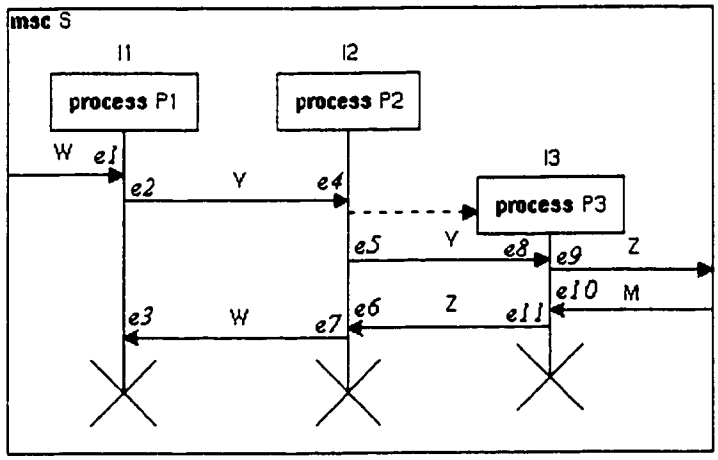
Figure 4.13.Overtaking messages.

4.3.7 ENVIRONMENT

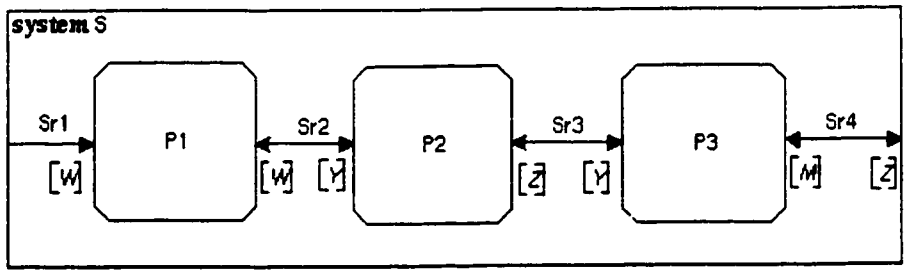
The approach assumes that the MSC environment consists of several independent instances with their independent behaviors. Moreover, no assumption is made about any order between the messages sent by these instances. Therefore, the approach generates a save construct for every message sent by the environment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

4.3.8 ENHANCED EVENT ORDER TABLE

The Event Order Table has been enhanced to include instance creation events as well as coregion events. The given MSC specification and SDL architecture in Figure 4.14 are used to demonstrate the manipulation of the Event Order Table.



(a) bMSC specification.



(b) SDL architecture.

Figure 4.14.Example for the enhanced Event Order Table.

To fill the Event Order Table for any input/output event, the approach uses the following criteria:

- An event precedes all events that follow it on the same instance axis except in coregion.
- Output events always precede their corresponding input events.
- An event precedes all events that are marked in the Event Order Table rows of any of the marked events in its row. For instance, since event *e7* is marked in *e5* row, as shown in Table 4.6, all marked events in *e7* row are marked in *e5* row, as shown in Table 4.7.
- Events, which precede an instance creation event, precede all events in the axis of the created instance.
- All possible order relations among coregion events are marked in the Event Order Table.

The calculation of Event Order Table may require several iterations for the third criterion, depending on the complexity of the bMSC specification. Table 4.6 shows the first iteration for the Event Order Table.

	e1	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*	*e9*	*e10*	*e11*
e1		**T**	T								
e2			T	**T**							
e3											
e4					**T**	T	**T**				
e5						T	**T**	**T**			
e6							**T**				
e7			T								
e8									T	T	**T**
e9										T	**T**
e10											**T**
e11						**T**					

Table 4.6. Event Order Table (first round) for example in Figure 4.14.

The second iteration is shown in Table 4.7, where the approach applies the third criterion on the Event Order Table shown in Table 4.6. For illustration purpose, cells having

caused the updating are printed in bolded font, cells, which are newly marked, are printed in italic font, and for newly marked cells which causes for a new iteration, we use bolded italic fonts.

	e1	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*	*e9*	*e10*	*e11*
e1		**T**	T	***T***							
e2			T	**T**	***T***	*T*	*T*				
e3											
e4			*T*		**T**	T	T	***T***			
e5			*T*			T	T	T	*T*	*T*	*T*
e6			*T*				T				
e7			T								
e8						***T***			T	T	**T**
e9						***T***				T	**T**
e10						***T***					**T**
e11			*T*			T	*T*				

Table 4.7. Event Order Table (second round) for example in Figure 4.14.

The Final Event Order Table is shown in Table 4.8 that is completed after the forth iteration.

	e1	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*	*e9*	*e10*	*e11*
e1		T	T	T	T	T	T	T	T	T	T
e2			T	T	T	T	T	T	T	T	T
e3											
e4			T		T	T	T	T	T	T	T
e5			T			T	T	T	T	T	T
e6			T				T				
e7			T								
e8			T			T	T		T	T	T
e9			T			T	T			T	T
e10			T			T	T				T
e11			T			T	T				

Table 4.8. Final Event Order Table (forth round) for example in Figure 4.14.

The generated SDL process behaviors are shown in Figure 4.15. In this example, after receiving signal *Y*, process *P2* saves the pid of sender process *P1* to use it later to send

signal *W*. Process *P2* also saves the pid of the created process *P3* because it will use it later to send signal *Y*. In the same manner, process *P3* saves the pid of sender process *P2* to use it later to send signal *Z*.

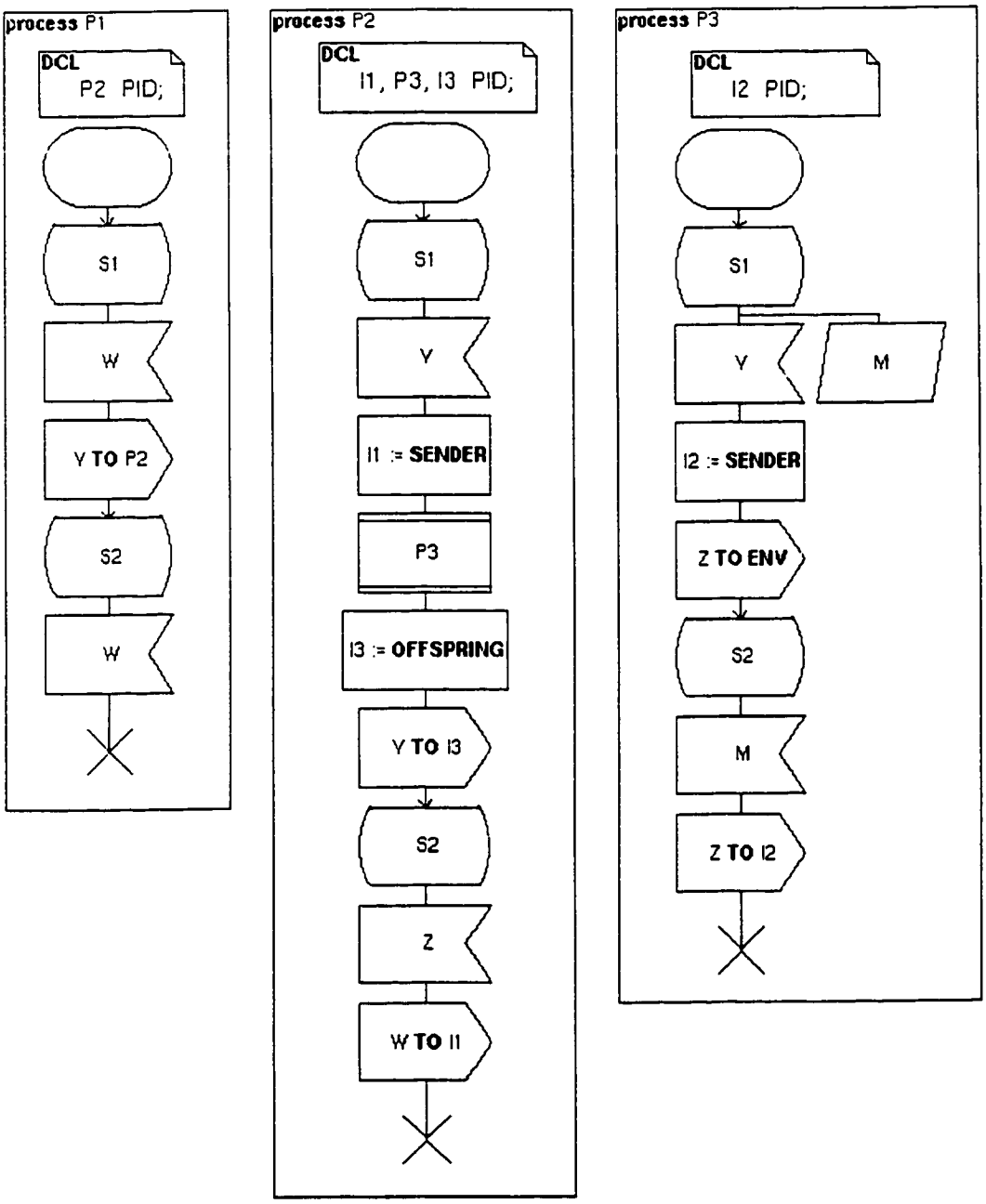


Figure 4.15.Generated SDL process behaviors for example in Figure 4.14.

4.3.9 bMSC ALGORITHM

In this section, the basic algorithm is rewritten to cover all previously discussed issues. Below is the general algorithm to generate an SDL specification from a bMSC specification.

1- **Check the architectural consistency between the given SDL and MSC.**
- For each process described in the MSC, there is a corresponding process type in the SDL architecture.
- Each message described in the MSC is enumerated in the SDL architecture by a route connecting the sending process and the receiving one.

2- **Number each input/output event uniquely.**

3- **Build the Event Order Table.**

```
For each instance I in the bMSC.
  For each input/output event E on the axis of instance I except events in
   coregion.
    Mark all of the following input/output events on the same I axis
     including events in coregion.
    If a create event exists below E.
      Mark all input /output events that exist on the created instance axis.
    EndIf
    If E is an output event.
      Mark the corresponding input event.
    EndIf
  EndFor
  If there are coregions on I axis.
    For each coregion area C on I axis.
      For each event E in C.
        Mark all events in C that have no order relation with E specified
         by the given bMSC.
      EndFor
    EndFor
  EndIf
EndFor
Repeat
  For each event row R1.
    For each event cell C1 in R1.
      Look for the correspond event row R2 of C1.
      For each event cell C2 in R2.
        If C2 is not marked in R1.
          Mark C2 in R1.
          The Event Order Table has been changed.
```

EndIf
EndFor
EndFor
EndFor
Until there is no change in the Event Order Table entries

4- **Create the Coregion Tree**

For each instance *I* in the bMSC.
If there are coregions on *I* axis.
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(reset 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

5- **Fill the Occupancy Tables**

For each instance *I* in the bMSC.
For each input event *E* on *I* axis, and corresponding output event *S*.
For each input event *Er* on *I* axis, and corresponding output event *Es*.
If ((*E* and *Er* does not have the same message name)AND((overtaking message)OR(NOT(*Er*≪*E*)AND(NOT (*E*≪*Es*)OR(*Es*=**env**))))))
Add message *Er* to *E* row in the Occupancy Table
EndIf
EndFor
EndFor
If a coregion exist on *I* axis.
Get the corresponding Coregion Tree.
Distribute the saved messages of coregion events among the tree nodes.
EndIf
EndFor

6- **Generate the SDL code**

```
For each instance I in the bMSC.
  Generate an SDL start node.
  For each event E on I axis.
    Case (E type) of
    Input event:
      Generate an SDL state construct.
      Generate an SDL input construct.
      If there are messages assigned to be saved
        Generate an SDL save construct for each one.
      EndIf
      If a timer has been active
        Generate an alternative SDL input construct.
      EndIf
      If a message will be sent later to the sender instance
        save the sender pid into an SDL variable.
        Generate an SDL task construct.
        Generate an SDL variable definition.
      EndIf
    Output event:
      If the pid address of the destination is known
        Use the pid variable to send the signal.
      Else
        Use the process name of the destination to send the signal.
      EndIf
      Generate an SDL output construct.
    Start of coregion:
      Translate the Coregion Tree.
      Go to the end of coregion region.
    Set timer event:
      Generate an SDL timer definition.
      Generate an SDL set timer construct.
    Time-out event:
      If timer used as a delay process
        Generate an SDL state construct.
        Generate an SDL input construct.
      EndIf
    otherwise:
      Generate the equivalent SDL construct.
    EndCase
  EndFor
EndFor
```

CHAPTER 5

GENERATION OF SDL SPECIFICATIONS FROM HMSCs

This chapter deals with HMSCs and multi-instances translation as well as MSC semantic errors.

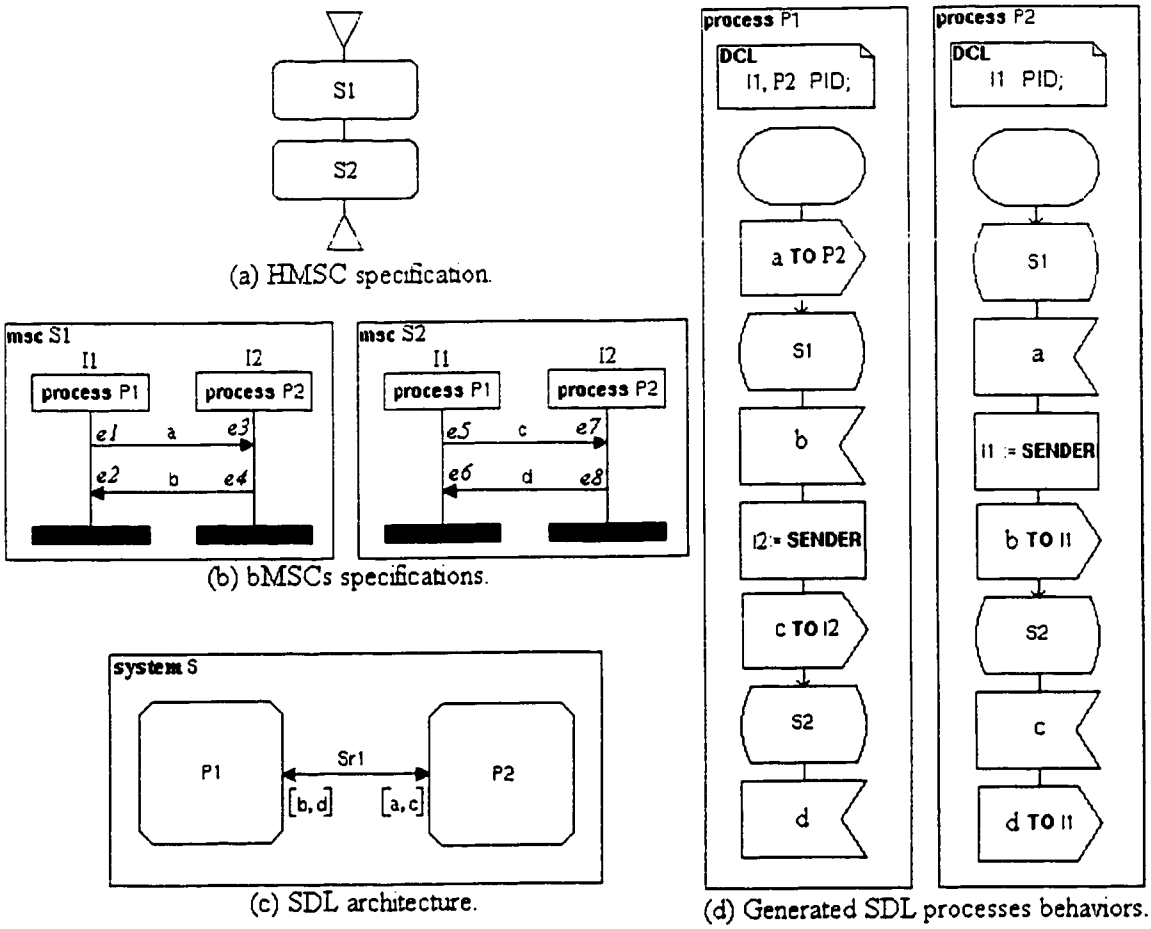
5.1 HMSC

HMSCs operators, sequential, iterative, parallel and alternative, are used to compose bMSCs in order to capture the complete system behavior. bMSC inline operator expressions, alternative, exception, optional, parallel and iterative, are used to define events composition within bMSCs. Because of the similarities among these operators, similar operators have been grouped and discussed together for the translation from MSC to SDL.

5.1.1 SEQUENTIAL OPERATOR

HMSC Sequential operator composes bMSCs in a sequential order. In fact, the bMSCs can be replaced by one bMSC that contains the entire scenario. The order relations between events in two different bMSCs can be determined by the order relation between the corresponding bMSCs. In general, if the two bMSCs have the same instances, then events of an instance in the preceding bMSC precedes all events of the same instance in the other bMSC. For the example shown in Figure 5.1, events *e1* and *e2*, precede events

e5 and *e6*, because events *e1*, *e2*, *e5* and *e6* are in the same instance axis (*I1*), and bMSC *S1* precedes bMSC *S2* in the sequential HMSC.



(a) HMSC specification.

(b) bMSCs specifications.

(c) SDL architecture.

(d) Generated SDL processes behaviors.

Figure 5.1. Sequential HMSC example.

Table 5.1 shows the Event Order Table for the sequential HMSC given in Figure 5.1.

	e1	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*
e1		T	T	T	T	T	T	T
e2					T	T	T	T
e3		T		T	T	T	T	T
e4		T			T	T	T	T
e5						T	T	T
e6								
e7						T		T
e8						T		

Table 5.1. Event Order Table for example in Figure 5.1.

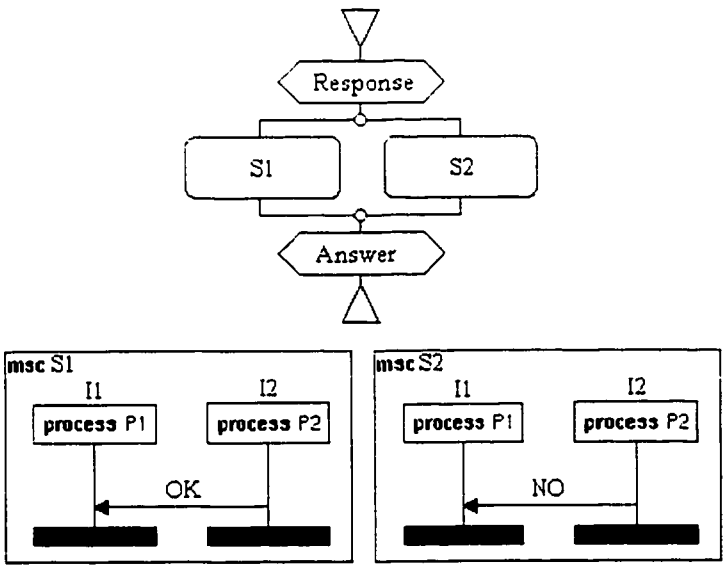
5.1.2 ALTERNATIVE OPERATORS

Alternative HMSC operators have the same semantics of alternative inline operators. They define alternative compositions between two scenarios or more. The exception inline operators have also the same semantics of alternative operators, but have only one scenario and the other scenario is represented by the rest of the bMSC specifications. Therefore, on each run, the system executes either the exception scenario or the rest of the bMSC specification. The optional inline operators have the same semantics of alternative operators, but have only one scenario while the other scenario is assumed empty. The system may or may not execute the optional scenario on each execution.

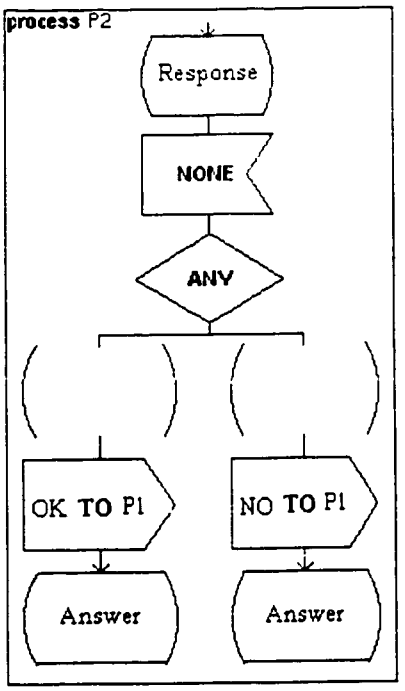
This group of operators can be translated into SDL states or SDL decisions depending on the initial MSC condition of the operator. Our approach searches for the initial condition in the following order:

- **Global Initial Condition**

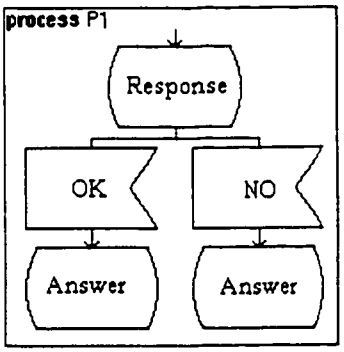
  If the operator is preceded by a bMSC global condition or HMSC condition (for HMSC operator only), then this condition will be the initial condition of the operator, as shown in Figure 5.2.a. Consequently, an SDL state is generated for each process specification in the alternative scenarios. Furthermore, the behaviors of the alternative scenarios of each instance are evaluated. Alternative instance specifications, which begin with an input message, are translated into SDL input construct, as shown in Figure 5.2.b. Otherwise, an SDL spontaneous transition will be generated followed by a non-deterministic SDL decision (if more than one non-input event is found), as shown in Figure 5.2.c.



(a) Alternative HMSC specification.



(c) Process P2 behavior.



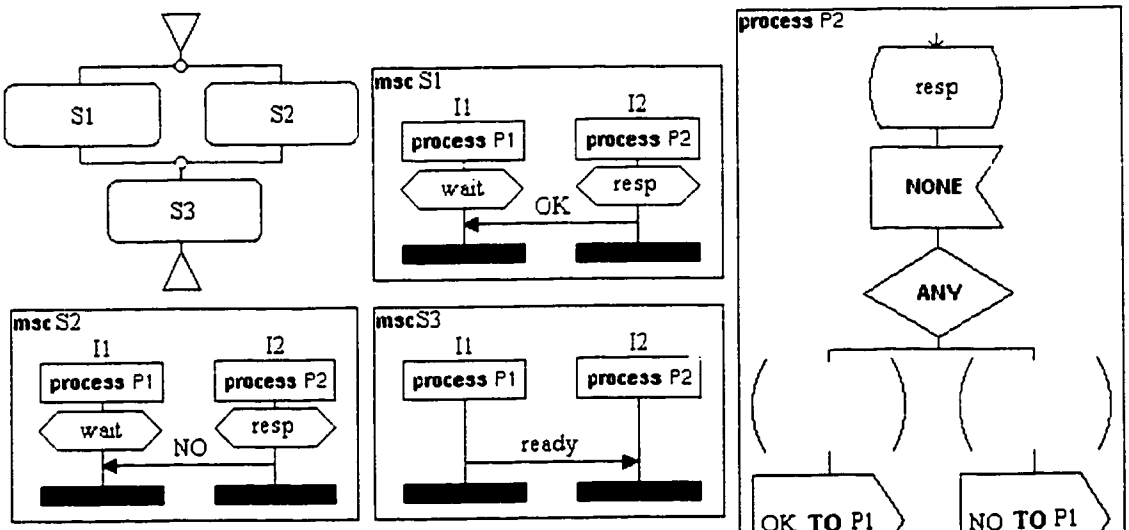
(b) Process P1 behavior.

Figure 5.2. Alternative HMSC example with global initial condition.

