An Execution Semantics for MSC2000

by

Gerardo Padilla

August 2000

Thesis supervisor: Professor Bengt Jonsson

ABSTRACT

Scenario-based specifications such as *Message Sequence Charts*, ITU-T MSC 2000 Z.120 (11/99), offer an intuitive and visual way of describing, for example, requirements. Such specifications focus on message exchange among communicating entities. We present an execution model for the *Message Sequence Charts* defined by an *Abstract Execution Machine* (AEM) whose features include: basic MSC (bMSC), inline expressions, High level MSC (HMSC) and data. The AEM can be used in two different ways: Accepting or generating traces. In the former case the AEM can be used as a tester, in the latter as a test generator. An example of test generation is presented.

Thesis approved: ______________________________
Thesis supervisor

____
Title and department

____
Date

An Execution Semantics for MSC2000

by

Gerardo Padilla

August 2000

Thesis supervisor: Professor Bengt Jonsson

To my parents, Maria Luisa and Enrique

ACKNOWLEDGMENTS

I had the privilege to develop the thesis at Uppsala University, where Bengt Jonsson functioned as my supervisor in Sweden. For his generosity and competent advice, regarding both research and other aspects of life, in those *early* mornings, I am deeply grateful. I wish to thank Anna Erikson my supervisor at Telelogic AB, for her support and advice throughout the work on this thesis. I thank my supervisor in Mexico, Manuel Guzman, who encouraged me to perform this work.

Thanks to all people who have discussed and commented this work at various stages: Parosh Abdullah, Marcus Nilsson and Mattias Lange. Thanks to Jan Docekal and Andre Engels for their help answering my questions.

I also specially thank to Nacho Rangel and Margret Johansson for their help and advice throughout my stay in Sweden.

This project has been supported by CONACyT-Mexico, Telelogic AB and Uppsala University, Sweden.

ABSTRACT

Scenario-based specifications such as *Message Sequence Charts*, ITU-T MSC 2000 Z.120 (11/99), offer an intuitive and visual way of describing, for example, requirements. Such specifications focus on message exchange among communicating entities. We present an execution model for the *Message Sequence Charts* defined by an *Abstract Execution Machine* (AEM) whose features include: basic MSC (bMSC), inline expressions, High level MSC (HMSC) and data. The AEM can be used in two different ways: Accepting or generating traces. In the former case the AEM can be used as a tester, in the latter as a test generator. An example of test generation is presented.

Contents

Page

LIST OF TABLES . . . viii

LIST OF FIGURES . . . ix

CHAPTER

1 Introduction . . . 1
1.1 An introductory example . . . 2

2 The MSC2000 Standard . . . 4
2.1 Introduction . . . 4
2.2 Basic elements . . . 4
2.2.1 Instances . . . 4
2.2.2 Messages . . . 4
2.2.3 Inline expressions . . . 5
2.2.4 High Level MSC . . . 9
2.2.5 Data . . . 10
2.2.6 Time . . . 16

3 Formalization . . . 21
3.1 Introduction . . . 21
3.2 Previous work . . . 21
3.3 Inconsistencies in the recommendation . . . 22
3.4 Formalization of the basic MSC . . . 22
3.4.1 Structural elements . . . 22
3.4.2 Behavioral elements . . . 22
3.4.3 The partial order . . . 23
3.5 Formalization of the Inline Expressions . . . 23
3.6 Formalization the High Level MSC . . . 24
3.7 Formalization of Data . . . 24
3.7.1 Basic elements . . . 24

3.7.2 State . . . 25
3.7.3 General semantics . . . 26

4 Execution Model for the basic MSC . . . 27
4.1 Introduction . . . 27
4.2 Basic concepts . . . 27
4.2.1 Event Structure . . . 27
4.2.2 Instance Reference . . . 28
4.3 Abstract Execution Machine . . . 28
4.3.1 Event Memory . . . 28
4.3.2 The *enabled* predicate . . . 28
4.3.3 The *action* operation . . . 30
4.3.4 Operational rules . . . 30
4.4 Example: The operation of the AEM without data . . . 31
4.4.1 Data space . . . 32
4.4.2 The *generate/accept* function . . . 32
4.4.3 Snapshot . . . 32
4.4.4 The *update* action . . . 32