- **Local Initial Condition**

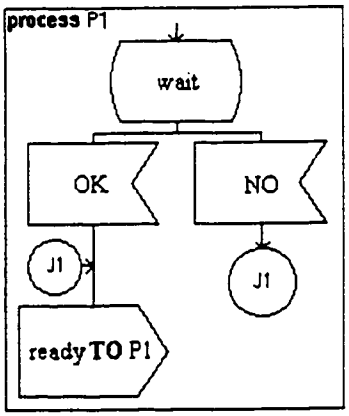
In the case where there is no global initial condition found, the approach analyzes the alternative behaviors of each instance separately. As shown in Figure 5.3.a, if a common local condition is found, then this condition will be the initial condition of this instance. Consequently, an SDL state is generated for the corresponding SDL process. Furthermore, as shown in Figure 5.3.b, alternative instance specifications, which begin with an input message, are translated into SDL input construct. Otherwise, an SDL

spontaneous transition will be generated followed by a non-deterministic SDL decision (if more than one non-input event is found), as shown in Figure 5.3.c.



(a) Alternative HMSC specification.

(c) Process P2 behavior.



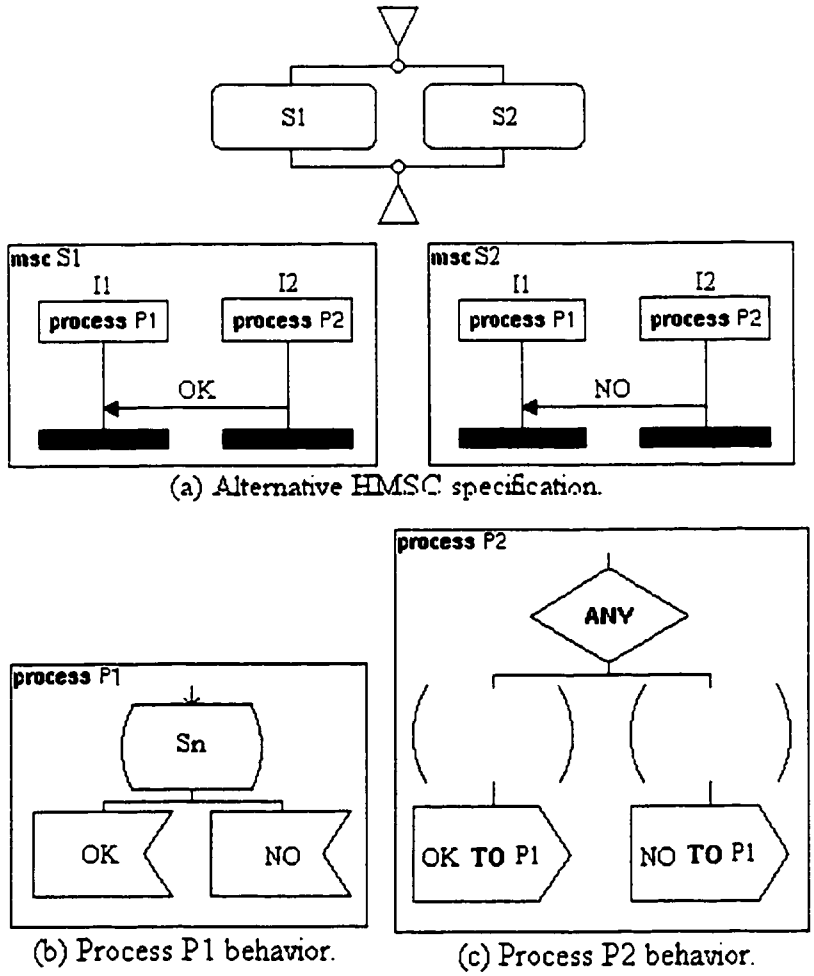
(b) Process P1 behavior.

Figure 5.3. Alternative HMSC example with local initial condition.

- **No Initial Condition**

If neither global initial condition nor local initial condition is found, as shown in Figure 5.4.a, the results of the previous analysis, done while searching for local initial condition, is used. If an MSC instance has alternative input messages, then an SDL state construct is generated and an

SDL input construct is generated for each input message, as shown in Figure 5.4.b. Otherwise, a non-deterministic SDL decision is generated and all alternative events are linked to this decision, as shown in Figure 5.4.c. In the case where the alternatives have mixed input and non-input events, then the approach alerts the user of a non-local choice problem (described in section 5.4.3).



(a) Alternative HMSC specification.

(b) Process P1 behavior.

(c) Process P2 behavior.

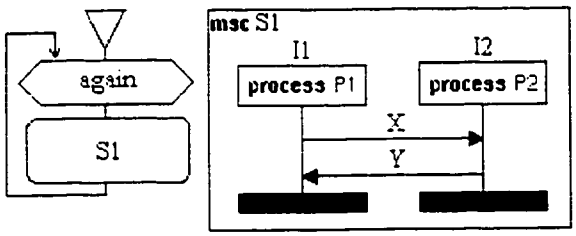
Figure 5.4. Alternative HMSC example with no initial condition.

Additionally, the final condition has to be determined, since all alternative scenarios have to be joined to return to the same point after the alternative operator. If the specification after the alternative operator begins with a global condition (as shown in Figure 5.2),

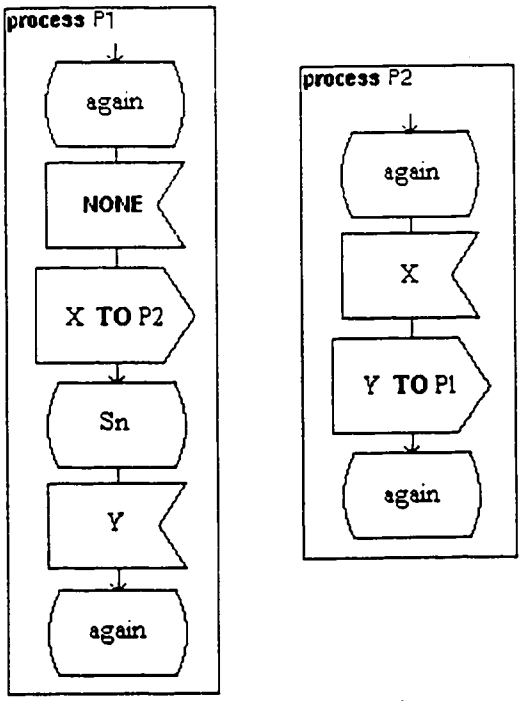
local condition, or input events (as shown in Figure 5.3.c), then all alternative scenarios are ended by SDL nextstate constructs. Otherwise, an SDL label is used to refer to the final condition (as shown in Figure 5.3.b). However, if there is no MSC specification after the alternative operator, then no further SDL construct is generated (as shown in Figure 5.4).

5.1.3 ITERATIVE OPERATORS

Translation of an iterative operator is similar to the translation of a sequential operator except that the initial states of the loop should be determined. The approach checks and saves the initial state of the iteration in order to refer to it at the end of the iteration. In case the initial state is neither condition nor input event, an SDL label is generated. If the last state is a condition, then nothing will be done. Otherwise, the approach generates an SDL nextstate construct (if the initial state is a condition or input event) or an SDL join construct. Figure 5.5 shows an example of iterative HMSC specification and the corresponding SDL process behaviors. In the given example, the initial state is HMSC condition *again*, and since the loop does not end with a condition, an SDL nextstate referring to the initial state *again* is issued.



(a) Iterative MSC specification.



(b) SDL process behaviors.

Figure 5.5. Iterative MSC example.

5.1.4 EVENT ORDER TABLE FOR HMSCs

The introduction of HMSC has made the order between MSC events - mainly sending and receiving events - more complex to trace and visualize. Moreover, loops need to be unfolded many times to get all order relations among the events in their corresponding loops. Furthermore, there is no order relation between events in alternative behaviors except when the alternative operator is inside a loop.

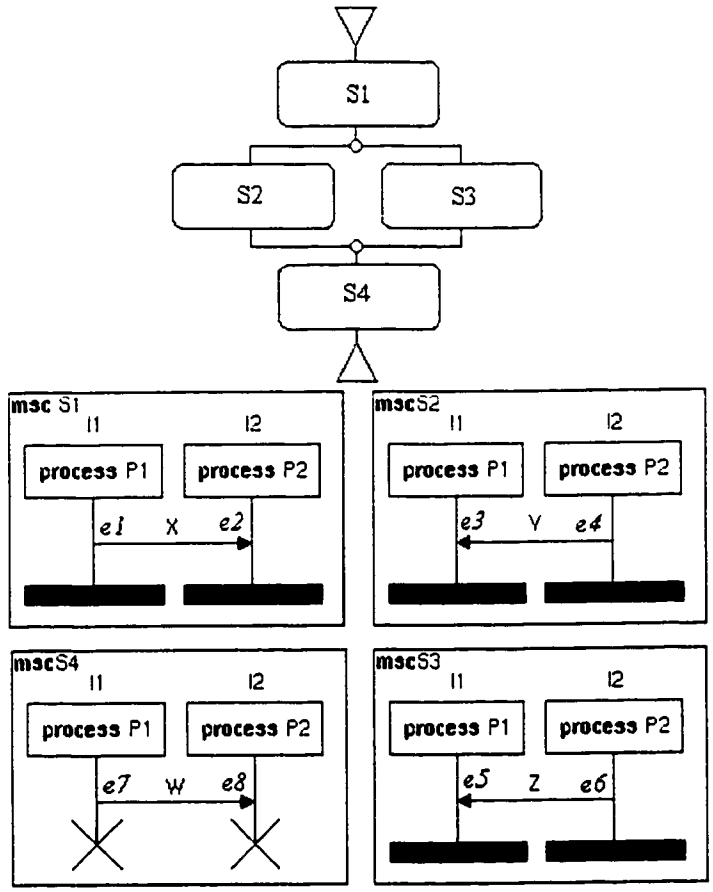


Figure 5.6. HMSC example for Event Order Table.

To create Event Order Tables for HMSCs, our approach first assigns a unique number to all input/output events in the given HMSC specification, as shown in Figure 5.6. Next, as shown in Table 5.2, the approach creates an individual Event Order Table of each bMSC in the HMSC by applying the bMSC Event Order Table algorithm, already mentioned in the previous chapter.

EOT1	*e1*	*e2*
e1		T
e2		

(a) bMSC S1.

EOT2	*e3*	*e4*
e3		
e4	T	

(b) bMSC S2.

EOT3	*e5*	*e6*
e5		
e6	T	

(c) bMSC S3.

EOT4	*e7*	*e8*
e7		T
e8		

(d) bMSC S4.

Table 5.2. Event Order Table (first step) for HMSC in Figure 5.6.

Then, the approach updates all events of an instance in the Event Order Table from events of the instance in the other Event Order Tables according to the order relations between their corresponding bMSCs. The columns of the Event Order Tables will be extended to manage all input/output events in the whole MSC specification. To illustrate our approach, we will update the first two tables (*EOT1* and *EOT2*).

Updating events *e1* and *e2* in *EOT1*.

Since bMSC *S1* precedes all other bMSCs, we will have:

Event *e1* precedes all events (*e3*, *e5* and *e7*) that relate to instance *I1* in bMSCs *S2*, *S3* and *S4*, consequently, their corresponding cells are marked. In addition, all marked cells (*e8*) in the rows of these events will also be marked.

Event *e2* precedes all events (*e4*, *e6* and *e8*) that relate to instance *I2* in bMSCs *S2*, *S3* and *S4*, consequently, their corresponding cells are

marked. In addition, all marked cells (*e3* and *e5*) in the rows of these events will be marked.

Table 5.3 shows the updated Event Order Tables *EOT1* and *EOT2*.

EOT1	*e1*	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*
e1		T	T		T		T	T
e2			T	T	T	T		T

(a) bMSC S1.

EOT2	*e1*	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*
e3							T	T
e4			T					T

(b) bMSC S2.

Table 5.3. Event Order Table (second step) for HMSC in Figure 5.6.

The next step will be updating each Event Order Table internally, i.e. events in the same Event Order Table will update each other. For instance, since event *e1* and *e2* in *EOT1* has been updated and *e1* has *e2* column marked, *e1* will be updated from *e2* as shown in Table 5.4.

EOT1	*e1*	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*
e1		T	T	T	T	T	T	T
e2			T	T	T	T		T

(a) bMSC S1.

EOT2	*e1*	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*
e3							T	T
e4			T				T	T

(b) bMSC S2.

Table 5.4. Event Order Table (third step) for HMSC in Figure 5.6.

The last two steps are repeated as long as there is at least an updated Event Order Table. After the last round, all Event Order Tables are joined to create one Event Order Table as shown in Table 5.5.

	e1	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*
e1		T	T	T	T	T	T	T
e2			T	T	T	T	T	T
e3							T	T
e4			T				T	T
e5							T	T
e6					T		T	T
e7								T
e8								

Table 5.5. Event Order Table (final step) for HMSC in Figure 5.6.

5.1.5 PARALLEL OPERATORS

Due to time limitations for our research, we did not investigate parallel compositions, and shall leave it for future investigation.

5.2 MULTI-INSTANCES

MSC specifications may visually describe the behavior of multi-instances of the same process. However, SDL specifications describe the total behavior of a process, which may lead to one or more instance behaviors. Therefore, to generate SDL process behavior from MSC multi-instances, the multi-instance behaviors need to be compared and merged. Comparing multi-instances within HMSCs is a complicated task, since instances can be initiated in different bMSCs, and instances can be discontinued where in some bMSCs they are not described. Because of the comparison, identical instances will be replaced with only one SDL process behavior, and different instances will be merged as

alternative SDL process behavior. The result of merging instances may produce a non-local choice, which may not be implemented in SDL.

5.2.1 MERGING IDENTICAL MULTI-INSTANCES

Identical multi-instances inherit the behavior of their corresponding process, as shown in Figure 5.7 instance *I1* and instance *I3* are identical instances of process *P1*. Consequently, only one instance of an MSC specification is translated into an SDL process behavior, as shown in Figure 5.8. The specifications of the other identical instances will be ignored. Furthermore, to ensure the creation of the instances, the corresponding SDL process definition should be modified to accommodate the number of instances that has been specified in the MSC specification.

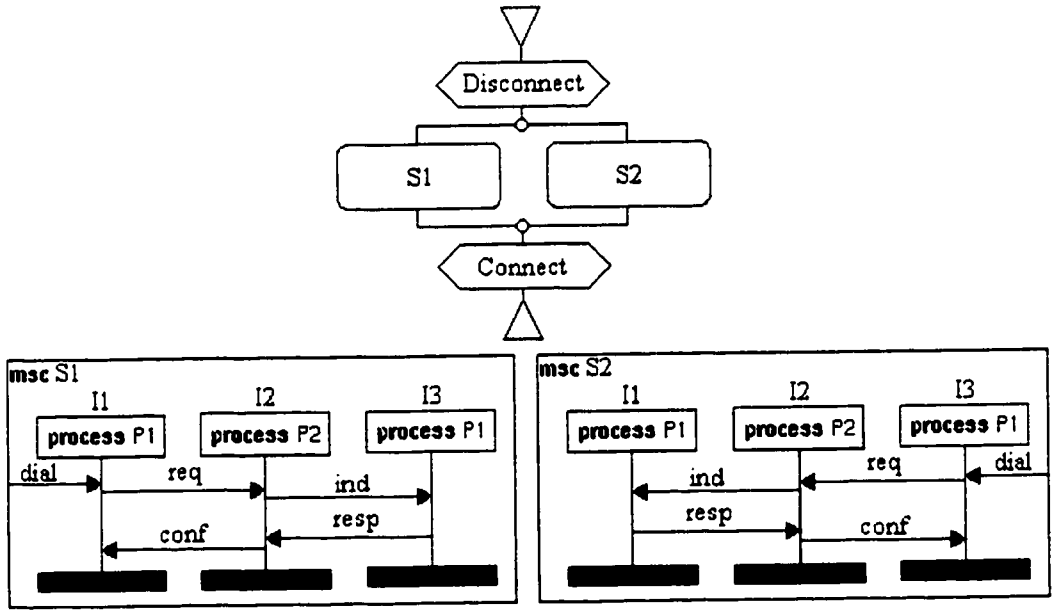


Figure 5.7. Identical multi-instance MSC specification.

5.2.2 MERGING DIFFERENT MULTI-INSTANCES

SDL process specification has only one start node, and each created instance has to interpret this node to start its life. SDL process behavior may be composed of only one

instance behavior, or of several instance behaviors. This leads us to the question, how one can distinguish between these instance behaviors? Behaviors of multi-instances should be checked against each other to find the divergence points between the different instances. These divergence points can be caused by an internal process decision, or by an external process decision.

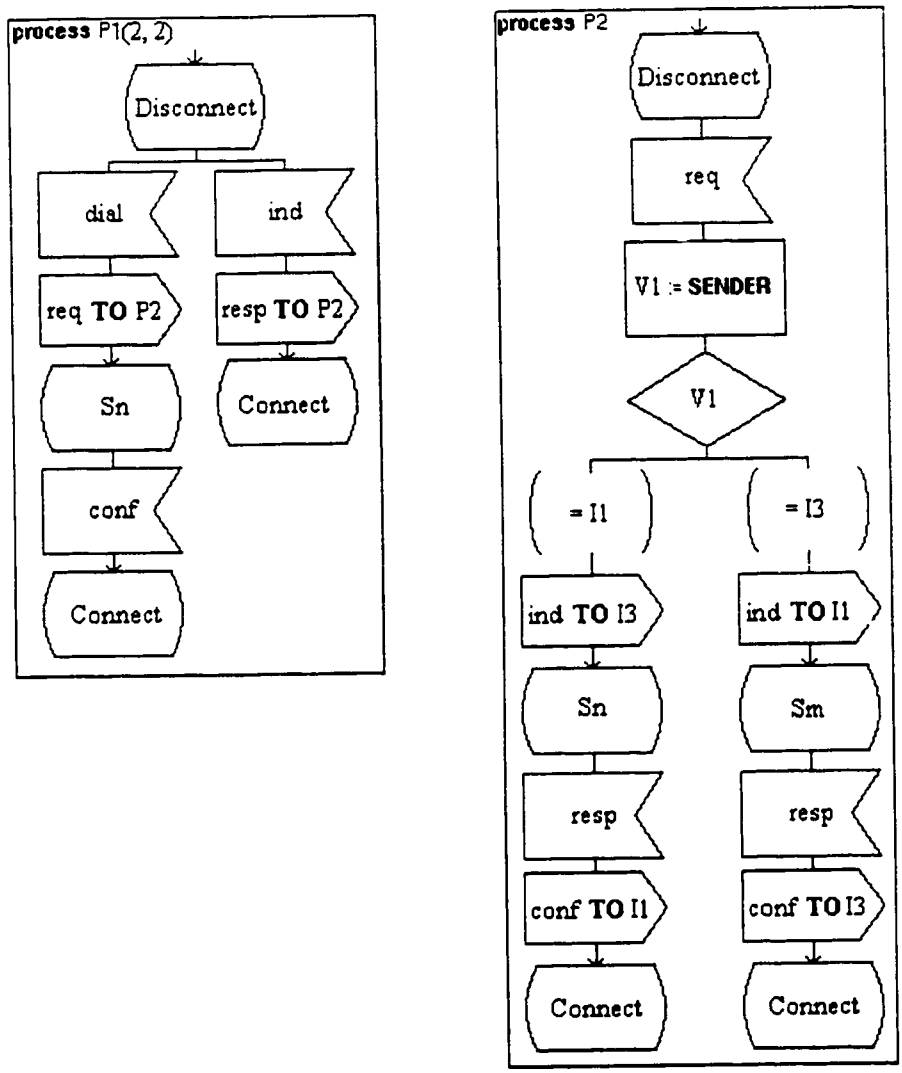
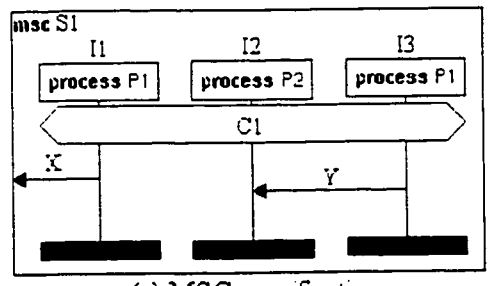


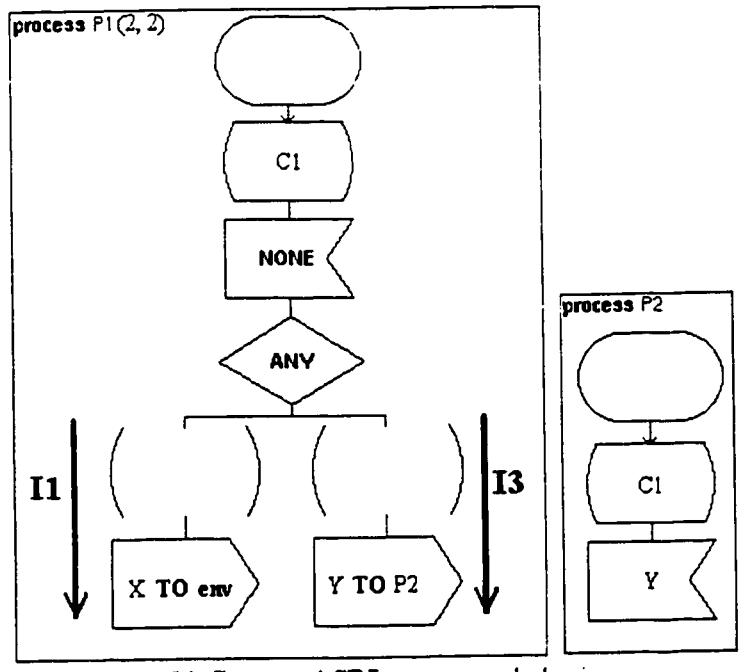
Figure 5.8. Generated SDL process behaviors for MSC specification in Figure 5.7.

5.2.2.1 INTERNAL PROCESS DECISION

The internal decision will generate a non-deterministic state, which may lead to non-intended behavior that is not specified in the corresponding MSC specification. To illustrate this case, process *P1* in Figure 5.9 has two instances *I1* and *I3*. According to the MSC specification, *I1* begins its life by sending message *X*, and *I3* begins by sending message *Y*. However, when instances *I1* and *I3* of the SDL process P1 are created, they interpret the start node. They will then randomly select to send signal *X* or signal *Y*. There is no guarantee that the generated SDL behavior will be consistent with the given MSC specification, since there is a high probability that the created SDL instances will take the same behavior (either *I1* or *I3*). However, the approach generates the SDL behavior and alerts the user, since the initialization part is sometimes handled in a later phase of the software process.



(a) MSC specification.

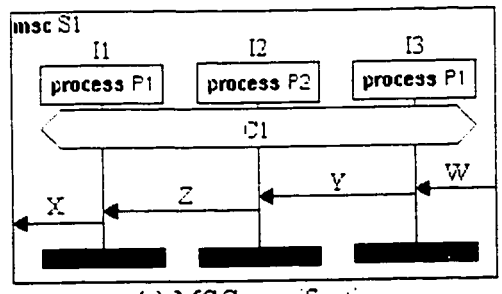


(b) Generated SDL processes behavior.