5 Execution model for the basic MSC with inline expressions . . . 35
5.1 Introduction . . . 35
5.2 The extended *instance reference* . . . 36
5.2.1 The *instance reference*state . . . 36
5.2.2 The *instance reference*counter . . . 36
5.2.3 Decision set . . . 36
5.2.4 The inline expression activation . . . 37
5.2.5 The *instance reference*relationships . . . 39
5.2.6 Inline expression counter set . . . 39
5.2.7 The operation *clean* . . . 39
5.2.8 The operation *stopExec* . . . 40
5.2.9 The operation *evalLoop* . . . 41
5.2.10 The extended *action* operation . . . 41
5.2.11 The extended operational rules . . . 41
5.3 Example: The operation of the AEM with inline expressions . . . 41

6 Execution Model of the HMSC . . . 45
6.1 Introduction . . . 45
6.2 Approach . . . 45
6.3 The new elements in the AEM . . . 46
6.3.1 The *node reference* . . . 46

6.3.2 The extended *clean* operation 46
6.3.3 The extended operation rules 47
6.4 The operation of the AEM with HMSC 48

7 Applications 49
7.1 Introduction 49
7.2 Previous work 49
7.3 The TTCN language 50
7.3.1 The declarations and constraints part 50
7.3.2 The dynamic part 51

8 Concluding remarks 52

List of Tables

Table Page

4.1 Description of the *enabled* predicate 29
4.2 Description of the *action* operation. 30
4.3 The AEM operational rules. 31
4.4 Execution of the bAEM using the MSC with data. 34
4.5 Execution of the bAEM using the MSC with data, this table only describes the event states. 34
5.6 Description of the extended *action* operation. 42
5.7 The AEM operational rules. 43
6.8 The AEM operation rules. 47

List of Figures

Figure		Page
1.1	Basic Message Sequence Chart	2
1.2	Basic Message Sequence Chart with an explicit representation of send and receive events.	2
2.3	Elements inside Message Sequence Chart	6
2.4	Example of inline expression in the Message Sequence Chart	7
2.5	Common preamble in both inline sections.	7
2.6	Example of HMSC with all basic MSCs.	9
2.7	Example of data declaration inside an MSC document.	13
2.8	Example of **def** and **undef** qualifier.	14
2.9	Example showing some event states.	15
2.10	Example of relative and absolute time points.	18
2.11	Example of relative and absolute measurements.	18
2.12	Example of a Timer.	19
2.13	Example of a time interval.	20
4.14	Visualization of the Event Structure.	27
4.15	The AEM and the event structure.	29
4.16	MSC Example.	31
4.17	MSC Example with data.	33
5.18	The interpretation of the inline expression in the AEM.	35
5.19	The *instance reference*states.	36
5.20	Example of common preamble between two alternatives in the MSC.	37
5.21	The inline expression activation in the AEM.	38
5.22	The relations among *instance references*.	40
5.23	Example MSC with inline expressions and the corresponding Event Structure	41
5.24	Execution example.	44
6.25	A HMSC and the instances presented in some nodes.	46
6.26	Example of the AEM and HMSC (initial step).	48
6.27	Example of the AEM and HMSC (alternative)	48
7.28	A TTCN tree and its corresponding MSC.	51

Chapter 1

Introduction

Message Sequence Charts (MSCs) is a graphical and textual standarized language[1] for the description and specification of the interactions between system components. The main area of applications for MSCs is as overview specification of the communication behavior of concurrent, message exchanging systems. Message Sequence Charts may be used for requirement specification, interface specification, simulation and validation, test case specification and, documentation.

There are commercial and research tools that support MSC. One example is the Telelogic Tau Suite that supports not only MSC but also SDL(Standartd Description LAnguage) and TTCN (Tree and Tabular Combined Notation). The Tau toll is used to develop software for telecommunications and embedded applications. The MSCs are used in this tool to capture requirements, to record traces of SDL model simulations and as source for test generation.

An important functionality of Tau is the ability to define and execute test sequences.(Derived from SDL specifications). However there is not current support to generate test sequences from the MSCs requirements. The next question were stated: How can the MSCs be used to generate test sequences ?. This question describes the main goal of this project.

There is previous work related in this field, e.g., the work developed by Grabowski in the Test Generation from MSC [**?**] and some other projects related in the test generation using both SDL and MSC. These works are developed using MSC'96 or previous releases.

[1]ITU-T MSC2000 Z.120 (11/99).

1.1 An introductory example

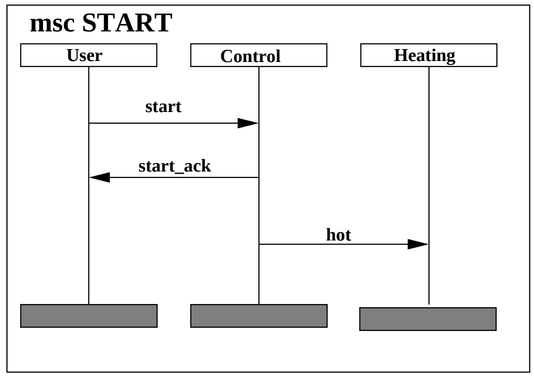


Figure 1.1: Basic Message Sequence Chart

Figure 1.1 describes a scenario where a toaster machine starts to work. Three different entities, named *instances*, interact in the scenario: **User**, **Control** and **Heating**. Each instance has three main elements, head, end and, time axis. The *instance head* and *instance end* describe the existence of the instance, not the creation or destruction of it. The arrows represent the message exchanges between two instances, the arrow head denotes the reception and the arrow tail denotes the sending. The set of messages is: **start**, **start_ack**, and **hot**. Each message arrow represents two events: the sending and the reception. Let $!m$ denote the send event and $?m$ the receive event for message m. Figure 1.2 shows the **msc START** where the sending and receiving events are highlighted as circles.

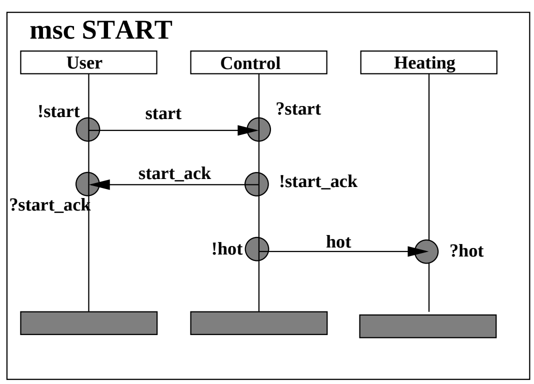


Figure 1.2: Basic Message Sequence Chart with an explicit representation of send and receive events.

1.1.0.1 The partial order of events

For each instance the time axis describes a total order among events called *instance order*. An MSC defines a partial ordering of events composed from:

- **Instance order:** The events are ordered over the axis time in every instance; there are exceptions as the coregions and inline expressions but these elements will be explained later. For every instance we have a sequence of events built from the instance head to the instance end. For example, instance **User** has the sequence of events **!start,?start_ack** , the instance **Control** has the sequence **?start, !start_ack, !hot**, and the instance **Heating** has the sequence **?hot** . The nature of the sequence defines a total order over the contained elements.

- **Send - receive relation:** There is a one-to-one correspondence between sending and reception of each message represented by the arrows e.g, in Figure 1.1, the event of sending message **!start** is related to the receiving event **?start**. This relation can be described by a bijective function from each send to its corresponding receive event.

> The instance order, together with the send-receive relation define a partial order over the events in the MSC. Let e_1 and e_2 be two different events, we say that e_1 precedes e_2, denoted by $e_1 < e_2$ if
>
> • e_2 and e_1 belong to the same instance and e_1 appears before than e_2.
>
> • e_1 is the send event and e_2 is the corresponding receive event.

MSCs describe a set of possible traces (test sequences). A trace is a sequence of events. The MSCs in Figure 1.1 describes the next set of traces:

Trace 1: start!, start?, start_ack!, start_ack?, hot!, hot?

Trace 2: start!, start?, start_ack!, hot!, start_ack?, hot?

Trace 3: start!, start?, start_ack!, hot!, hot?, start_ack?

Chapter 2

The MSC2000 Standard

2.1 Introduction

In this chapter we shall present the most important MSC 2000 features. The features presented are:

- The MSC basic elements, such as instances, messages, timers, condition, actions and coregions.
- Inline expressions.
- High level MSC.
- Data and time concepts.

2.2 Basic elements

2.2.1 Instances

Instances are reactive entities whose communication behavior is described by the MSCs. Each instance can store information in local variables (called dynamic variables in MSC2000). Instances may be created and destroyed dynamically. The message exchanging is the only mean of communication among instances. Within the instance body the ordering of events is specified (Figure 2.3).