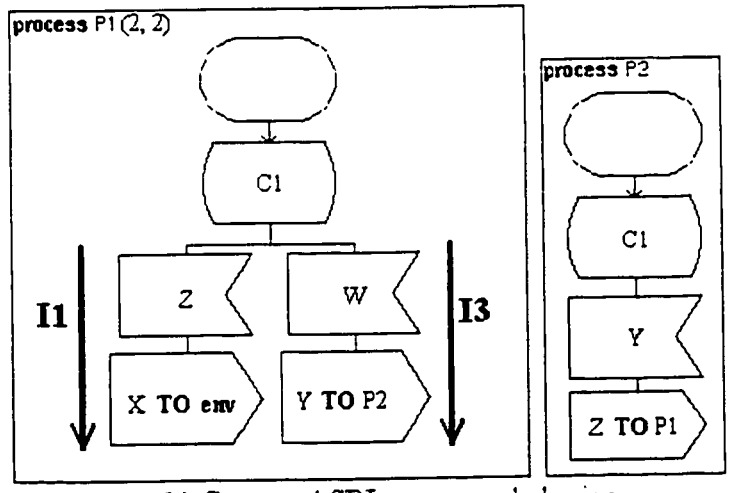
Figure 5.9. An example for internal process decision.

5.2.2.2 EXTERNAL PROCESS DECISION

Instances with external decision will be controlled by other instances messages. Upon consuming a message, the instance will identify which behavior to follow. As shown in Figure 5.10, instances of SDL process *P1* will wait for receiving either signal *X* or signal *Z* to continue their execution. Unfortunately, the instance identification problem, which has been addressed in section 4.3.1, can occur. This is caused by the fact that during the system startup neither the environment nor instance *I2* has the pids of instances *I1* and *I3*. Consequently, instances *I1* and *I3* will be implicitly addressed, and signals *X* and *Z* may be received by only one instance (*I1* or *I3*).



(a) MSC specification.



(b) Generated SDL processes behavior.

Figure 5.10. An example for external process decision.

5.3 SHARED CONDITIONS

A condition name may be used repeatedly within a MSC instance specification. This repetition states an alternative composition between specifications below the shared condition. As shown in Figure 5.7, condition *Disconnect* is used to identify the beginning of bMSCs *S1* and *S2*. Furthermore, same condition name may be shared by multi-instance specifications to identify similarities among their specifications, as shown in Figure 5.7 where condition *Disconnect* is shared by instances *I1* and *I3* of process *P1*.

Consequently, since MSC conditions are equivalent to SDL states, the MSC specifications below the shared condition are merged by the corresponding SDL state. These specifications may be identical as in the case between specification of instance *I1*

in bMSC *S1* and specification of instance *I2* in bMSC *S2*. As a result, only one specification will be translated, as shown in the generated behavior of process *P1* in Figure 5.8. On the other hand, specifications below the shared condition may be different and will be merged as alternatives under the generated SDL state. As shown in Figure 5.8, instance *I1* has different specifications in bMSC *S1* and bMSC *S2* that share the same condition *Disconnect*.

Shared conditions may have the same MSC constructs but different semantic specifications. As shown in Figure 5.7, shared condition *Disconnect* of process *P2* has the same MSC constructs (input *req*, output *ind*, input *resp* and output *conf*) in both bMSC *S1* and *S2*. However, the messages' sources and destinations are different. Our approach detects this case and generates the appropriate SDL behavior to distinguish between the sources, as shown in Figure 5.8. Unfortunately, this case may raise the implementability issue (discussed later), if one of the distinguishable sources pid (in this case the pids of instances *I1* and *I3*) are unknown at the current SDL state.

5.4 MSCs SEMANTIC ERRORS

The specification of telecommunications systems should satisfy some general properties such as the absence of non-local choice, deadlocks, etc. in order to ensure correctness. To ensure that our generated SDL specification satisfies these properties, the approach checks the given MSC specification against these properties. This section describes our techniques for detecting some of the MSC semantic errors.

5.4.1 DEADLOCK

Our approach works efficiently when the MSC specification and SDL architecture are free of syntax and semantic errors. However, as in the tool presented in [7], the approach detects semantic errors such as deadlocks. A deadlock can be defined as a system state where all the queues in the system are empty and there is no possible progress for any process. In Figure 5.11, the system cannot progress because each instance (*I1* and *I2*) is waiting to receive a signal (*X* and *Y* respectively) from the other instance. The detection of deadlocks is done on fly during the generation of the Event Order Table. The approach checks for an event entry that refers to itself in the Event Order Table which should only be caused by unfolded loops. For instance, bold cells in the generated Event Order Table (as shown in Table 5.6) are an indication of a deadlock. As soon as the approach encounters a deadlock, the user is prompted and the translation is cancelled.

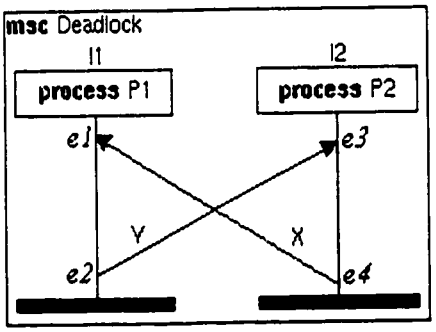


Figure 5.11.Deadlock example.

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

Table 5.6. Event Order Table for the deadlock example in Figure 5.11.

5.4.2 PROCESS DIVERGENCE

During system execution, instances run with equal priorities. Depending on the underlying architecture and the speed of processes, some instances may run faster than others. When the receiver instance cannot cope with the speed of the sender instance, and there is no handshaking mechanism between the two, process divergence may happen. Process divergence can be present in loop specifications. For instance, in Figure 5.12, if *I1* is faster than *I2*, *I1* can send several messages before *I2* can consume even the first message. Consequently, after a while the input queue of *I2* will be flooded. To overcome the process divergence error, instances should have some kind of handshaking protocols inside iterative operators. This handshaking can be achieved indirectly through other parties.

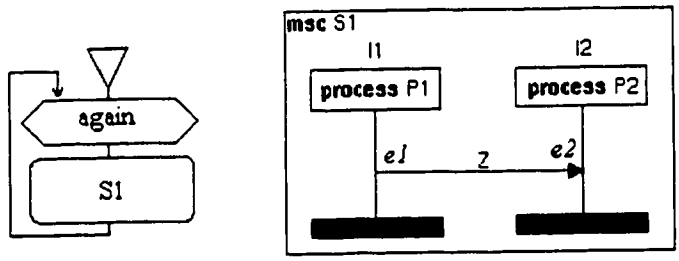


Figure 5.12.Process divergence.

Our approach detects process divergence by unfolding the loop one time, and updating the Event Order Table. Next, all input events in the loop are checked to see whether or not their corresponding output events are in their Event Order Table rows. A process divergence may exist, if an input event does not have the corresponding output event cell marked in the Event Order Table, as shown in Table 5.7 the corresponding cell of event *e1* is not marked in event *e2* row. In this situation, the user will be alerted of the presence of process divergence.

	e1	*e2*
e1	T	T
e2		T

Table 5.7. Event Order Table for divergence situation.

5.4.3 NON-LOCAL CHOICE

The introduction of alternative operators may cause a non-local choice. The semantics of MSC states that the first instance to reach the alternative is in charge of picking one of the alternative scenarios, and all other instances must follow the same choice. However, this semantics cannot be forced in distributed systems.

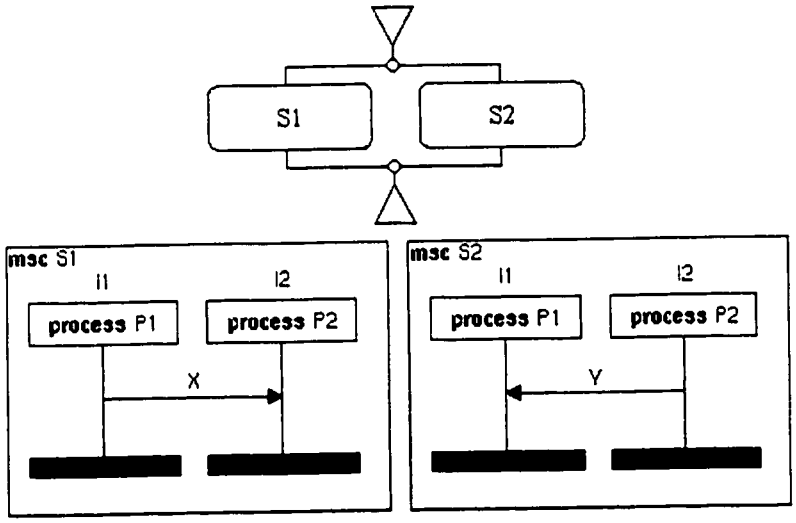


Figure 5.13.Non-local choice.

Figure 5.13 shows a simple non-local choice scenario. The specification says that if *I1* reaches the alternatives first and chooses to send message *X*, *I2* must take the same bMSC *S1* and consume message *X*, and vise versa. Nevertheless, there is no synchronization between the two instances, while I1 executing bMSC *S1* and sending message *X*, *I2* may decide to execute bMSC *S2* and sends message *Y*.

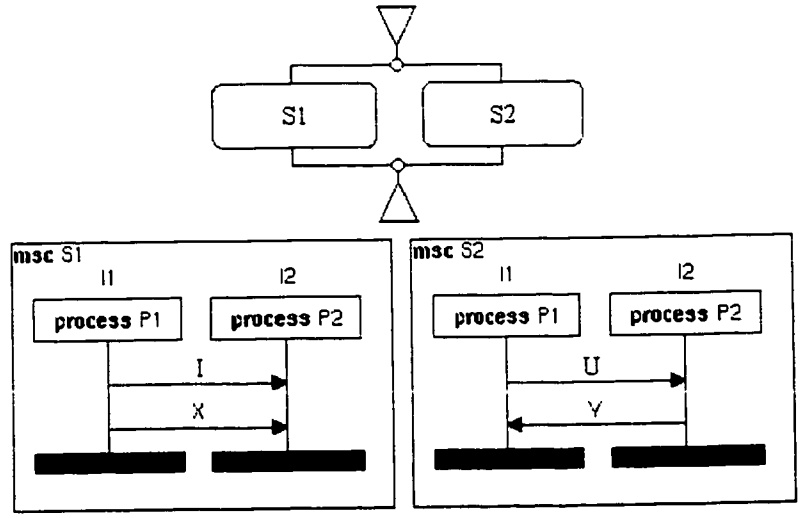


Figure 5.14.Synchronization between Instances.

Figure 5.14 shows a solution for the non-local choice presented in Figure 5.13. As illustrated in this solution, *I1* is in charge of picking one of the alternatives, while *I2* has to wait for the decision made by *I1* and follow the same bMSC.

Non-local choice can be more complex and harder to detect, visually, within nested alternative bMSCs. Let us follow the scenario in Figure 5.15, the choice between bMSCs *S1* and *S2* controlled by instance *I1*, and the choice between bMSCs *S3* and *S4* controlled by instance *I3*. One can conclude that independently, each alternative is free of non-local choice. However, there is a non-local choice within the whole MSC. The MSC semantics states that system should execute bMSC *S1* only or bMSC *S2* then bMSC *S3* or bMSC *S4*. But, since *I1* and *I3* are independent instances, and run in parallel, *I3* may decide to execute bMSC *S3* or bMSC *S4*, while *I1* is executing bMSC *S1*.

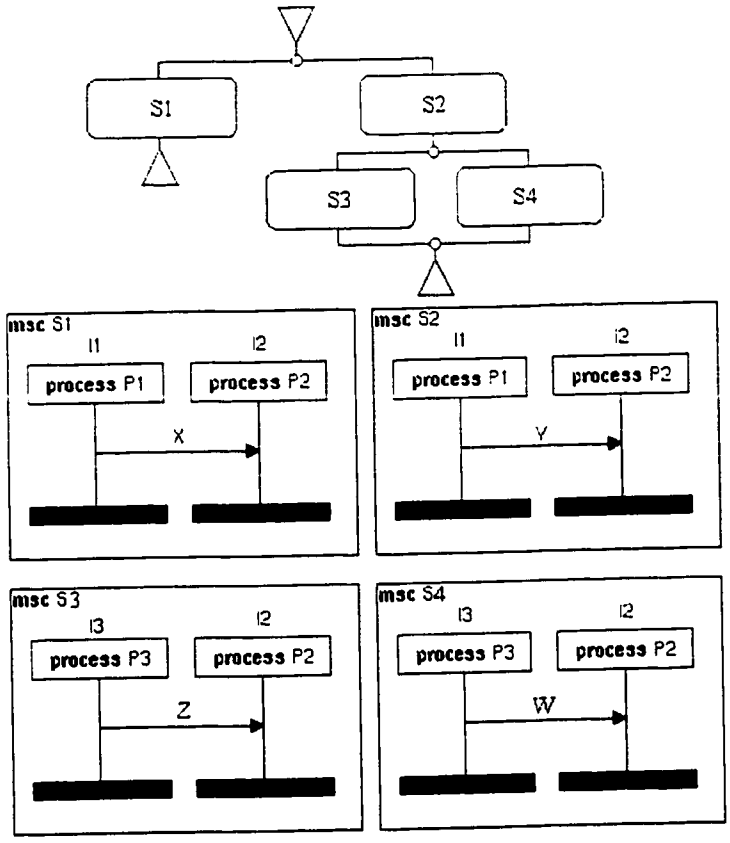


Figure 5.15.Nested non-local choice.

To detect the presence of non-local choice, our approach uses the following criterion:

- At least one alternative bMSC is controlled by more than one instance.
- bMSCs of the same alternative are controlled by different instances. As shown in Figure 5.13, while bMSC *S1* is controlled by instance *I1*, bMSC *S2* is controlled by instance *I2*.
- In case of nested alternatives, two criterion 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. For instance in Figure 5.15, the lower alternative is controlled by instance *I3*, while the higher alternative is controlled by instance *I1*.
  - The controller instance of the lower alternative is not depending on the controller instance of the higher alternative. To illustrate this criteria, instance *I3* is not depending on instance *I1* in Figure 5.15. However, the non-local choice can be avoided if instance *I3* receives a message from instance *I1* or instance *I2* before the lower alternative.

5.4.4 UNSPECIFIED RECEPTION

Our approach also detects the unspecified reception, which is the other semantic error. The unspecified reception error can be stated as being for an instance while waiting for receiving a message in the input queue, it receives an unexpected message. In Figure 5.16, choosing one of the alternative bMSCs is decided by instance *I1*. As soon as *I1* executes one of the bMSCs (*S1* or *S2*) all other instances must follow the same choice and execute the same bMSC. Let us assume that instance *I1* has executed the following sequence bMSC *S1* then bMSC *S2*, Consequently, instances *I2*, *I3* and *I4* have to follow the same sequence. Since instances *I3* and *I4* have only been specified in one side of the alternatives, they will follow that side. However, after extracting all possible messages interleaves for this scenario, as shown in Figure 5.17, there is a chance that instance *I2* executes bMSC *S2* before executing bMSC *S1*. By following the bolded executable trace in Figure 5.17, it may receive (unexpected) message *Z* while instance *I2* is waiting for message *Y*.

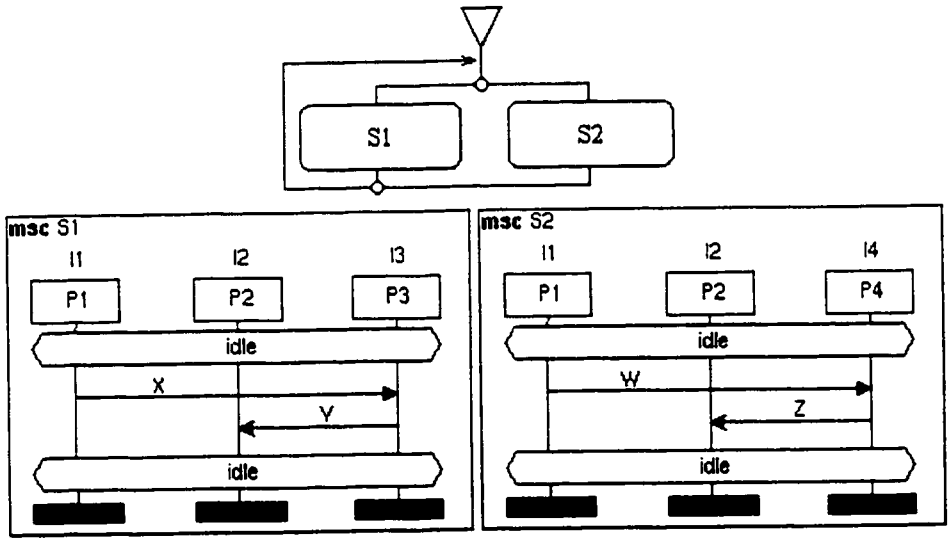


Figure 5.16. Unspecified reception.

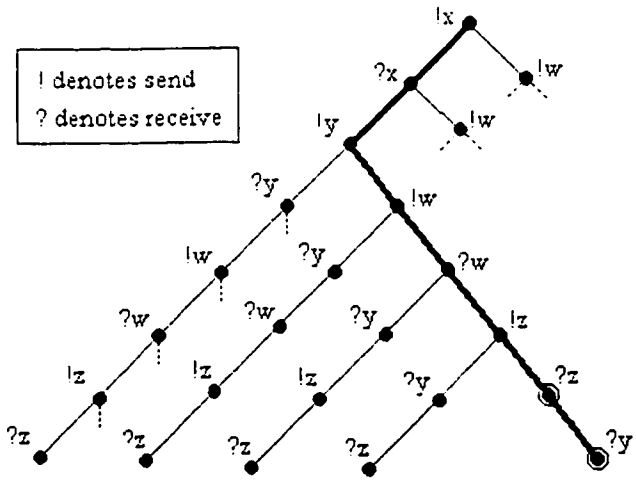


Figure 5.17.Trace diagram for unspecified reception example.

We characterize this problem if an instance has at least two alternative specifications that start with a message consumption. Consequently, we check the messages interleaving of the execution of these two alternative bMSCs, and verify if there is a trace that does not satisfy the MSC specification.

5.4.5 IMPLEMENTABILITY OF MSCS UNDER TARGET ARCHITECTURE

The implementability issue has been discussed in [6]. Depending on the given SDL architecture, MSC specifications may be implementable or non-implementable. Messages that control the flow of the instance behaviors are of particular interest in regards to this approach. Since these messages effect the behavior of the system, there should be no ambiguity or overlapping among them. The implementability issue can arise at alternative operators, where more than one controlled message is required to distinguish between the alternative scenarios as shown in Figure 5.18. The controlling messages in this figure are *a* and *b*, and depending on the reception order of the messages, instance *I2* will execute bMSC *send_a_b* or bMSC *send_b_a*. In this case, the order of the controlled messages is crucial, and depends on the given target SDL architecture. If the given target architecture

allows messages (*a* and *b*) to only travel on the same route as shown in Figure 5.19.a, then the messages will be received in the input queue of process P2 in the same order. However, if the given target architecture allows the messages to travel on two different routes, as shown in Figure 5.19.b, then the order between *a* and *b* cannot be predicted. Consequently, process P2 cannot figure out which alternative process P1 has taken.

Our detection approach is on the fly, while translating MSC specification into SDL specification. The approach checks, at every generated SDL state, if any of the controlling messages requires saving by the other controlling messages. To illustrate our detection technique, Figure 5.19.c and Figure 5.19.d show the generated SDL behavior for process *P2* under two different SDL system architectures *S1* and *S2*. In the first figure, there is no need to save both messages *a* and *b* because they are conveyed on the same route *Sr*. In the second figure, the save construct is required since the two messages come through different routes (*Sr1*, *Sr2*). The approach alerts the user with the error and terminates the translation.

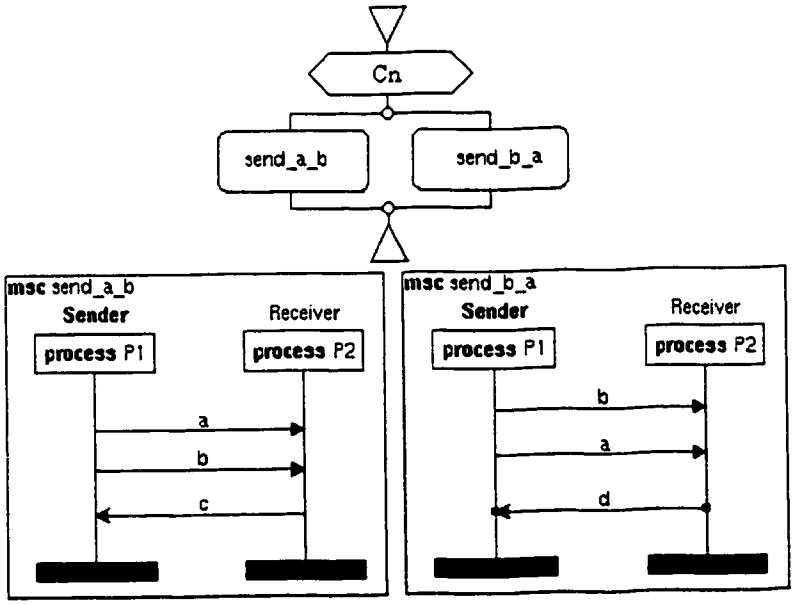


Figure 5.18. An example: non-implementable MSC specification under some SDL architectures.

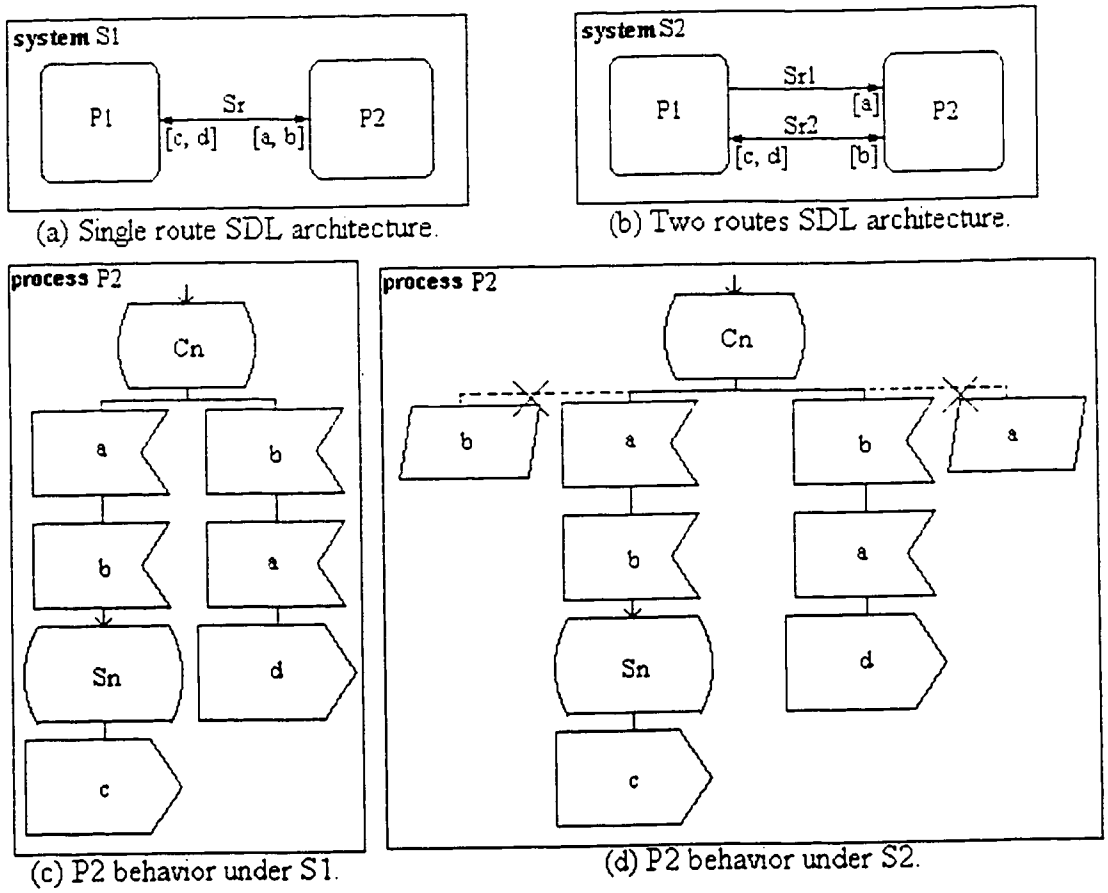


Figure 5.19.Generated SDL process behaviors for the non-implementable MSC example in Figure 5.18.