2.2.2 Messages

Messages are the units of information exchange between instances. A message can be as simple as a signal or as complex as a sophisticated data packet. Usually a message consist of an identifier and zero or more data parameters. Two different events are associated to the message: the send and receive events (Figure 2.3).

2.2.2.1 Timers

Timers are mechanisms to count time units. Each timer is also a reactive entity which belongs to some instance [1]. Three different events are associated to the timers: timeout, setting, and stopping the timer (Figure 2.3).

2.2.2.2 Conditions

Conditions are multi-instance events, which may span over several instances. There are two types of conditions: setting and guarding conditions. The setting conditions describe the global state of the system (as labels), the guarding conditions must be local and attached to one instance. The guard conditions are used to enable or disable sections in inline expressions. The guarding conditions can contain predicates associated to data in the MSCs (Figure 2.3).

2.2.2.3 Actions

Actions are events that can have either informal text associated to it (labels), or formal data statements. An action describes an internal atomic activity of an instance. When an action contains data statements, the event modifies the state by the evaluating each statement concurrently. This concurrency reflects the atomicity of an action (Figure 2.3).

2.2.2.4 Coregions

A coregion is a special mechanism introduced to describe unordered sets of events, i.e. to remove the order described in the time axis. A coregion is part of the instance axis; the events specified within that part are assumed to be unordered in time (Figure 2.3). A coregion covers, for example, the practically important case of two or more incoming messages where the ordering of reception may be interchanged.

2.2.3 Inline expressions

Inline expressions provide a mean to formulate the composition of MSCs within the MSC language. The use of inline expressions reduces the need for several MSCs to

[1] This element will be explained in the next sections

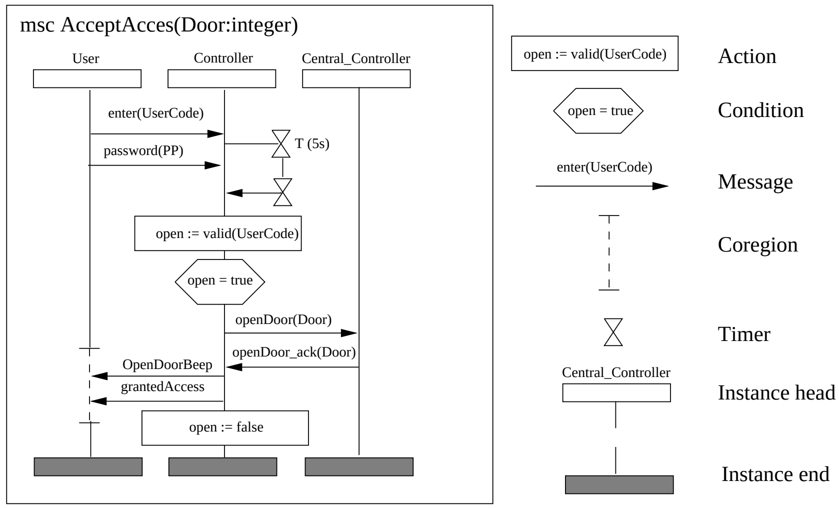


Figure 2.3: Elements inside Message Sequence Chart

describe complex behaviors. Graphically an inline expression consists of an *inline expression* symbol that is attached to a number of instances (at least one).This inline expression symbol contains in the left-upper corner one of the keywords **alt, par, seq, exc, opt** or **loop**. These keywords indicate the composition operation that is described by the inline expression:

- Weak sequential composition (seq)
- Alternative composition (alt)
- Parallel composition (par)
- Iteration (loop)
- Optional composition (opt)
- Exceptional composition (exc)

Both alternative and parallel composition can have any finite, positive number of inline sections (the inline section is another MSC). These sections are all drawn inside the inline expression symbol and they are separated by a dashed vertical line (Figure 2.4).