5.5 THE COMPLETE ALGORITHM

The following algorithm generates SDL process behaviors from MSC specifications for the given SDL architecture.

1- **Check the architectural consistency between the given SDL and MSC.**
   - For each process described in the MSC, there is a corresponding process type in the SDL architecture.
   - Each message described in the MSC is enumerated in the SDL architecture by a route connecting the sending process and the receiving one.

2- **Number each input/output event uniquely.**

3- **Check the presence of non-local choice.**

4- **Check the presence of unspecified reception.**

5- **Build the Event Order Table.**

Build Event Order Table for each bMSC in the given specification as has been described in bMSC algorithm.

```
Repeat
  For each Event Order Table EOT1, and corresponds to bMSC bmsc1.
    For each event row R1 in EOT1, and belongs to instance I1.
      For each Event Order Table EOT2, and corresponds to bMSC
       bmsc2.
        If bmsc1 precedes bmsc2.
          For each event row R2 in EOT2, and belongs to instance I2.
            If I1 equal I2.
              EOT1[R1,R2] = T
              /* X[A,B]=T ⇒ mark the cell at the row A and the Column B in
                Table X */
                For each cell C that is marked in row R2
                  EOT1[R1,C]=T
                EndFor
              The Event Order Table has been changed.
            EndIf
          EndFor
        EndIf
      EndFor
    EndFor
  EndFor
Until there is no change in the Event Order Table entries
Merge all Event Order Tables.
```

6- **Create the Coregion Tree**

As described in bMSC algorithm.

7- **Fill the Occupancy Tables**

As described in bMSC algorithm.

8- **Check the existence of multi-instances**

Check the presence of multi-instances. If there are multi-instances of a process, compare the specification and merge alternative instances. During the comparison, check for the divergence points and alert the user of any ambiguities.

9- **Generate SDL code.**

Same as for bMSC algorithm.

5.6 DISCUSSION

In this chapter, we have presented our approach for generating SDL specifications from MSCs, under a given SDL architecture. Our approach ensures high quality SDL specification, free of any design error. Moreover, in our approach we check the correctness of the given MSC and its implementability.

Recently, during the last year, many research groups have started similar projects [12] [19] [16]. In this section, we describe briefly each one of them.

In his thesis, [12] is mainly concerned with extracting SDL architecture specification from the given MSC specification. The approach searches the MSC instance headers for architectural keywords such as block, system and process. According to the given information, the main objects of SDL architecture are constructed. Then, the approach creates a single route, signal route/channel, between each two communicated processes to convey their signals. In his thesis report, our work was mentioned as a complement to his work.

In [19] approach, MSC processes have a parallel behavior. Therefore, a generated SDL process behavior composed of only one state and all input signals are initiated from this state. Furthermore, the approach is more interested in measuring the performance of the specification than translation of MSC into SDL.

In contrast to our approach, the main characteristic of the ongoing research [16] is the generation of the SDL architecture in addition to the SDL process behavior. However, since MSC specifications are gathered during the requirement phase, it is too early to decide the target system architecture. In addition, the MSC specification becomes more complicated by the enriched information required by the approach. Moreover, most of the

requirement specifications may target different hardware, and that may lead to an earlier redundancy of the specified work. Consequently, at the requirement phase, the specifications lose their properties of reusability and maintainability. Furthermore, most of the time, the target hardware (architecture) is already known and the MSC specification is an additional or updating feature to the current specification, for instance adding new features to telephone systems.

CHAPTER 6

THE MSC2SDL TOOL AND APPLICATIONS

6.1 OVERVIEW

Our approach has been implemented on a unix platform. It can be easily ported to other platforms such as DOS, Windows and Linux. The tool, MSC2SDL, has been implemented using the C language [15]. The user interface is built with the Tcl/Tk[2] scripting language.

The MSC2SDL tool has been interfaced with ObjectGEODE, as shown in Figure 6.1. (see section 6.4 for more information about ObjectGEODE.) Consequently, we adopted the ObjectGEODE graphical format for both MSC and SDL representations. The MSC2SDL tool takes two files as an input. One contains the textual representation of the given MSC specification, and the other contains the textual representation of the target SDL arcitecture. Both files contain informal graphical information in CIF format[9]. In the current version, the two input files follow the ObjectGEODE format, and ObjectGEODE MSC and SDL editors are used for the creation of these files.

The MSC2SDL tool extracts the MSC and SDL specifications and builds the corresponding data structures. Then the algorithm is applied, and the tool produces the textual representation as well as the CIF graphical information for the generated SDL process behaviors. The generated SDL behaviors are inserted into the given SDL arctitecture, and saved into the input file or, as an alternative, into a new file. The output

file can be viewed and modified with ObjectGEODE SDL editor. Moreover, the generated SDL specification can be simulated using ObjectGEODE tools.

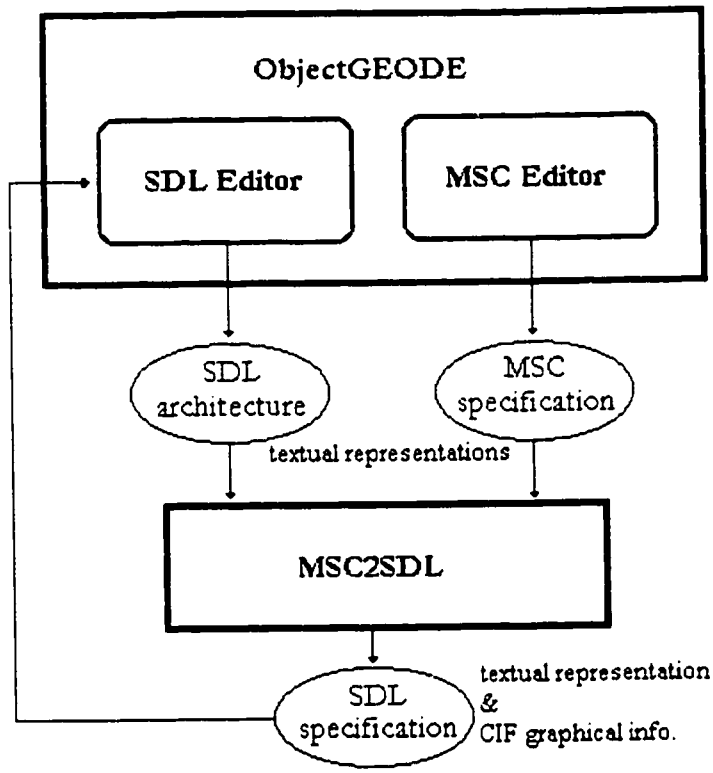


Figure 6.1. MSC2SDL and ObjectGEODE.

6.2 ARCHITECTURE OF MSC2SDL TOOL

The code of the MSC2SDL tool consists of several modules, as shown in Figure 6.2. Below, we give a brief description of each module in the tool. All modules are coded in C except for *Msc2sdl.tcl*, which is in Tcl/Tk.

- *Main* module is the main module. It starts the execution, and initializes all other modules. It also interprets user events received by *Msc2sdl.tcl* module and passes them to the appropriate module.

- *Msc2sdl.tcl* is the user interface module. It provides the main window as shown in Figure 6.3, as well as all other necessary dialog windows required by the MSC2SDL tool. The module passes all user requests to *Main*.

- *FileInterface* reads MSC and SDL specification files as well as writes the generated SDL process behaviors into the output file. In the current version, this module recognizes only ObjectGEODE file format.

- *Algorithm* applies the translation on the given data structure to generate SDL behaviors. It is also responsible for the actual translation of MSC constructs into its corresponding SDL constructs.

- *SDL* creates and manages the SDL data structure.

- *MSC* creates and manages the MSC data structure.

- *CheckComp* checks the static architecture consistency between the given MSC specification and the SDL architecture.

- *HMSC* is responsible for checking the semantics within the HMSC specification. In the current version, the module checks for the presence of non-local choice errors in the given MSC specification.

- *EventOrder* creates the Event Order Table for the given MSC specification.

- *MultiInstance* checks the existence of multi-instances and evaluates their behavior.

- *MatchCondition* checks for shared conditions among multi-instances or alternatives of the same instance.

- *Coregion* is responsible for detecting any general order among coregion events and creating coregion trees.

- *SaveMessages* checks for the messages that are required to be saved at a certain state.

- *RefineSDL* enhances the produced SDL process behaviors. After the behavior has been generated, the module checks for the existence of different states that have the same behavior. These states merge into one state and update all the related links.

- *Graphics* is responsible for distributing the generated SDL behavior on the graphical sheet. The produced graph may be spread over several pages.

- *Tree* handles tree data structure that is required for coregion.

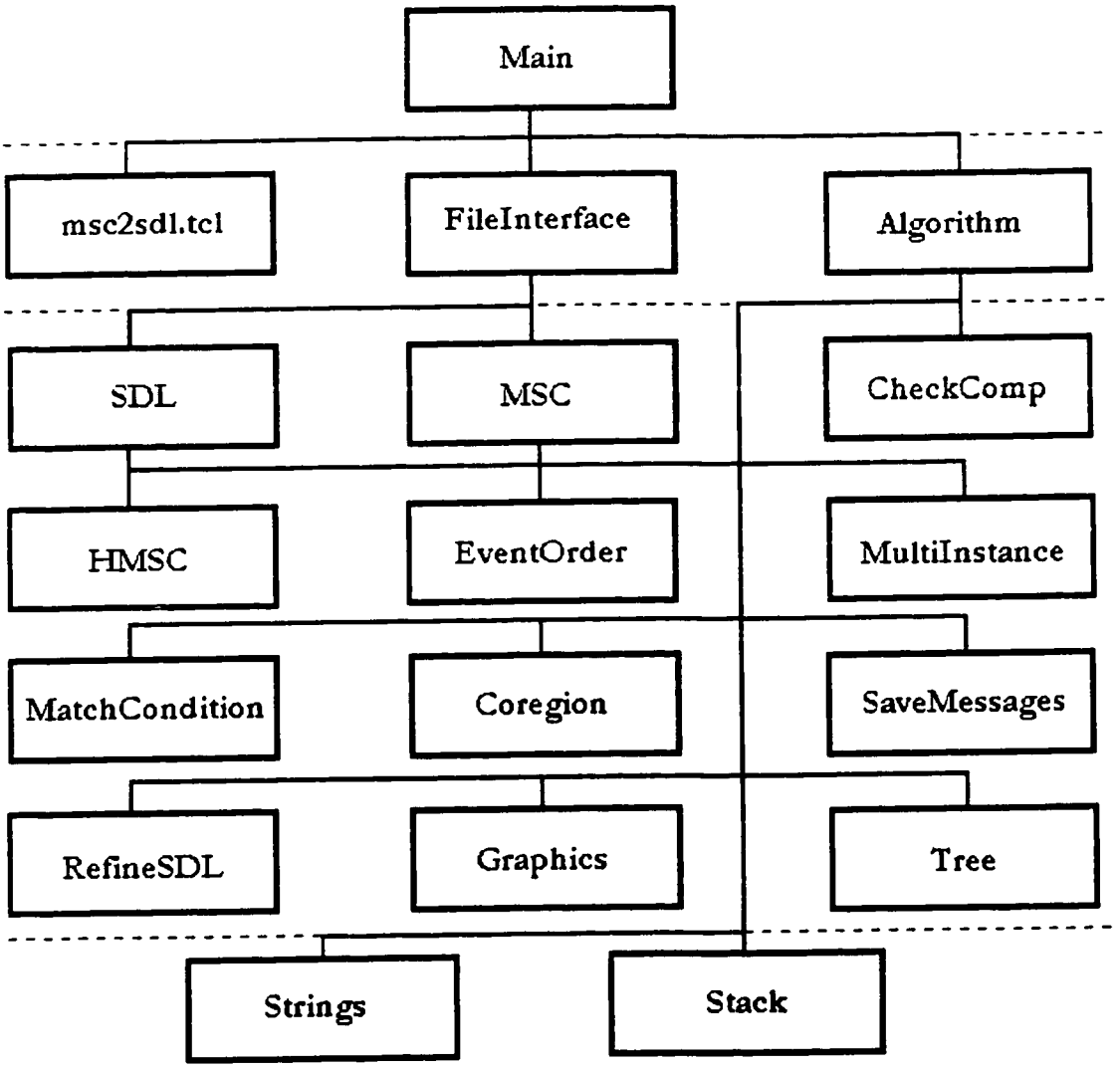


Figure 6.2. MSC2SDL architecture.

- *Strings* handle special string functions used by the algorithm. It is used by almost all of the other modules.

- *Stack* handles stack operations. The stack is used by *HMSC* module to save the status of the current generated FSM when an alternative operator is translating. As a result, all other alternative branches will be joined at the same position.

The execution of the tool starts with the *Main* module. The *Main* calls the *Msc2sdl.tcl* module, which controls the interactions with the user.

In response to user commands, *Msc2sdl.tcl* will trigger *Main* with the corresponding event. When the user gives an input file (MSC or SDL specification), *Main* will call

FileInterface to retrieve the file and store the specification into a data structure through *MSC* or *SDL* module. *Main* calls *Algorithm* to apply the translation. Then, *Algorithm* will call the other modules in the following order: *CheckComp, EventOrder, Coregion, SaveMessages, MultiInstance, HMSC, RefineSDL then Graphics.*

Finally, *Main* calls *FileInterface* to store the produced process behaviors in the output file.

6.3 THE USER INTERFACE

The MSC2SDL tool has a simple user interface. The main window, as shown in Figure 6.3, is divided into three parts: *main menu*, *user info* and *status bar.*

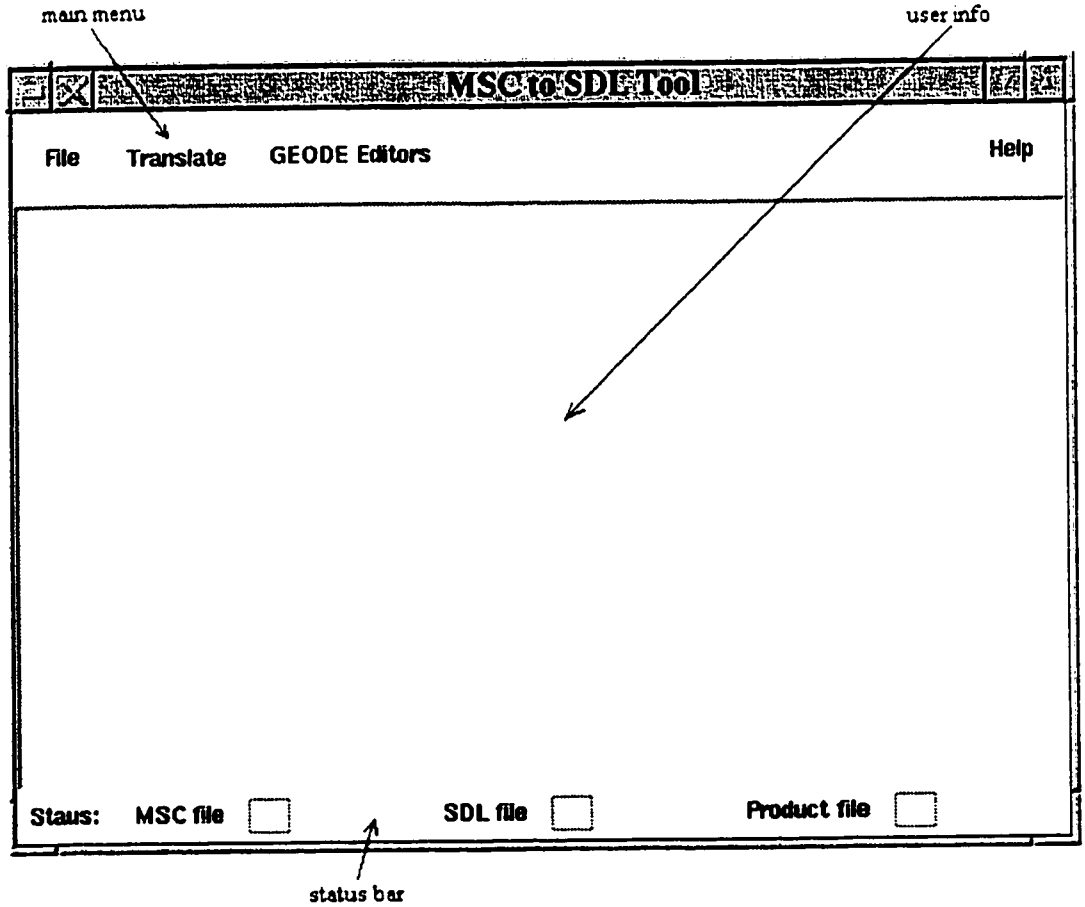


Figure 6.3. MSC2SDL main window.

The *main menu* provides the user with all necessary commands required to apply our approach. Figure 6.4 shows the file popup menu which will be activated when the user selects the *file* button from the *main menu*.

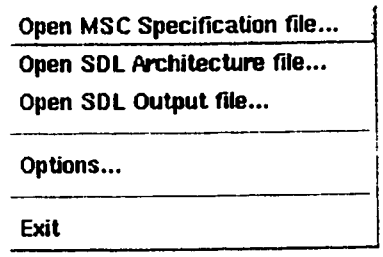


Figure 6.4. File popup menu for MSC2SDL tool.

The user feeds the tool with the input files and the name of the output file. (In case the output file name is not provided, the tool will write the generated specification into the given SDL architecture file.) As soon as the user chooses the file type, the file window (Figure 6.5) will pop up to browse the system for the appropriate file.

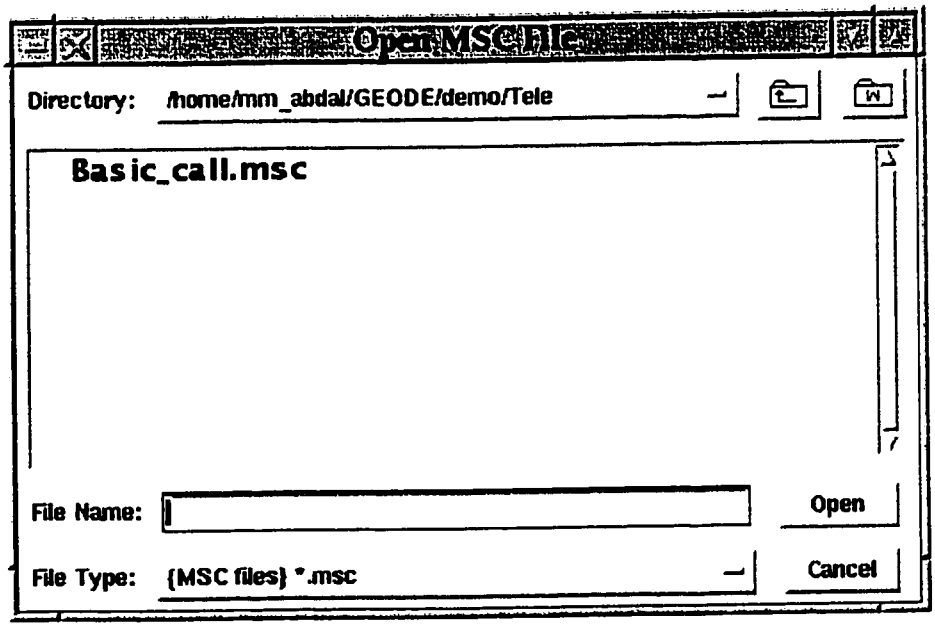


Figure 6.5. File window for MSC2SDL tool.

As shown in Figure 6.3, the *translate* button in the *main menu* performs the translation of the MSC specification that is provided in the MSC input file. *GEODE Editors* button, in

the *main menu*, assists the user to launch ObjectGEODE MSC and SDL editors. The *help* button provides the user with useful information about the tool.

The *user info* area is designed to prompt the user with any potential errors that may be detected during the translation.

The *status bar* shows the names of the input files as well as the output file.

6.4 OBJECTGEODE

ObjectGEODE is a toolset based on complementary formal languages, UML, MSC and SDL. ObjectGEODE provides graphical editors for these languages. It also supports the SDL language with simulation and verification tools as well as a C/C++ code generator. ObjectGEODE is intended to analyze, design, validate through simulation and verification, for code generation and testing of real-time and distributed systems.

While ObjectGEODE supports the full SDL Language, it supports only a subset of UML and MSC Languages. Within ObjectGEODE, MSC is mainly used to derive test cases and observers for the validation of the SDL specification. ObjectGEODE does not support HMSCs or inline expressions, though it provides logical operators (as shown in Figure 6.6) which may be used for HMSCs and inline expressions.

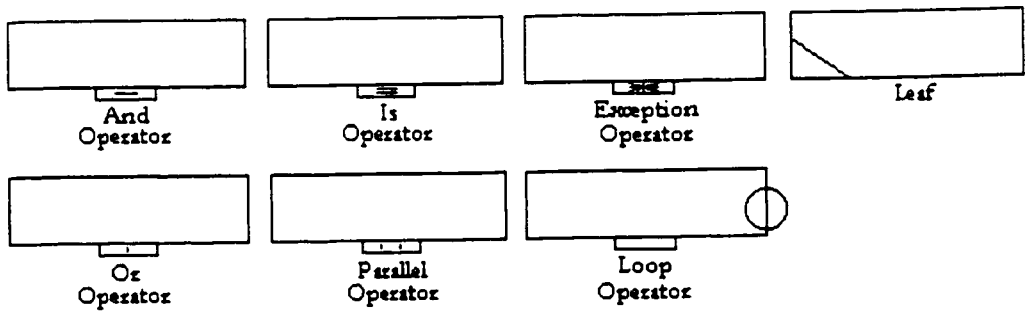


Figure 6.6. ObjectGEODE logical operators.

The *And* and *Is* operators are equivalent to the sequential operator in HMSC. The *Or* operator is equivalent to the alternative operator in HMSC and the alt inline expression in bMSC. The *Parallel* operator is equivalent to the parallel operator in HMSC and the par inline expression in bMSC. The *Loop* operator is equivalent to the iterative operator in HMSC and the loop inline expression in bMSC. The *Exception* operator is equivalent to the exception inline expression in bMSC. The *Leaf* is equivalent to a bMSC.

Unfortunately, ObjectGEODE allows users to add timer events within coregions in the graphical representation of MSC, even if they will be thrown out of the coregion in the textual representation.

6.5 APPLICATIONS

In this section, we will illustrate the application of MSC2SDL tool to real case studies.

6.5.1 BASIC TELEPHONE CALL

This subsection demonstrates the application of the MSC2SDL tool to a basic call in the telephone system. The MSC specification and SDL architecture have been created using ObjectGEODE editors.

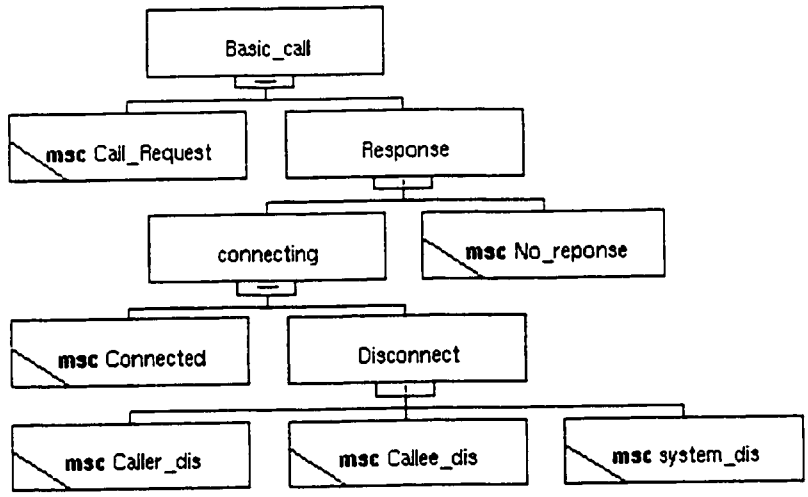
SCENARIOS

The basic call scenarios have three actors: the caller's local controller, the callee's remote controller and the telephone system. In this scenario, the users and their telephone machines are represented by the system environment. The caller will dial the callee phone number (Call_Request bMSC.) After a short time, in the case of no response from the callee, the caller will receive a busy signal (No_response bMSC.) If the callee responds to the ringing signal, the two will be connected (Connected bMSC.) To terminate the call,

one of the parties has to hang up the telephone (Caller_dis and Callee_dis bMSCs.) On the other hand, the telephone system may malfunction and disconnect the parties (system_dis bMSC.)

MSC SPECIFICATION

Figure 6.7 shows the MSC specification of the basic call. This example demonstrates multi-instances translation, as well as sequential and alternative operators.



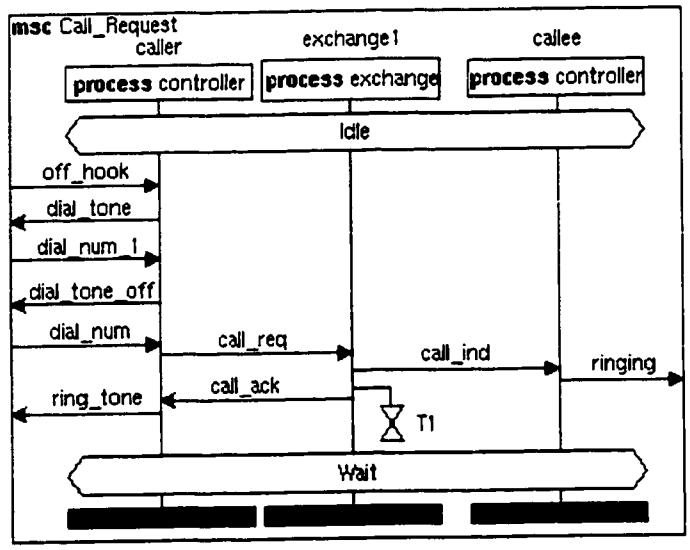


Figure 6.7. Basic call MSC specification.

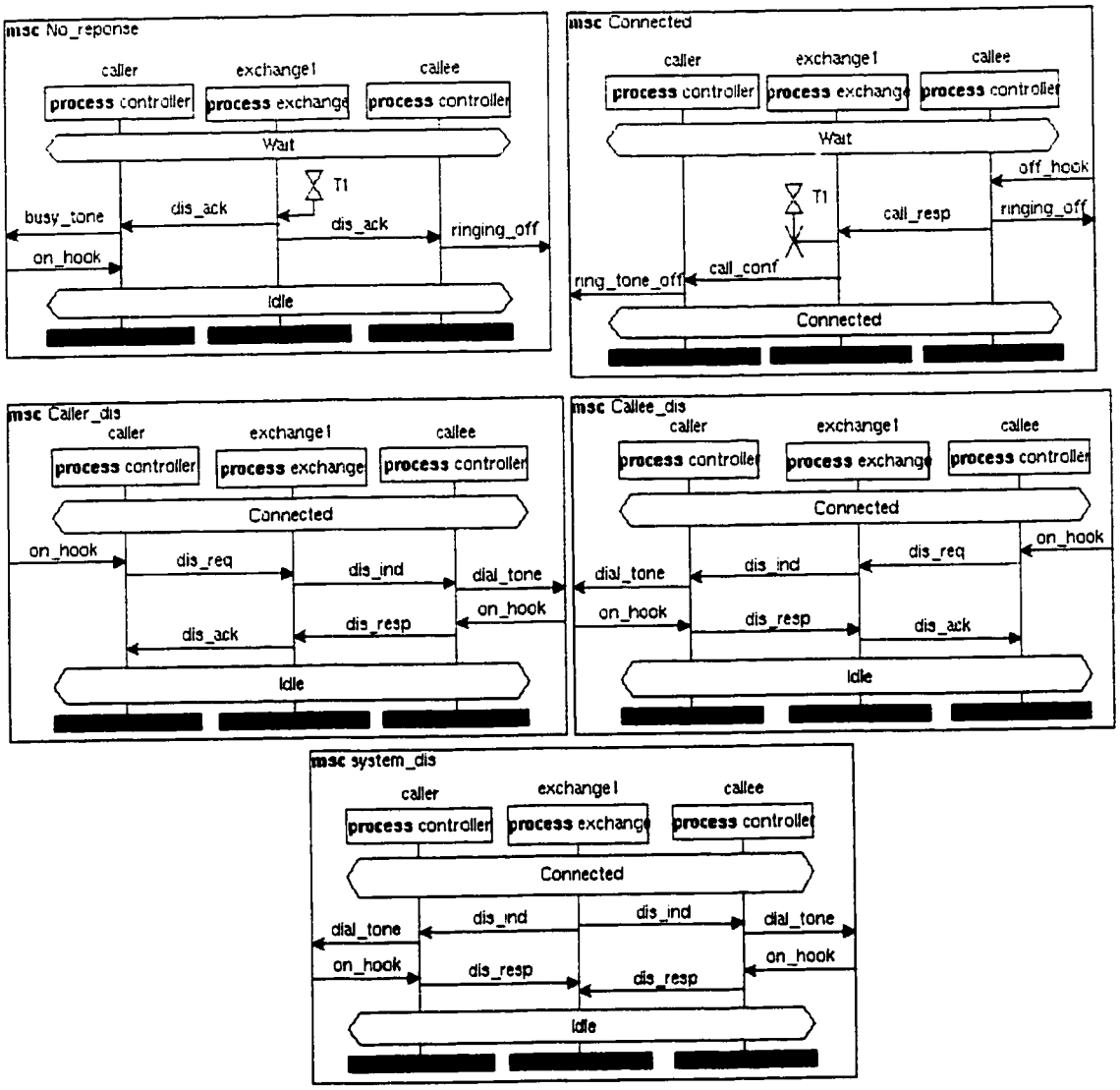


Figure 6.7. Basic call MSC specification. (cont...)

SDL ARCHITECTURE

The SDL architecture of this system, as shown in Figure 6.8, consists of two blocks - *controller* and *exchange*. The two blocks exchange signals through channel *Channel2*. The *controller* block communicates with the environment through channel *Channel1*. The *exchange* block (Figure 6.9.a) contains one process *exchange*, which communicates with the block environment through *Signalroute1*. Signal route *Signalroute1* is connected to channel *Channel2* at the block border.

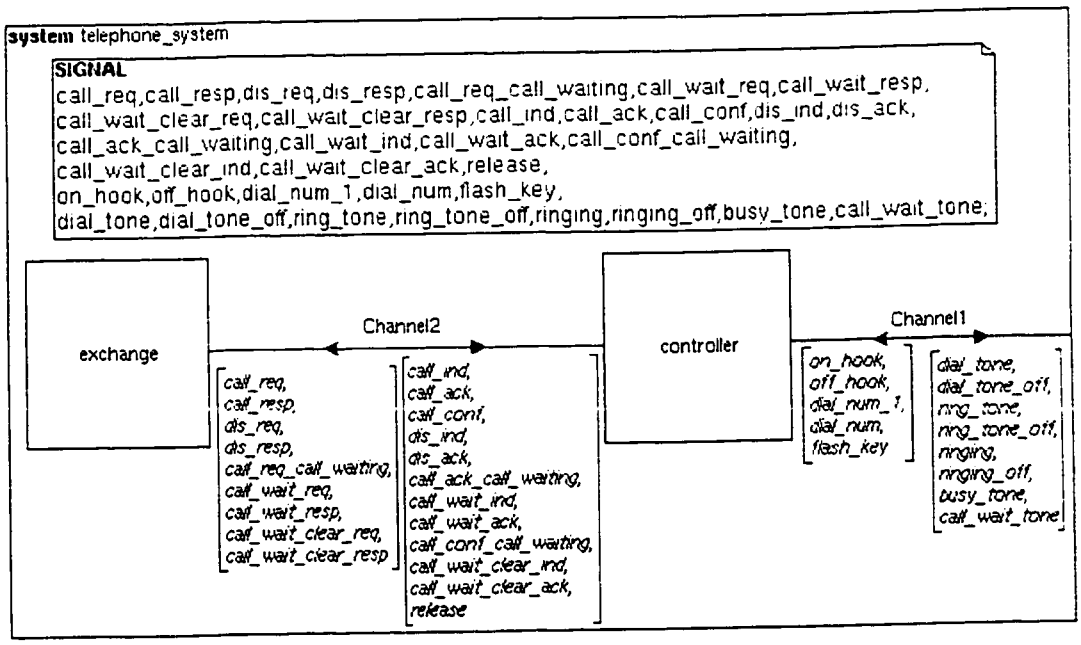
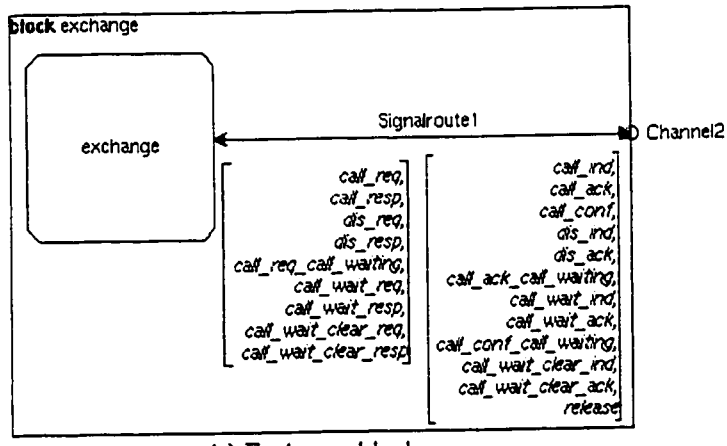
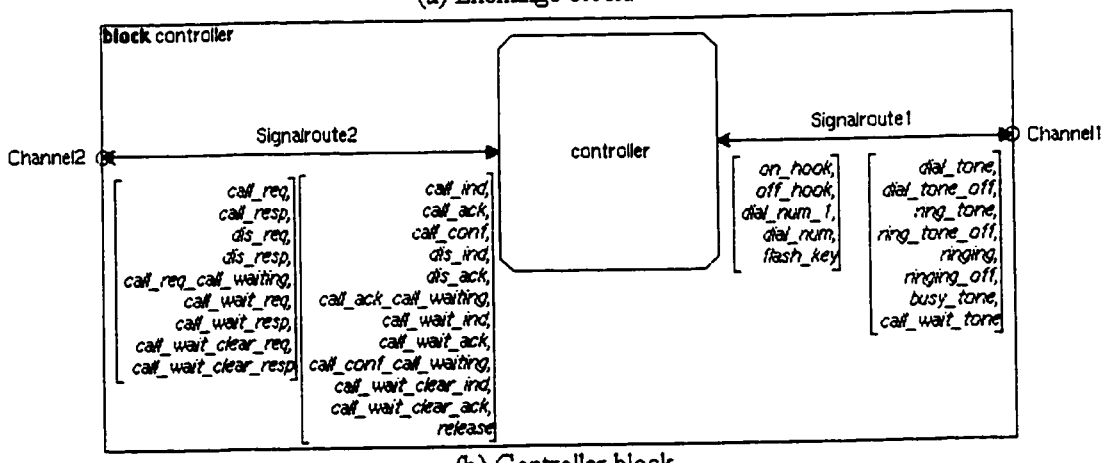


Figure 6.8. Basic call SDL architecture (system).



(a) Exchange block.



(b) Controller block.

Figure 6.9. Basic call SDL architecture (blocks).

The *controller* block, (Figure 6.9.b) contains one process *controller*, which communicates with the block environment through *Signalroute1* and *Signalroute2*. At *controller* border, signal route *Signalroute1* is connected to channel *Channel1*, and signal route *Signalroute2* is connected to channel *Channel2*.

SEMANTIC ERRORS

MSC2SDL tool has detected two non-local choice problems in the given MSC specification.

The first non-local choice is caused by alternative *Response*, which is controlled by two different instances *env* and *exchange1*. As shown in Figure 6.10, bMSC *Connected* is controlled by *env*, while bMSC *No_response* is controlled by *exchange1*.

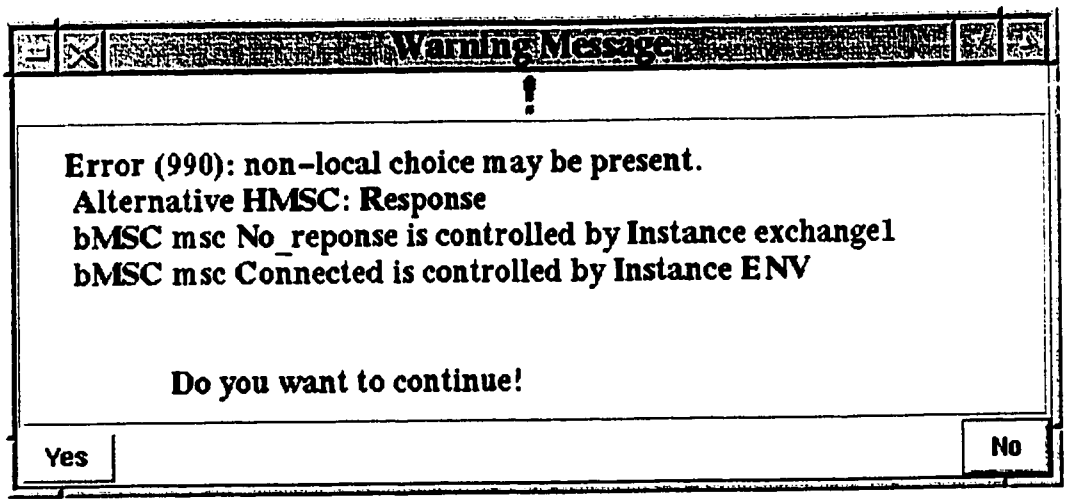


Figure 6.10.Non-local choice alert1 for basic call specification.

The second non-local choice is caused by alternative *Disconnect*, which is controlled by two different instances *env* and *exchange1*, as shown in Figure 6.11. In Figure 6.7, bMSC *Caller_dis* and bMSC *Callee_dis* are controlled by *env*, while bMSC *system_dis* is controlled by *exchange1*.

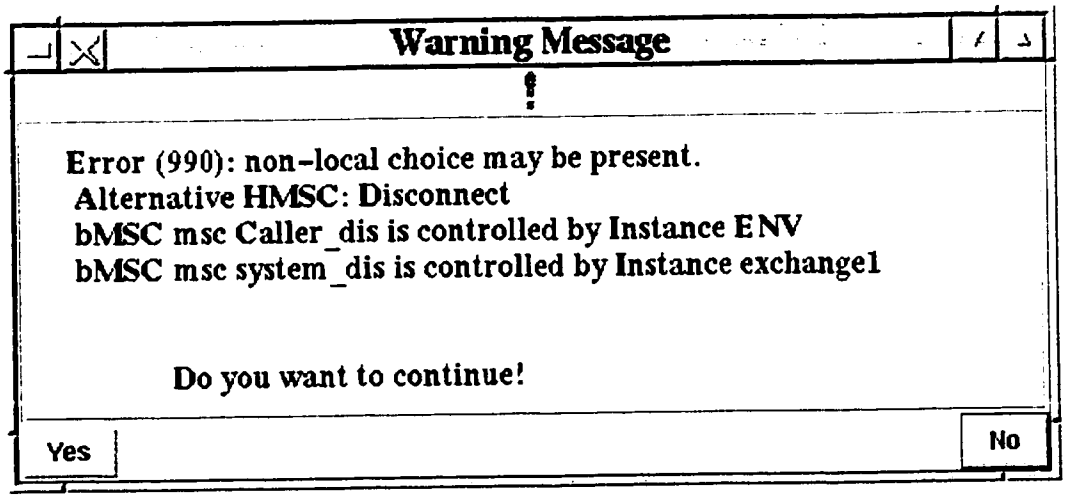


Figure 6.11.Non-local choice alert2 for basic call specification.

GENERATED SDL BEHAVIOR

The generated SDL behavior of process *exchange* is shown in Figure 6.12. The process behavior saves the *caller* pid (state *Idle*) and the *callee* pid (state *Wait*) to use them later in order to identify the sender instance of the received signal *dis_req* in state *Connected*. However, state *Connected* is not stable because of the spontaneous transition *NONE*. The process *exchange* may execute the spontaneous transition, even if there is a *dis_req* signal in the input queue. During the generation, the MSC2SDL tool has alerted the user of such problem as shown in Figure 6.11.

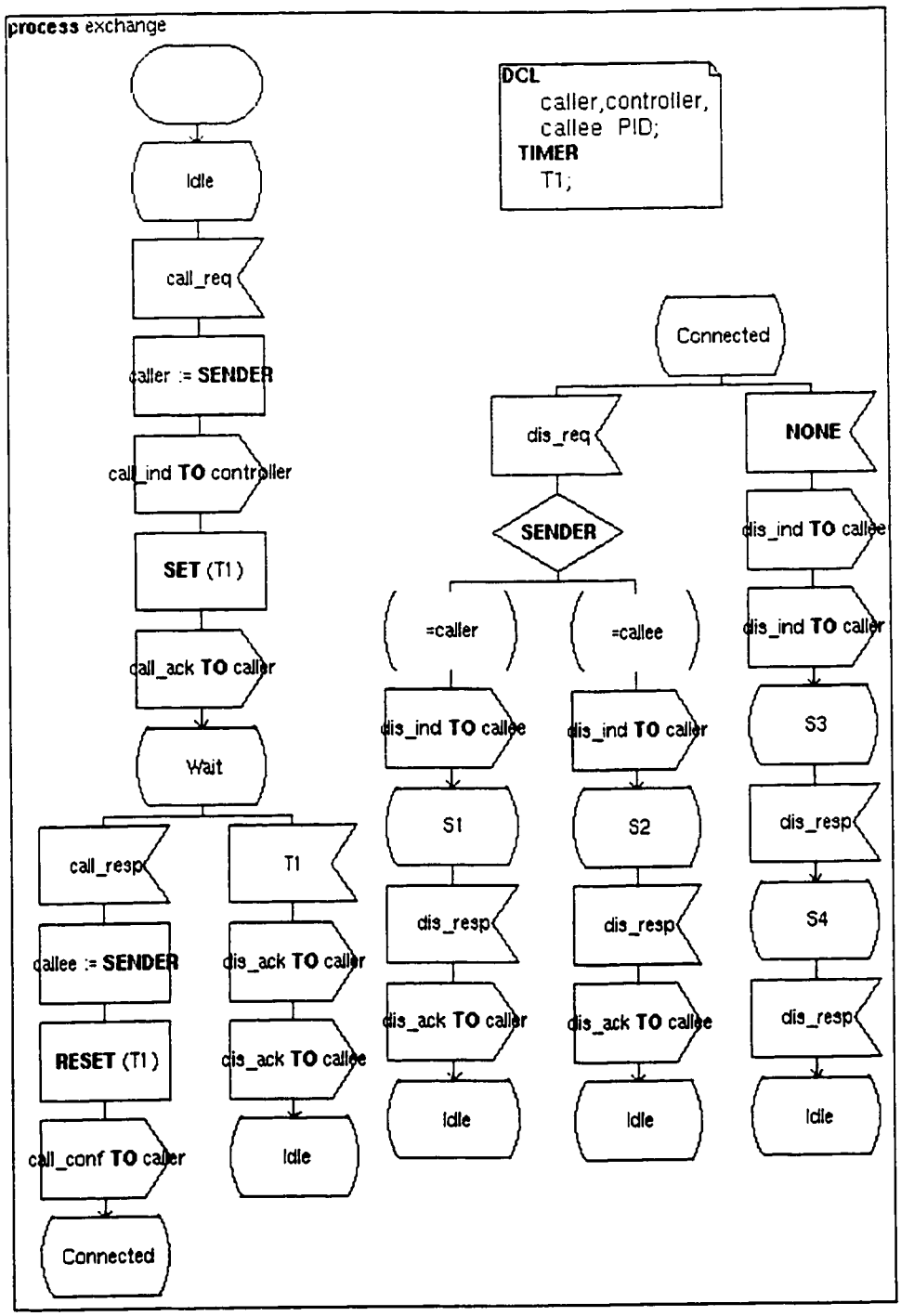


Figure 6.12. Generated process behavior (*Exchange*) for MSC specification in Figure 6.7.

The generated SDL behavior of process *controller* is shown in Figure 6.13. The process behavior is split into two pages. The behaviors of the *caller* instance and the *callee*

instance are merged to produce the process behavior. In the state *Idle*, each instance will save its pid in order to distinguish itself later in state *Wait*.

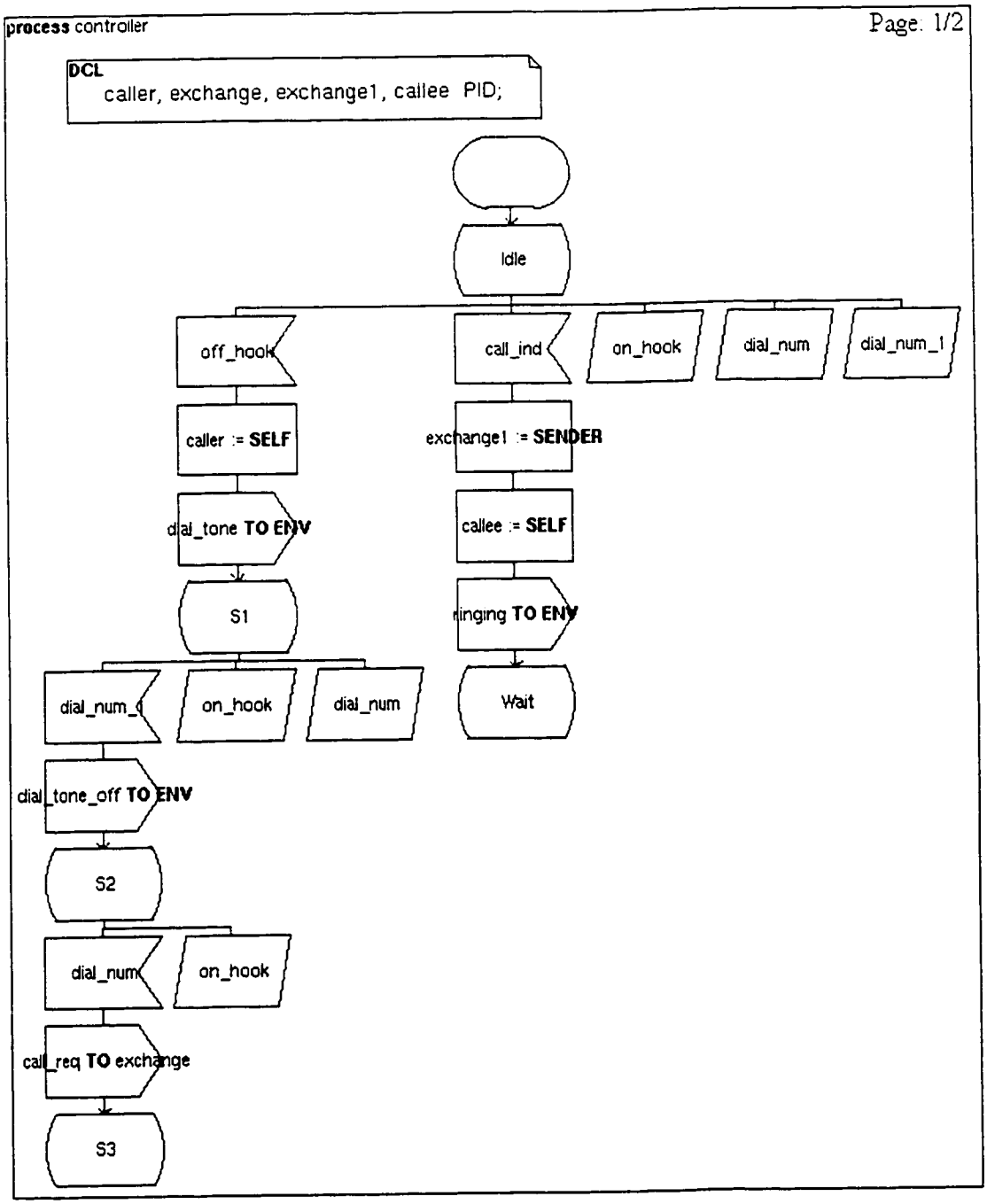


Figure 6.13.Generated process behavior (*controller*) for MSC specification in Figure 6.7.