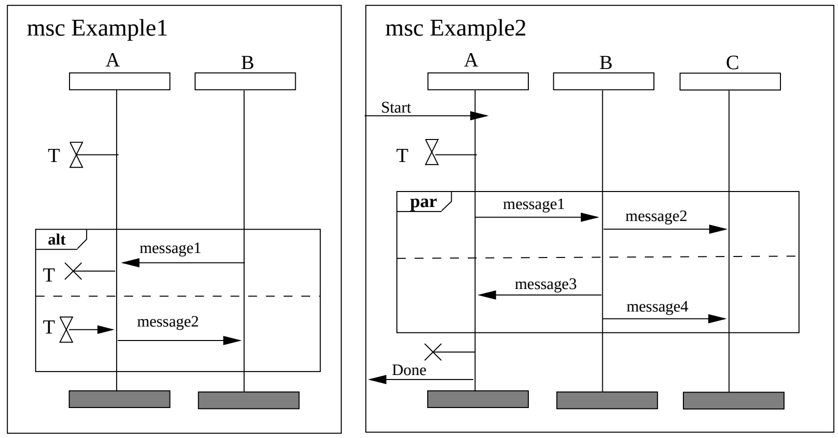


Figure 2.4: Example of inline expression in the Message Sequence Chart

2.2.3.1 Alternative composition

The alternative composition defines alternative executions of inline sections. This means that if several inline sections are meant to be alternatives only one of them will be executed. In the case where alternative inline sections have common preamble (the same set of traces) the choice of which inline section will be executed is performed after the execution of the common preamble (until one section can really be selected). Figure 2.5 presents an MSC containing a common preamble in the two inline sections. Notice that the initial set of messages is the same in both sections.

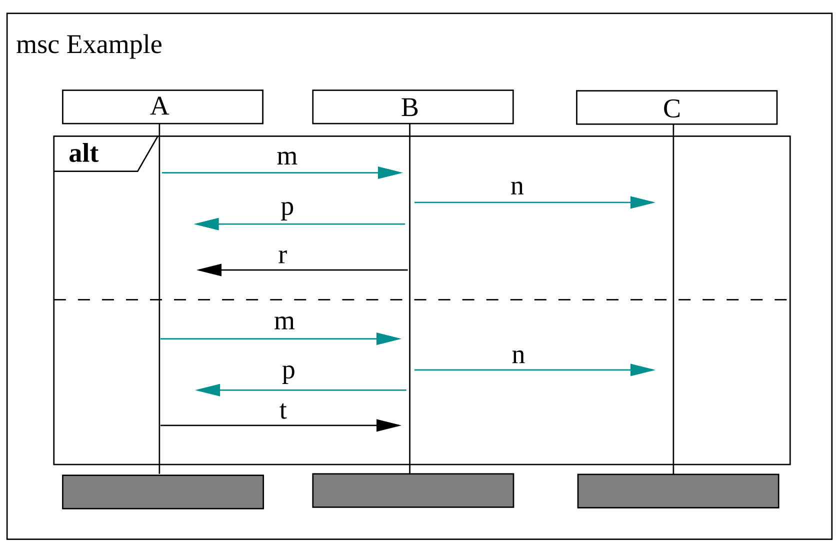


Figure 2.5: Common preamble in both inline sections.

2.2.3.2 Parallel composition

The parallel composition defines the parallel execution of inline sections. This means that all events within the parallel MSC sections will be executed, where the only restriction that the event order within each section will be preserved.

2.2.3.3 Iteration

An inline loop expression has exactly one inline section. The keyword **loop** is followed by a loop boundary. This loop boundary refers to the number of repetitions of the inline section. The loop boundary, if present, indicates the minimal and maximal number of inline sections.

The most basic form is $\mathbf{loop} < \mathbf{n}, \mathbf{m} >$ where **n** and **m** are expressions of type natural numbers. This means that the operand may be executed at least **n** times and at most **m** times. The expressions may be replaced by the keyword **inf**, like $\mathbf{loop} < \mathbf{n}, \mathbf{inf} >$. This means that the loop will be executed at least **n** times. If the second operand is omitted like in $\mathbf{loop} < \mathbf{n} >$ it is interpreted as $\mathbf{loop} < \mathbf{n}, \mathbf{n} >$. Thus $\mathbf{loop} < \mathbf{inf} >$ means an infinite loop. If the loop bounds are omitted like in **loop**, it will be interpreted as $\mathbf{loop} < \mathbf{1}, \mathbf{inf} >$. If the first operand is greater than the second one, the loop will be executed 0 times. The loops can be parameterized using static variables (explained in the next sections).

2.2.3.4 Optional composition

The optional composition is the same as an alternative where the second operand is the empty inline section.

2.2.3.5 Exception composition

The exceptional composition is a compact way to describe exceptional cases in an MSC. The meaning of the operator is that either the events inside the inline section are executed and then the MSC is finished or the events following the section are executed. The exceptional inline expression can thus be viewed as an alternative where the second operand is the entire rest of the MSC. All exception inline expressions must be shared by all instances in the MSC.