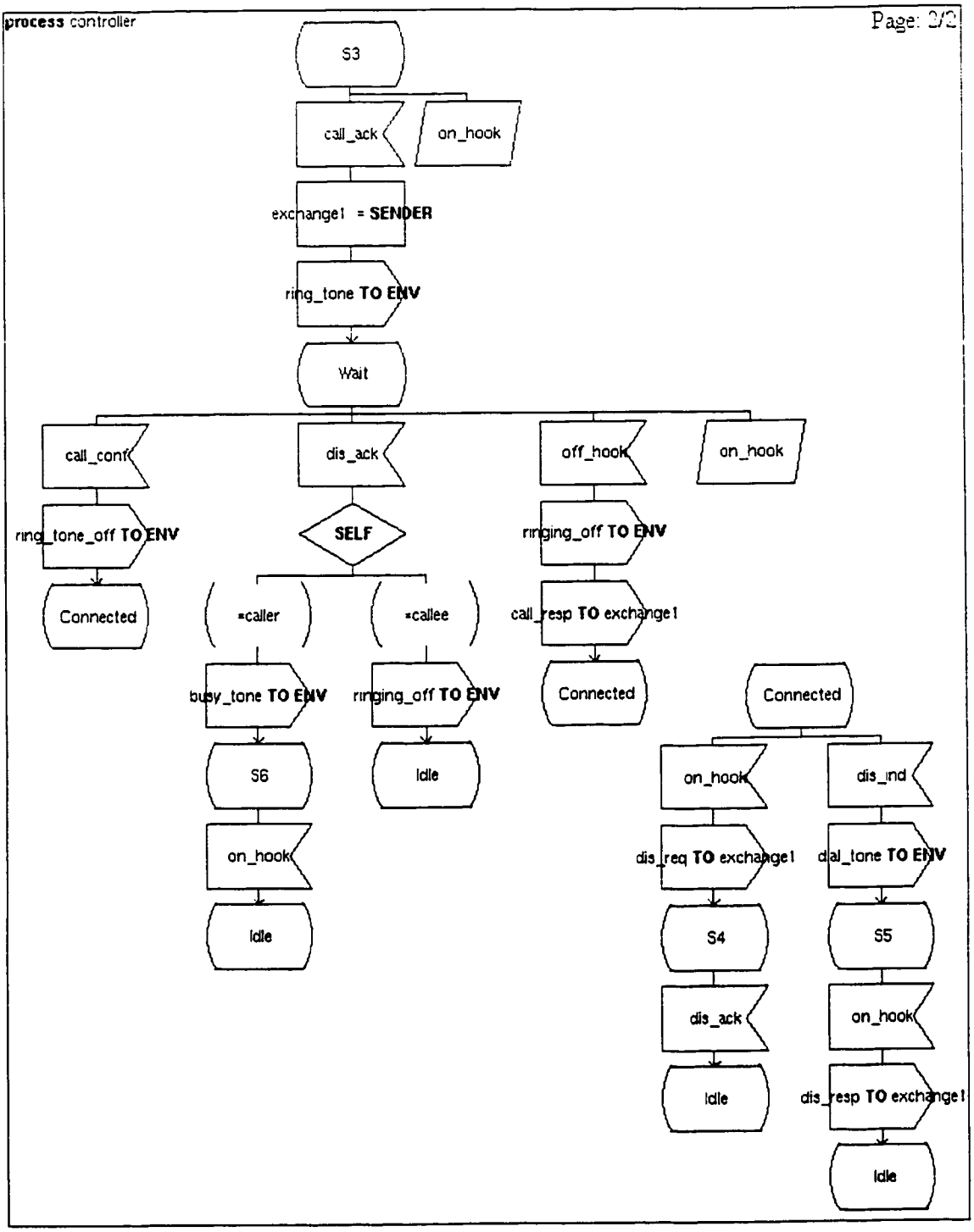


Figure 6.13.Generated process behavior (*controller*) for MSC specification in Figure 6.7. (cont...)

6.5.2 AUTOMATIC TELLER MACHINE (ATM)

SCENARIOS

The ATM scenario consists of three actors: the user, the ATM machine and the bank system. For simplicity, we have specified only two ATM functions (withdraw and get balance). In this specification, the user will supply his card and password to the ATM machine. Then, he will choose one of three options:

- Get balance of his account.
- Withdraw allowable amount from his account.
- End the transaction.

However, the user card may be rejected for the following reasons:

- Password has not been supplied.
- Incorrect password.

MSC SPECIFICATION

The MSC specification of the ATM is shown in Figure 6.14. The specification demonstrates Sequential, Alternative and Iterative HMSC operators.

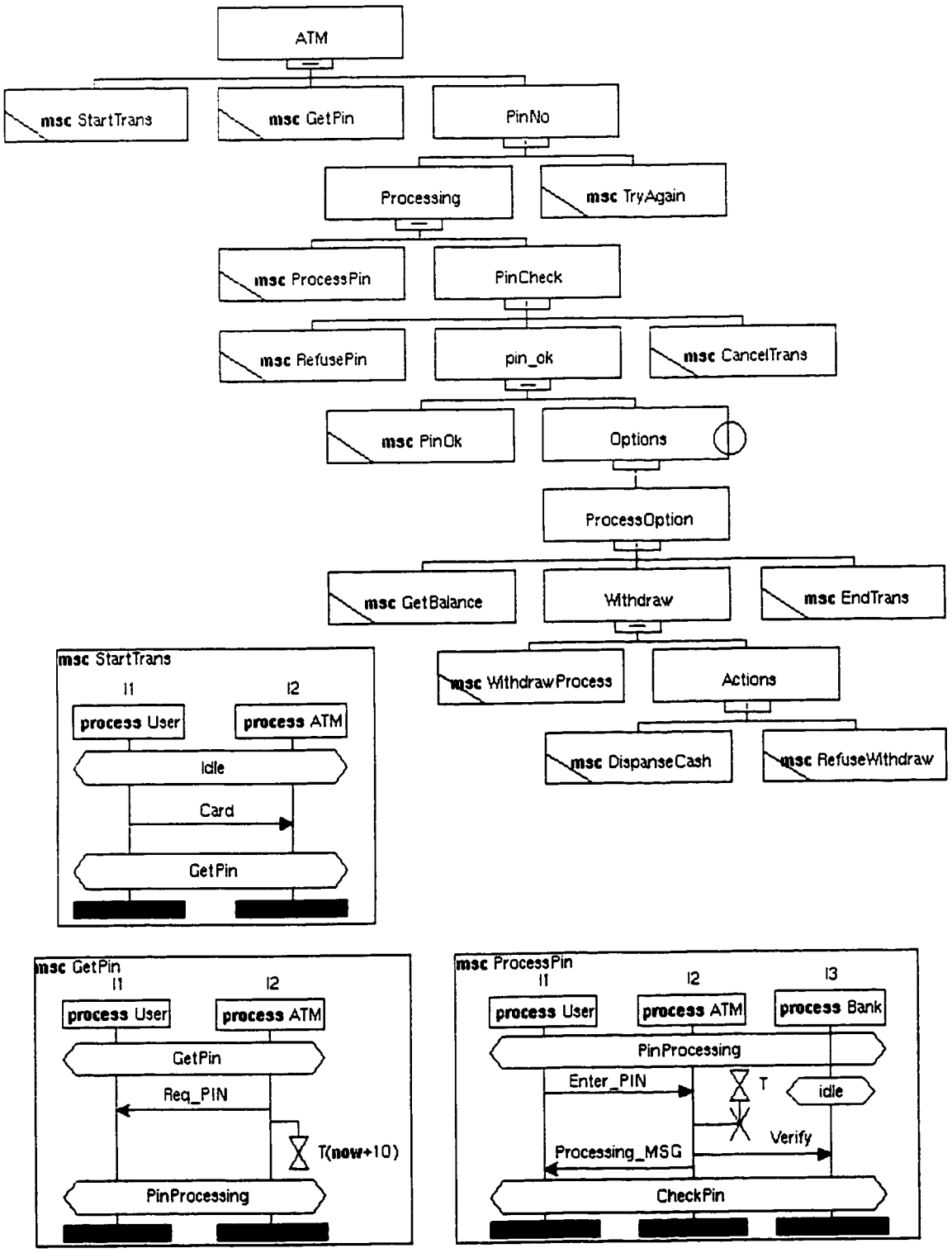


Figure 6.14. ATM MSC specification.

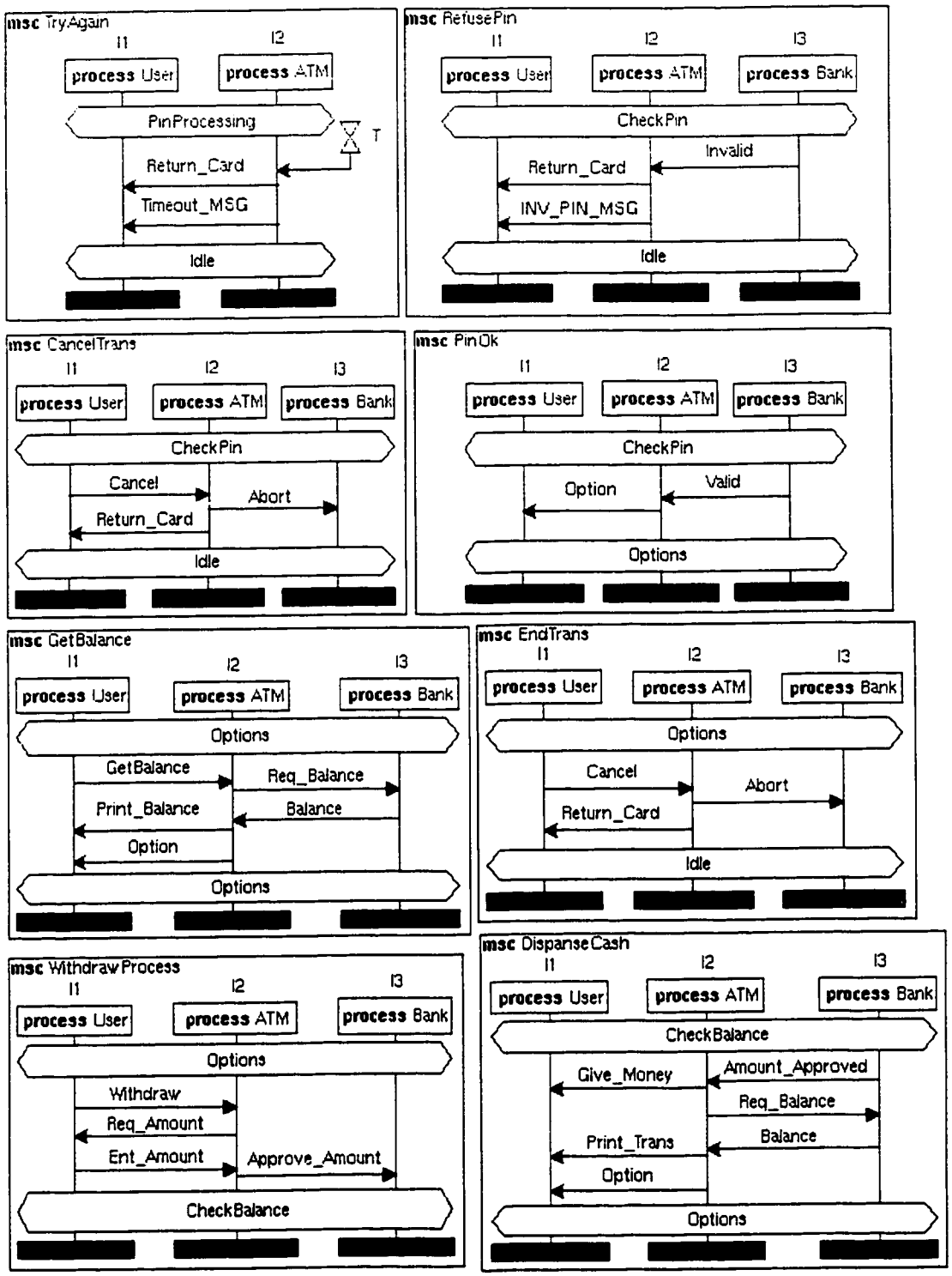


Figure 6.14. ATM MSC specification. (cont...)

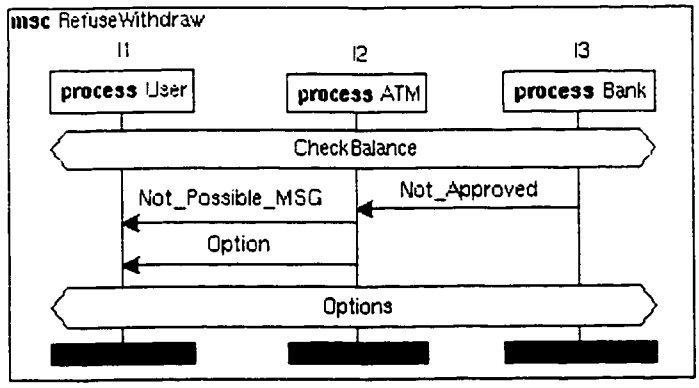


Figure 6.14. ATM MSC specification. (cont...)

SDL ARCHITECTURE

Figure 6.15 shows the SDL architecture of the system, which consists of one block *ATM_Block*. The *ATM_Block* consists of three processes *User*, *ATM* and *Bank*. Processes *User* and *ATM* communicate through *Signalroute1*. Processes *ATM* and *Bank* communicate through *Signalroute2*.

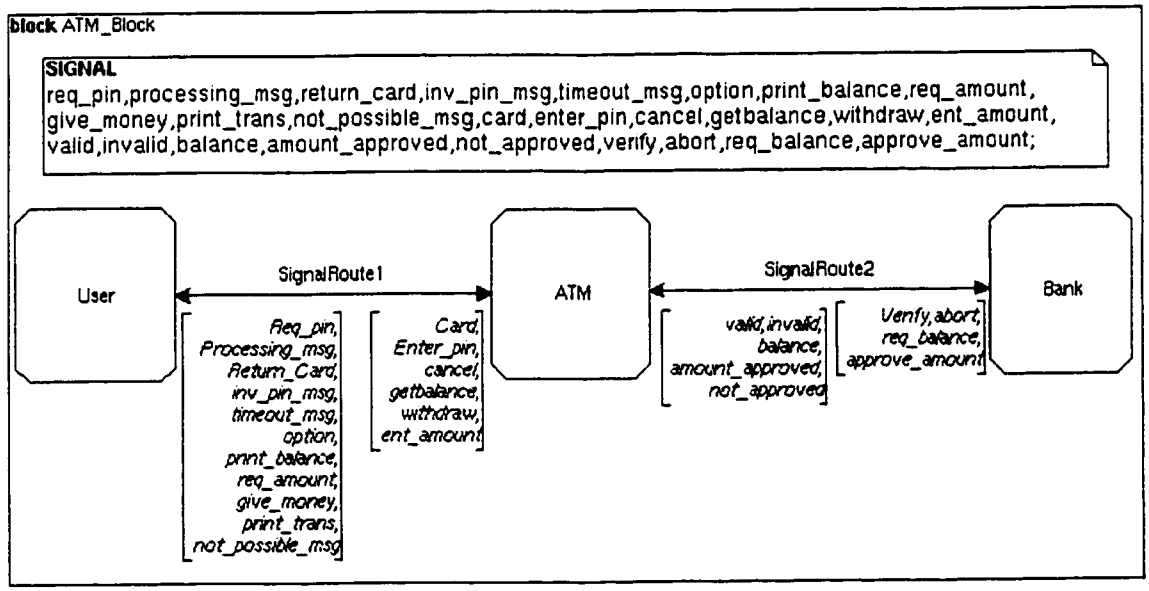


Figure 6.15. ATM SDL architecture.

SEMANTIC ERRORS

MSC2SDL tool has detected two non-local choice problems in the given MSC specification.

The first non-local choice is caused by alternative *PinNo*, which is controlled by two different instances *I1* and *I2*. As shown in Figure 6.16, bMSC *ProcessPin* is controlled by *I1*, while bMSC *TryAgain* is controlled by *I2*.

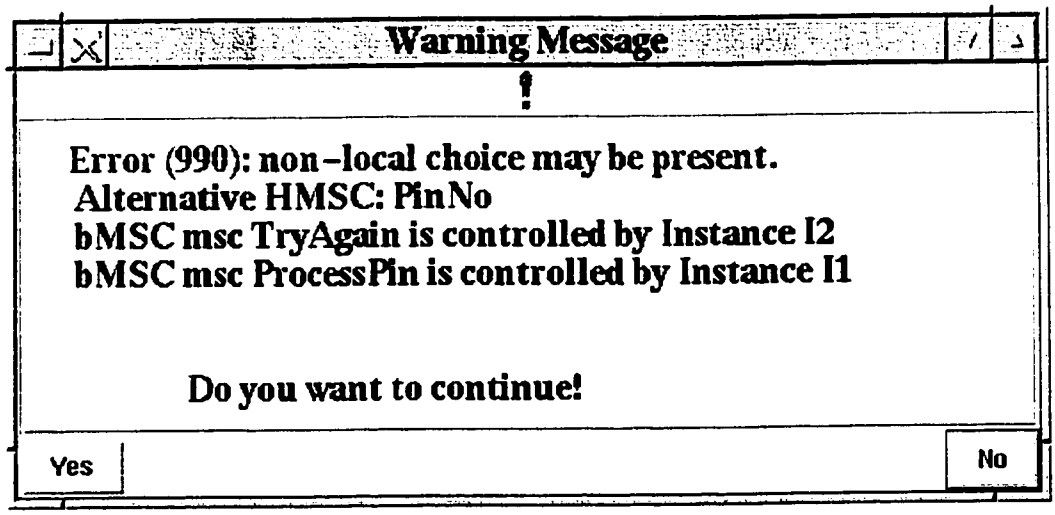


Figure 6.16.Non-local choice alert1 for ATM specification.

The second non-local choice is caused by alternative *PinCheck*, which is controlled by two different instances *I1* and *I3*, as shown in Figure 6.17. In Figure 6.14, bMSC *RefusePin* and bMSC *PinOk* are controlled by *I3*, while bMSC *CancelTrans* is controlled by *I1*.

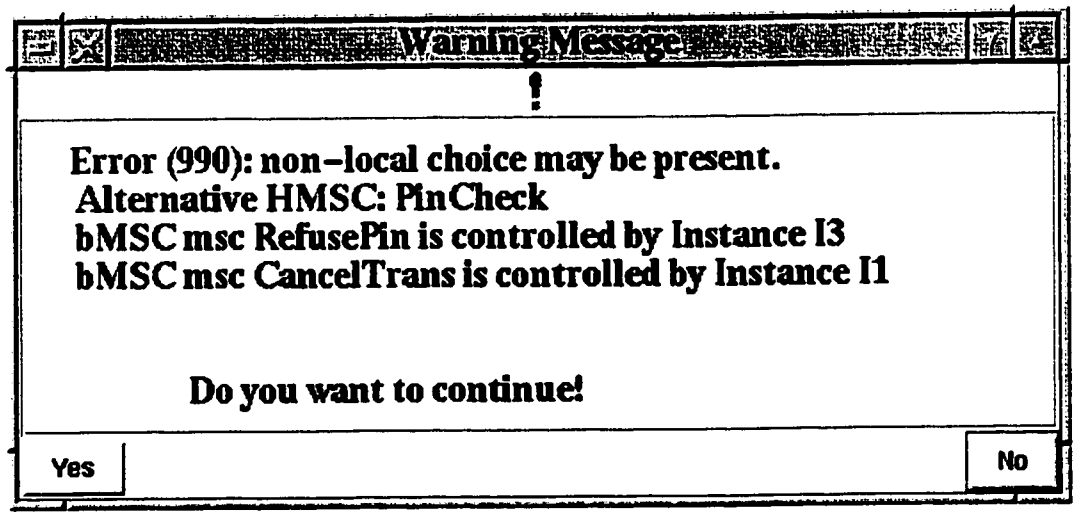


Figure 6.17.Non-local choice alert2 for ATM specification.

GENERATED SDL PROCESSES

The generated SDL behavior of process *User* is shown in Figure 6.18. The process behavior is split into two pages. State *PinProcessing* and state *CheckPin* are unstable because of the spontaneous transition *NONE*.

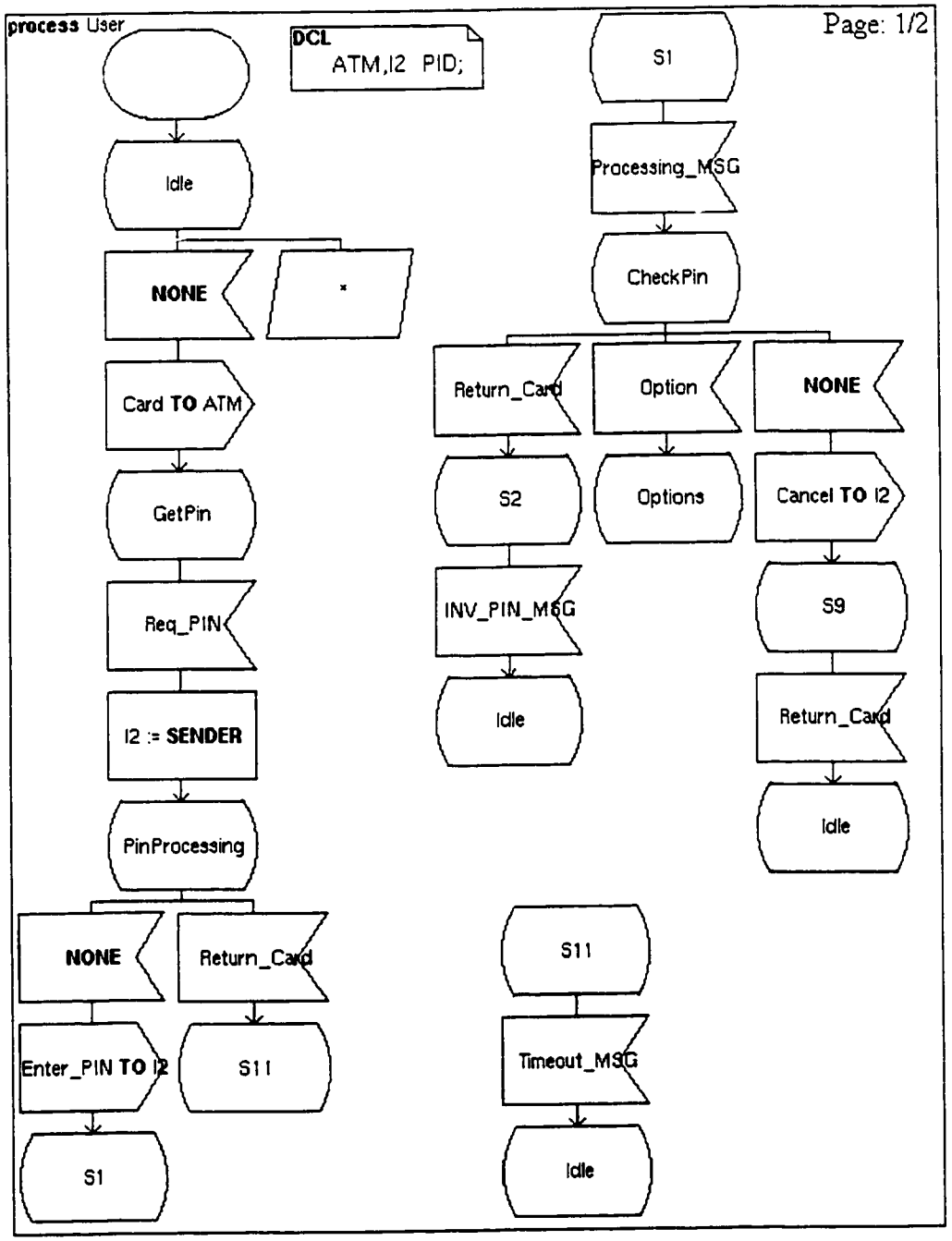


Figure 6.18.Generated process behavior (*User*) for MSC specification in Figure 6.14.

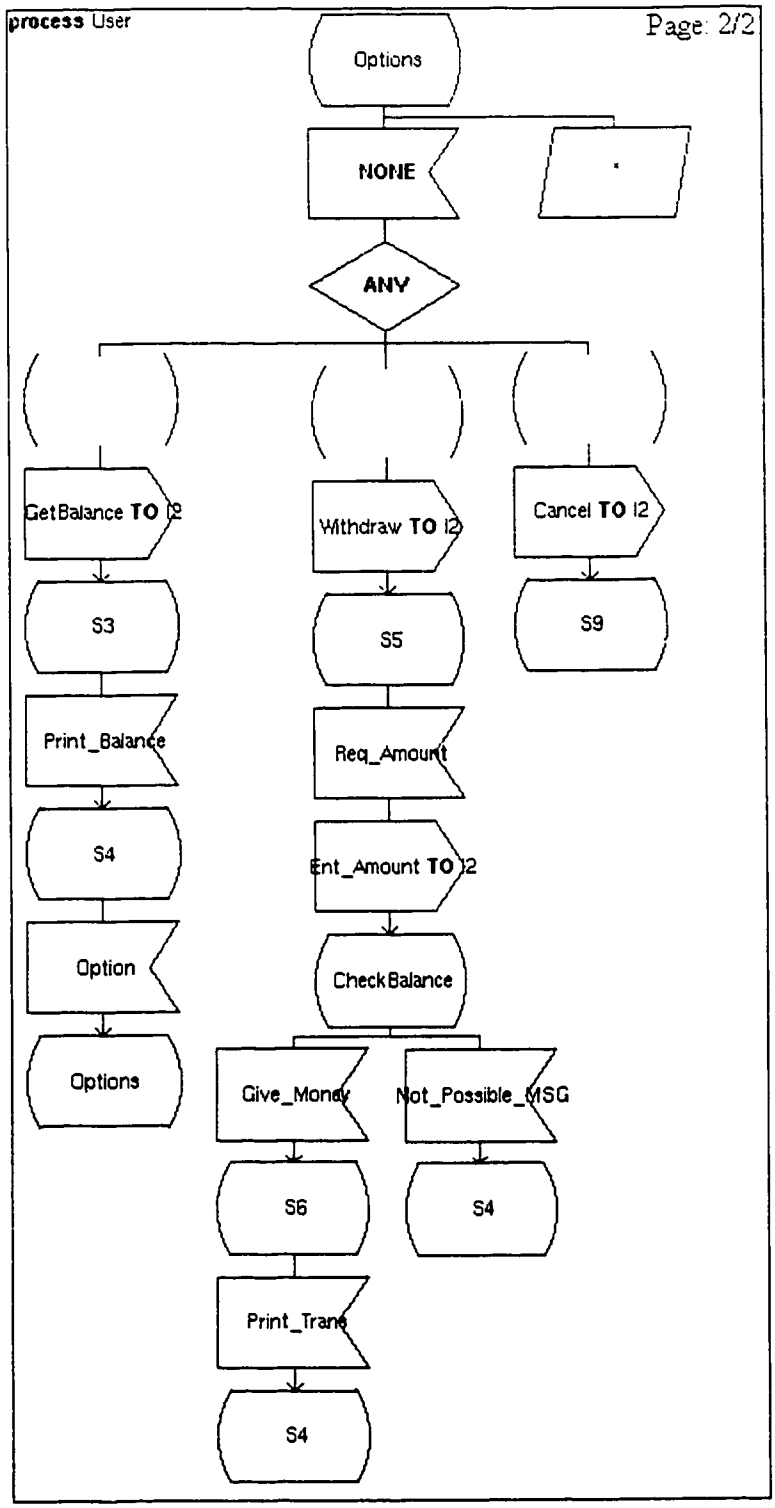


Figure 6.18.Generated process behavior (*User*) for MSC specification in Figure 6.14. (cont...)

The generated SDL behavior of process *ATM* is shown in Figure 6.19. The process behavior is split into two pages.

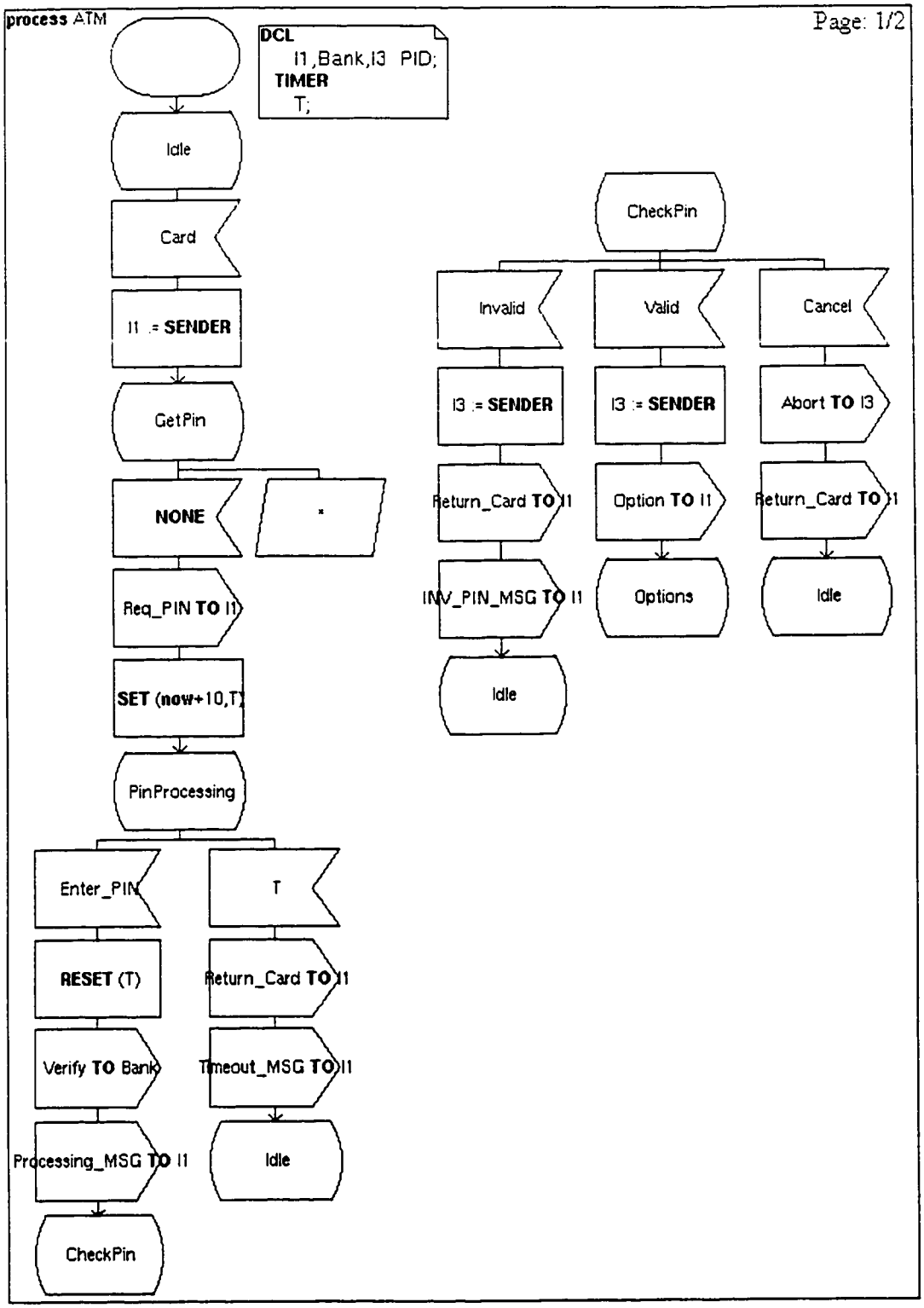


Figure 6.19.Generated process behavior (*ATM*) for MSC specification in Figure 6.14.

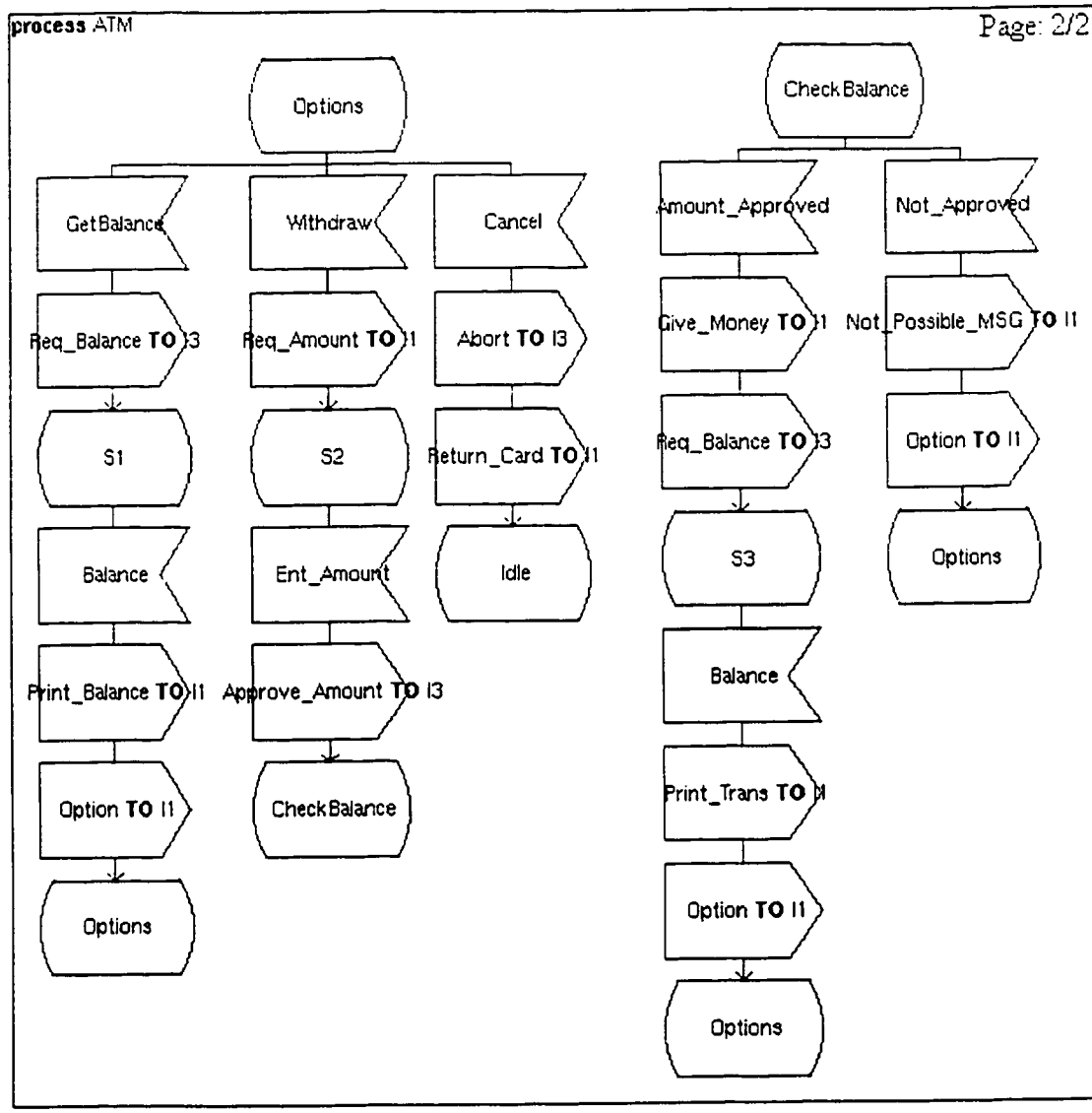


Figure 6.19.Generated process behavior (*ATM*) for MSC specification in Figure 6.14. (cont...)

The generated SDL behavior of process *Bank* is shown in Figure 6.20. State CheckPin is unstable because of the spontaneous transition NONE. In this state, the process may choose, non-deterministically, to wait for signal *Abort* and consumes it, or to activate the spontaneous transition and sends signal *Invalid* or signal *Valid.*

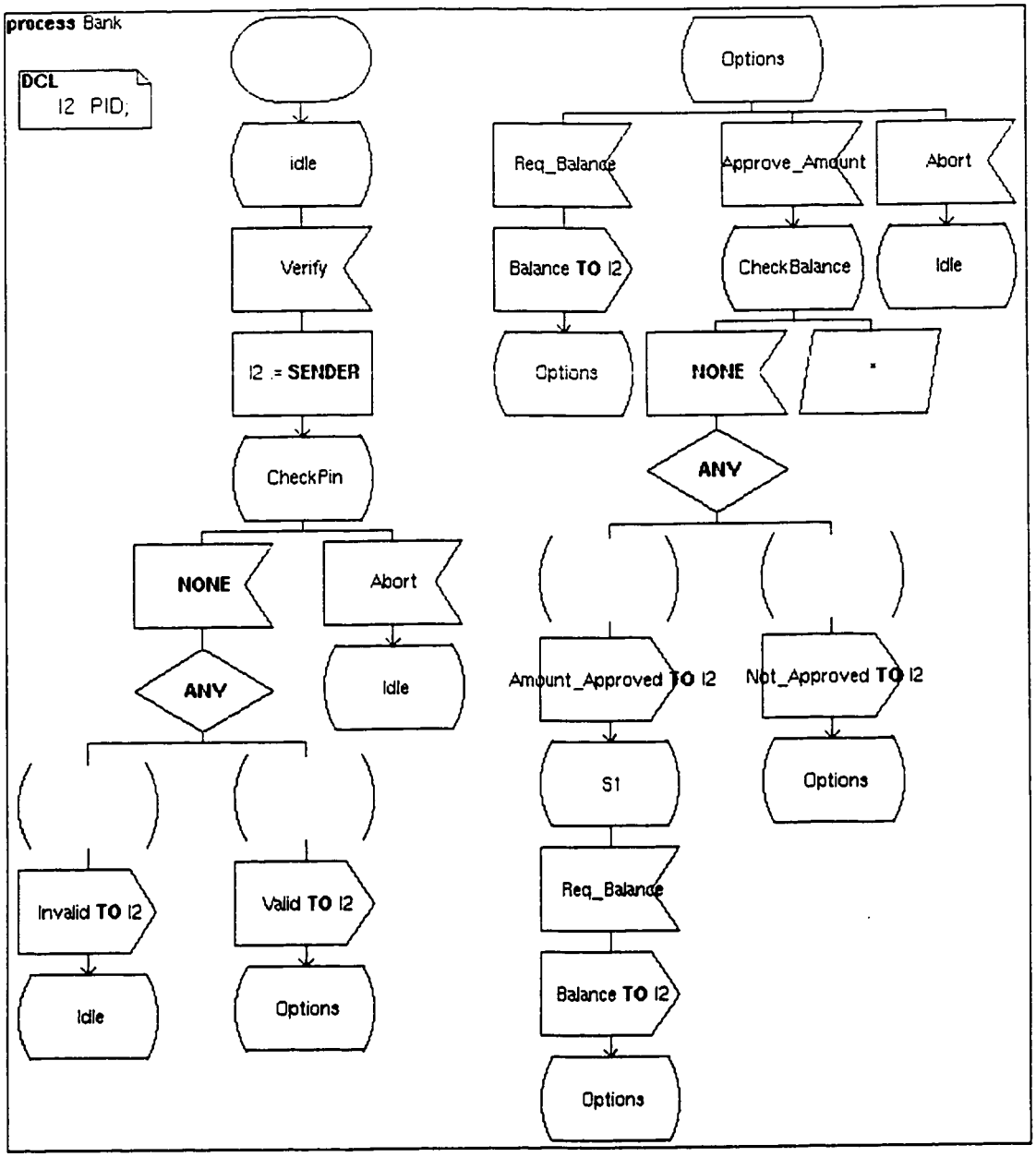


Figure 6.20.Generated process behavior (*Bank*) for MSC specification in Figure 6.14.

6.5.3 INRES PROTOCOL

SCENARIOS

The INRES system contains five actors: *Initiator*, *Coder_ini*, medium, *Coder_res* and *Responder*. The specification shows the three phases of the protocol: connection phase,

data transfer phase and disconnection phase. In our specification, we assume a reliable medium.

MSC SPECIFICATION

Figure 6.21 shows the MSC specification of the INRES protocol. This specification demonstrates the multi-instance translation, where instance *Initiator* and instance *Responder* are of the same process type *INRES*. Also, instance *Coder_ini* and instance *Coder_res* are of the same process type *Coder*. Instances *Initiator* and *Coder_ini* represent the initiation side, while instances *Responder* and *Coder_res* represent the responding side.

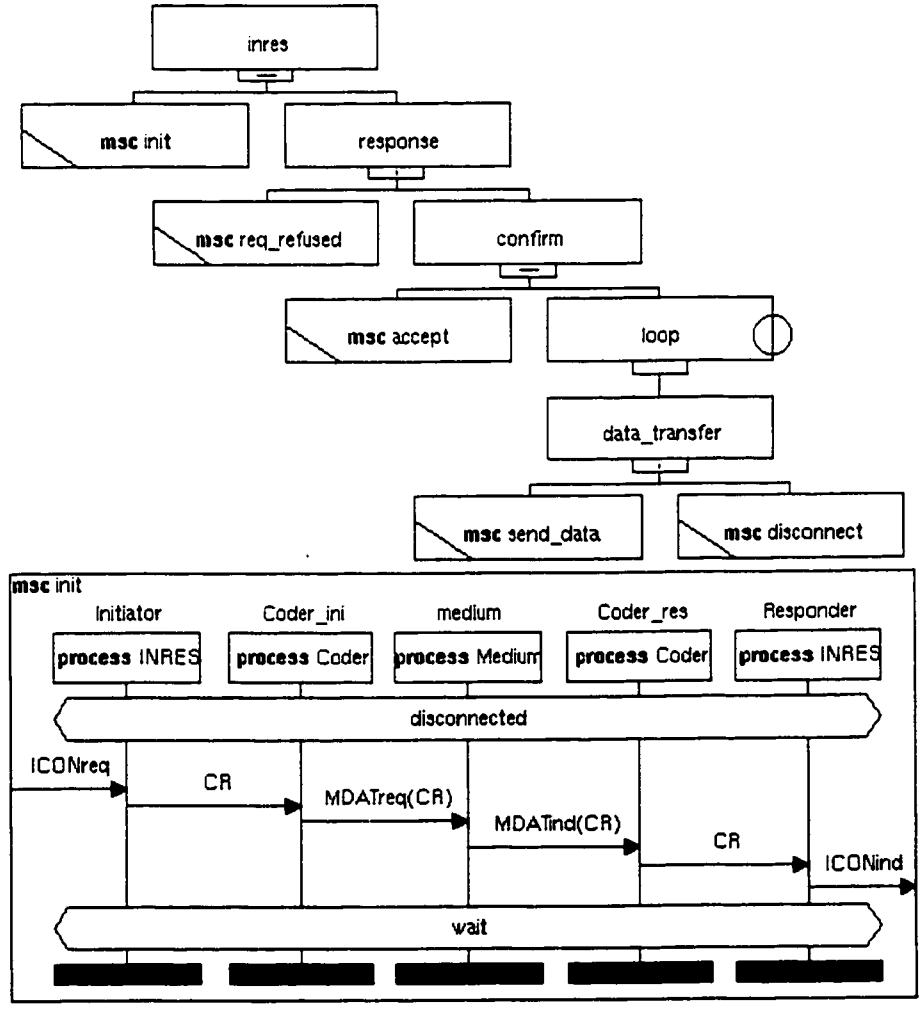


Figure 6.21.INRES MSC specification.

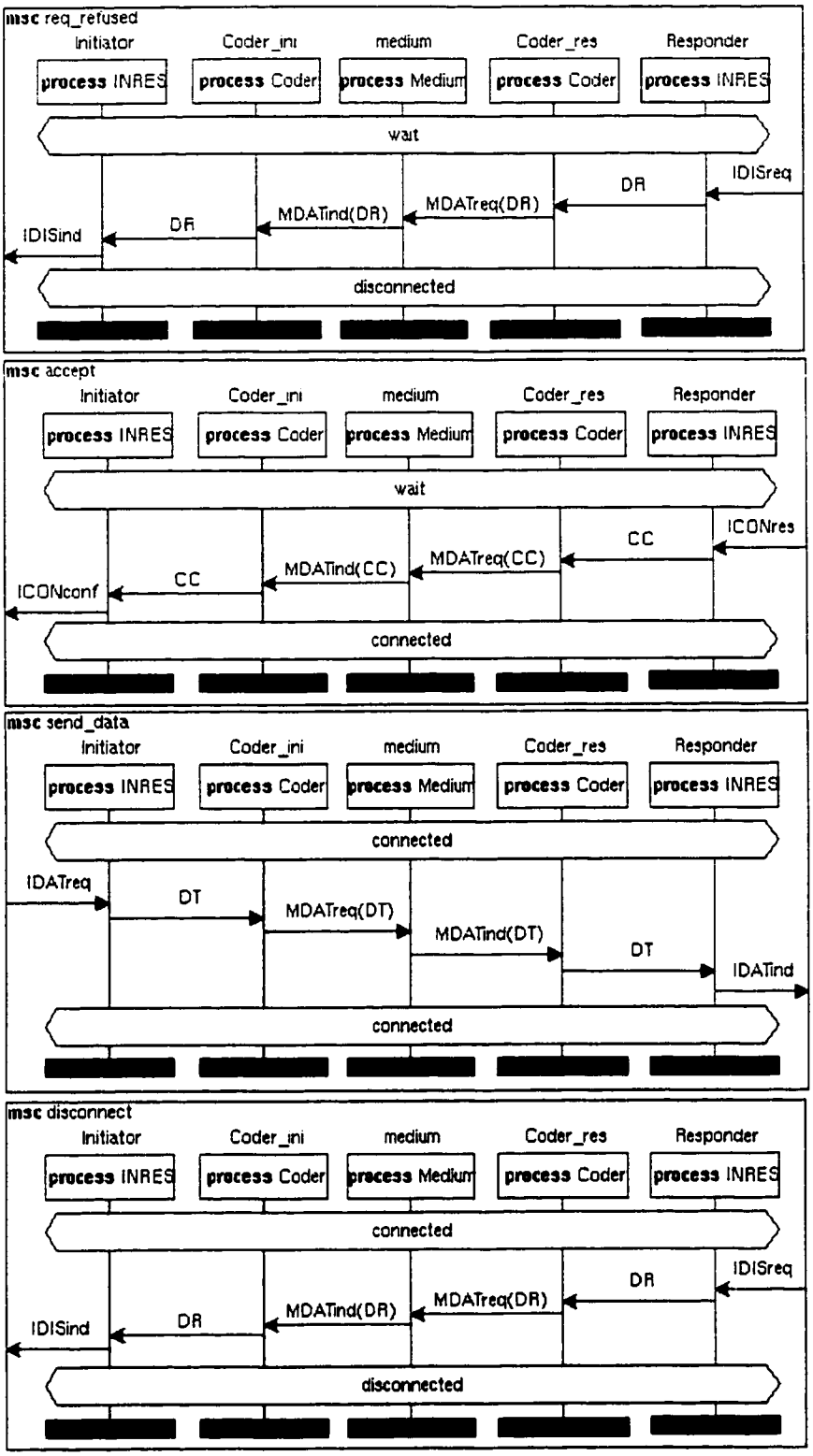


Figure 6.21.INRES MSC specification. (cont...)

SDL ARCHITECTURE

Figure 6.22 shows the SDL architecture of the INRES protocol, which consists of two blocks *INRES* and *Medium*. The two blocks communicate with each other through channel *Channel2*, and block *INRES* exchanges signals with the environment through channel *Channel1*.

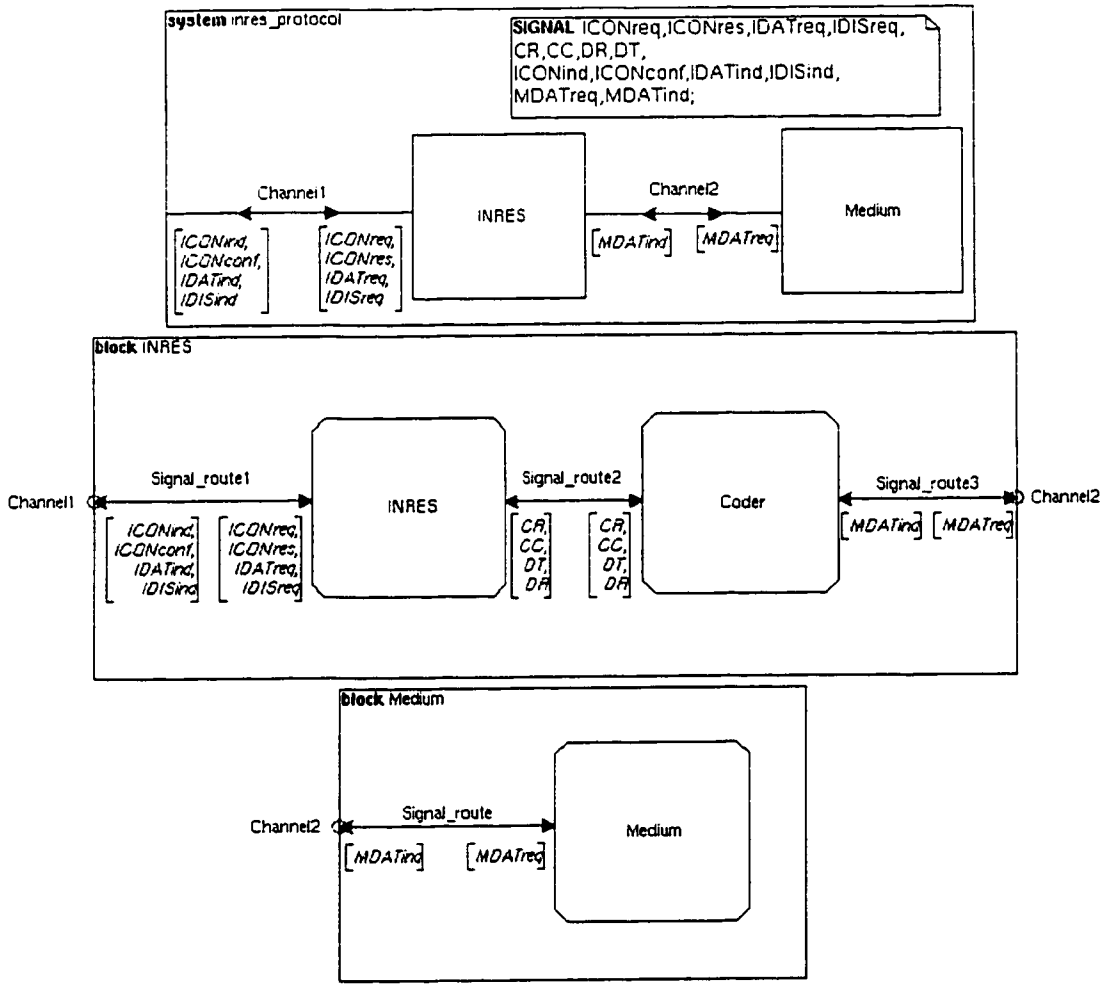


Figure 6.22. INRES SDL Architecture.

The block *INRES* consists of two processes *INRES* and *Coder*. Processes *INRES* and *Coder* communicate through *Signalroute2*. *INRES* communicates with the block environment through *Signalroute1*, which is connected to channel *Channel1* at the block

border. *Coder* communicates with the block environment through *Signalroute3*, which is connected to channel *Channel2* at the block border.

The block *Medium* consists of one process *Medium* communicating with the block environment through *Signal_route*. *Signal_route* is connected to channel *Channel2* at the block border.

GENERATED SDL PROCESSES

Figure 6.23 shows the generated SDL behavior of process *INRES*, which play the role of the instance *Initiator* and the instance *Responder*. As shown, there is no shared behavior between the two instances, so, there is no need for saving their pids.

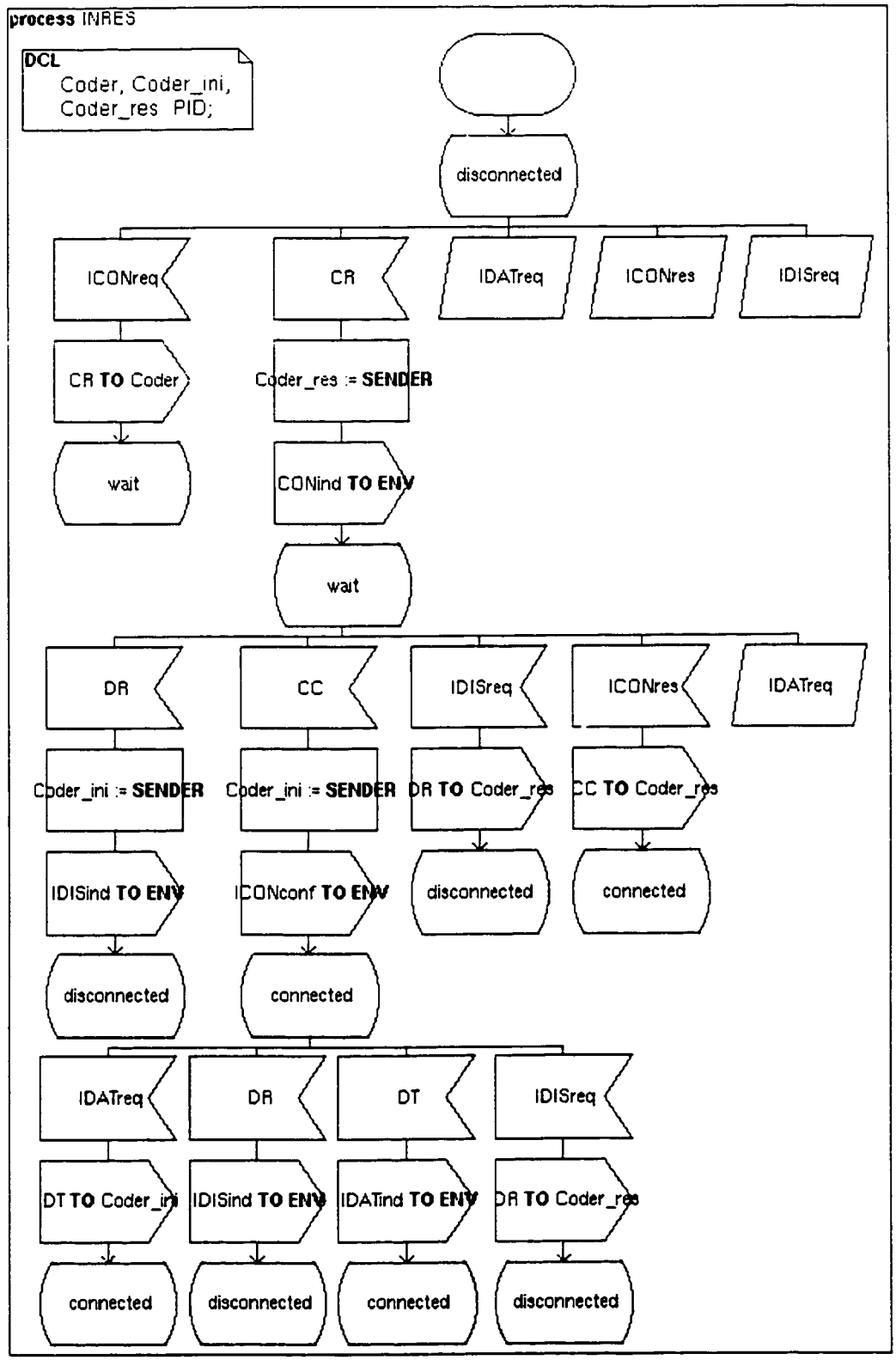


Figure 6.23. Generated process behavior (*INRES*) for MSC specification in Figure 6.21.

Figure 6.24 shows the generated SDL behavior of process *Coder*, which play the role of the instance *Coder_ini* and the instance *Coder_res*. As shown, there is no shared behavior between the two instances, so, there is no need for saving their pids.

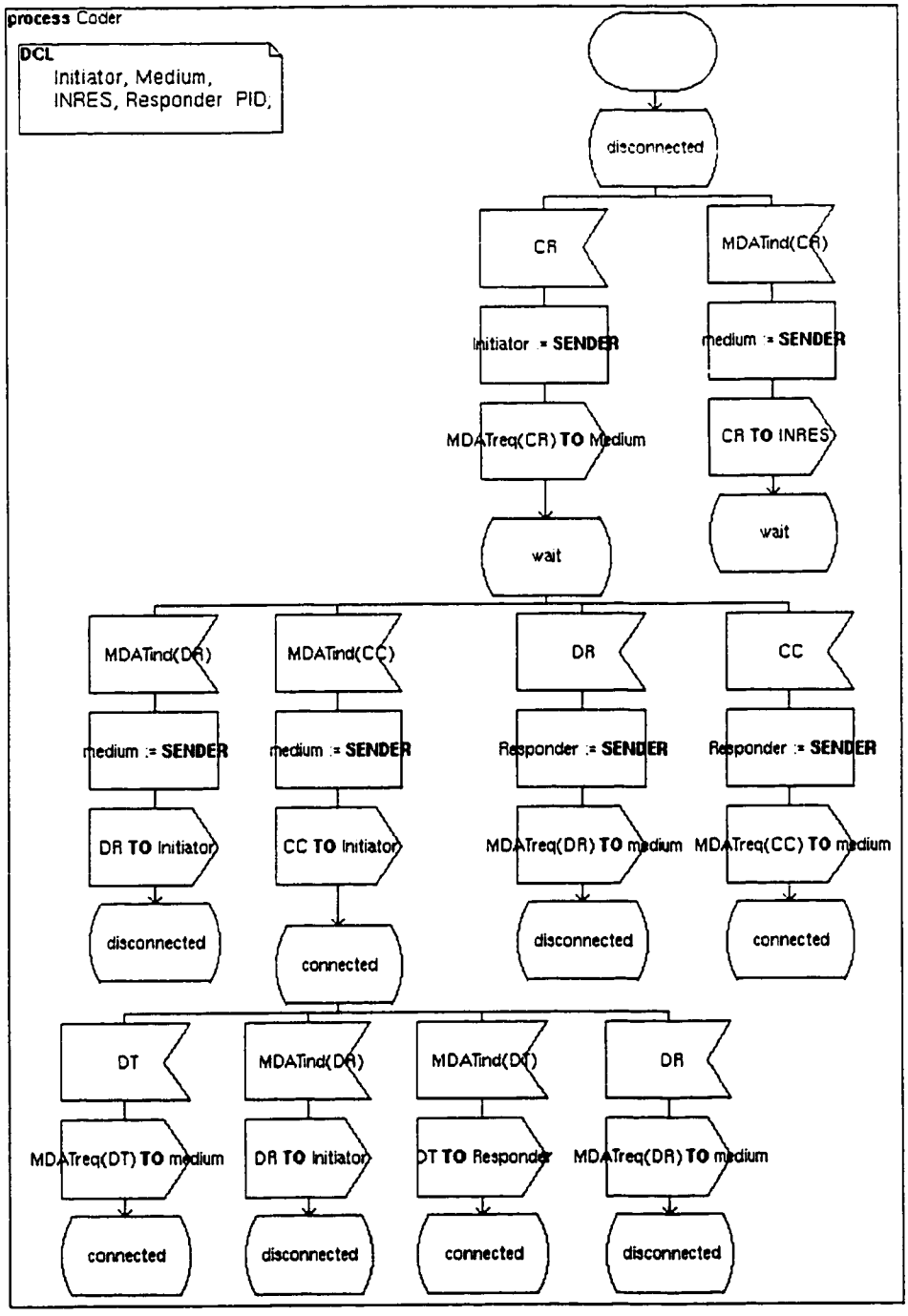


Figure 6.24.Generated process behavior (*Coder*) for MSC specification in Figure 6.21.

The generated SDL behavior of process *Medium* is shown in Figure 6.25. The process behavior is composed of only one instance behavior *medium*.

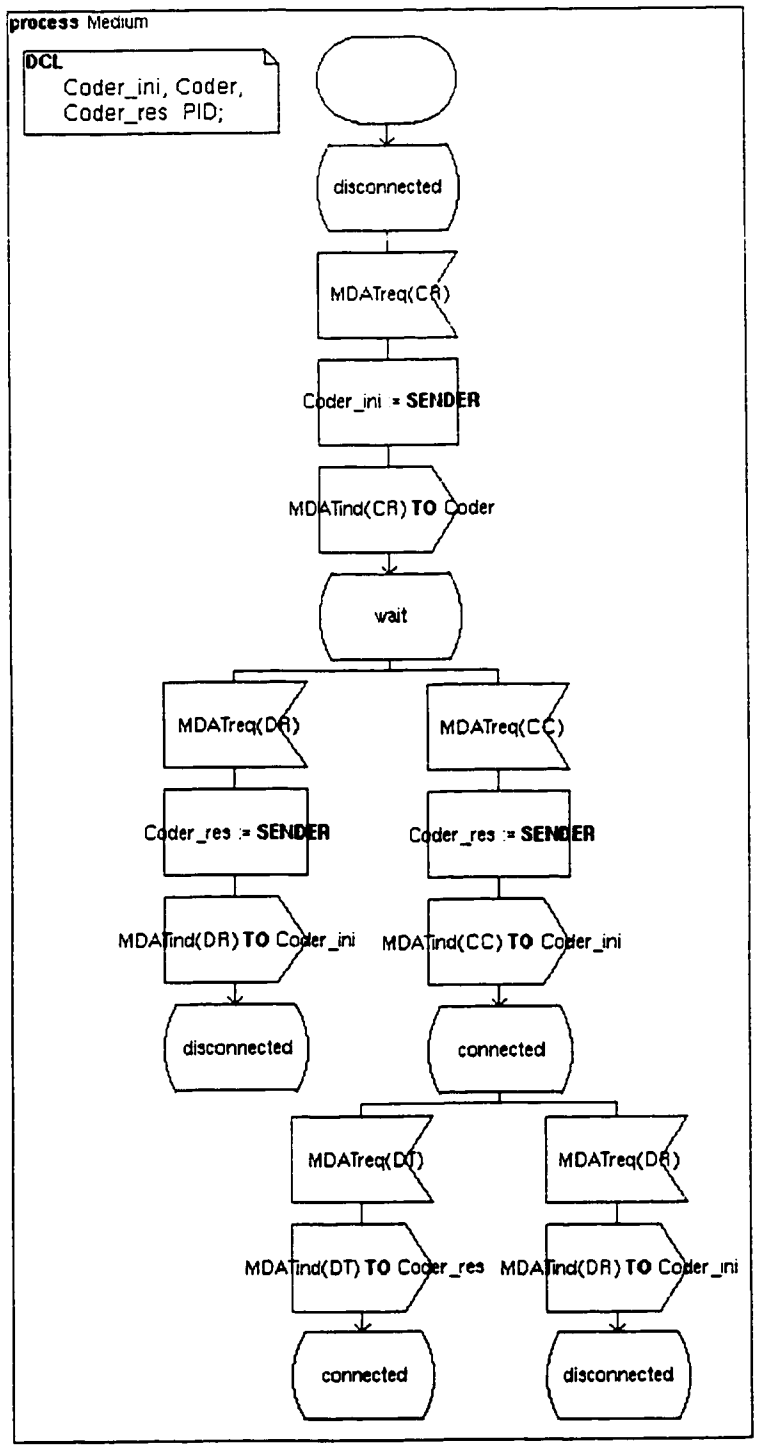


Figure 6.25.Generated process behavior (*Medium*) for MSC specification in Figure 6.21.

6.6 MSC2SDL LIMITATIONS AND RESTRICTIONS

The current version of MSC2SDL tool does not support full MSC'96, and has some restrictions and limitations, which are listed below.

MSC

- Process type instances are the only instances recognized by the tool. System, block and service instance types are not supported by the current version.
- In addition to instance names, process names must be specified within the given MSC specification.
- Time-out events are not allowed in MSC coregions.
- Instance decomposition, MSC reference, message lost and message found are not implemented in the current version.
- Unspecified reception and process divergence are not implemented in the current version.

SDL

- Communication between processes must be explicitly specified in the given SDL architecture.
- The given SDL architecture must not exceed one level of block hierarchy.

CHAPTER 7

CONCLUSION

The main contribution of our work is an approach to tighten the bridge between the requirement phase and the design phase in the software process. We have chosen MSC language as a requirement specification language, and SDL language as a design specification language. We have proposed an automatic approach for synthesizing a consistent SDL specification from an MSC specification. The approach guarantees that the generated SDL specification is free of deadlocks, non-local choices, divergences and unspecified receptions. The approach also checks for the implementability of the MSC specification under the target SDL system. Therefore, the approach guarantees consistency between the MSC specification and the SDL specification. Further, there is no need for more validation of the SDL specification against the MSC specification. To apply our approach, the target SDL architecture is required in addition to the MSC specification. In general, for a given MSC, the approach generates different SDL behaviors for different SDL architectures. Some real case studies have been presented in Chapter 6 for the evaluation of our approach and tool.

More specifically, our contribution consists of: extending the basic approach, introducing HMSC and multi-instances and addressing the consistency between MSC and SDL as well as MSC semantic errors. We have improved the basic approach, which was introduced in [4], and extended the basic approach to cover MSC timers, coregions and inline expressions. Furthermore, we have provided an approach to translate HMSC

operators, sequential, alternative and iterative, into SDL. In Addition, our approach manages multi-instances MSC specifications. We have also addressed the inconsistency between MSC instance identifications and SDL instance identifications. Finally, we have provided MSC semantic error detection approaches, which detect deadlocks, process divergences, non-local choices, unspecified receptions and the implementability of a MSC specification under the target system.

Our approach is very promising for telecommunications software development. The software process for telecommunications systems will be supported by full systematic tools that will ensure the quality of the product as well as reduce the development time.

Suggested future work is the investigation of the MSC parallel operators. Furthermore, the investigation of an incremental approach where the given MSC specification will enrich the existing SDL behavior is highly recommended.

MSC2000 has significantly improved MSCs. Data types, time constraints and control flow have been introduced and MSC specifiers can now describe their requirements more precisely. Therefore, our approach should be extended to take into accounts MSC2000 and SDL2000.

From a theoretical point of view, the correctness of our translation algorithms has to be examined formally.

REFERENCES

[1] A. Behforooz and F. J. Hudson, *Software Engineering Fundamentals*, Oxford University Press, New York, 1996.

[2] B. B. Welch, *Practical Programming in Tcl and Tk*, Printice-Hall, Upper Saddle River, New Jersey, 1997.

[3] E. Rudolph, J. Grabowski and P. Graubmann, "Tutorial on Message Sequence Charts (MSC'96)", Tutorial of the FORTE/PSTV'96 conference in Kaiserslautern, Germany, 1996.

[4] 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.), SDL'97: Time for Testing - SDL, MSC and Trends, Proceedings of the eighth SDL Forum, Evry, France, Sept. 22 - 26, 1997.

[5] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[6] 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.

[7] H. Ben-Abdallah and S. Leue, "Syntactic Analysis of Message Sequence Chart Specifications", Technical Report 96-12, University of Waterloo, Electrical and Computer Engineering, Nov. 1996.

[8] ITU, "Recommendation Z.100 - Specification and Description Language (SDL)", 1993.

[9] ITU, "Recommendation Z.106 - Common interchange format for SDL", 1996.

[10] ITU, "Recommendation Z.120 - Message Sequence Chart (MSC)", 1996.

[11] J. Ellsberger, D. Hogrefe and A. Sarma, *SDL*, Printice-Hall Europe, Hertfordshire, 1997.

[12] J. Persson, "MSC Transformations", MASC thesis, Department of Communication Systems, Lund Institute of technology Instructor, Sweden, 1998.

[13] J. P. Bowen, A. Fett and M. G. Hinchey, "The Z Formal Specification Notation", in ZUM'98 meeting, Springer-Verlag, LNCS 1493, 1998.

[14] J. Rumbaugh, I. Jacobon and G. Booch, *The Unified Modeling Language reference Manual*, Addison-Wesley, 1999.

[15] Kernighan, B. W. and D. M. Ritchie, *The C Programming Language*, Printice-Hall, Englewood Cliffs, New Jersey, 1978.

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

[17] ObjectGEODE, Verilog, Toulouse, France, 1996.

[18] SDT, Telelogic, Malmö AB, Sweden, 1996.

[19] 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, Proceedings of the ninth SDL Forum, Montreal, Quebec, Canada, June 21 - 25, 1999.