2.2.3.6 The guarding inline sections

The guarded sections contain an initial local condition (guard). If this condition evaluates to false then the entire corresponding section is disabled, the condition must be the first event in the section, and all events inside any guarding section must be causally dependent on the guarding condition.

2.2.4 High Level MSC

The high level MSC (HMSC) provides a mean to graphically define how a set of MSC can be combined. In Figure 2.6 a complete example of a HMSC is presented. The example describes different scenarios for a Toaster machine, taken from [9]. The HMSC is the diagram named **msc TOASTER** . The HMSC is a directed graph[5] where different types of nodes can be found, e.g., the *start symbol* represented by ▽, *connection points* represented by ◯, and MSC references represented by □. There are also other nodes, such as conditions, end symbols, and parallel frames.

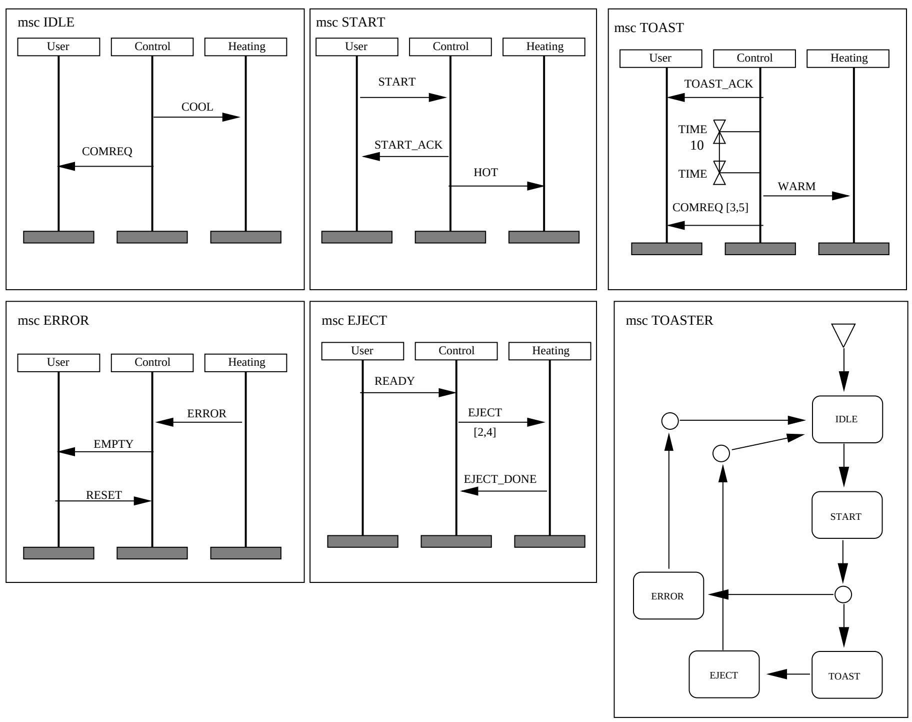


Figure 2.6: Example of HMSC with all basic MSCs.

The flow lines connect the nodes in the HMSC and they indicate the sequencing that is possible among the nodes in the HMSC (Figure 2.6). In our example, the initial scenario that occurs in our system is the MSC IDLE. After the occurrence of this scenario, the MSC START follows. Following the occurrence of the MSC START, two different alternative scenarios are possible, MSC TOAST or MSC ERROR. If the MSC ERROR occurs, the next available scenario is the MSC IDLE. Loops can be also represented, the loops can be interpreted as a specification of a continuous execution of the system. The start symbol only shows where the system can start, there is no additional meaning to this element. According to the recommendation Z.120 [5], the connection points do not have meaning, they can be used to improve the readability of the chart.

2.2.4.1 Weak vertical composition

The *weak vertical composition* means that all the events in the instance A appearing in top MSC finish before any event in the second MSC occurs (instance A must appear in both MSCs). The *strong vertical composition* means that all the events in the top MSC finish before any event in the second MSC occurs.

2.2.4.2 Parallel composition

The parallel composition, also called horizontal composition, means that the multiple MSC "runs" in parallel. There is no restriction among the multiples MCSs.

2.2.4.3 Alternative composition

The alternative composition means the choice among different scenarios. The only additional consideration is when the alternatives have a common preamble, the same initial behavior. According to the semantics expressed in the [5] the choice of one alternative is postponed until a real alternative is found.

2.2.4.4 Loops

The Loops are not explicit declared as an operation, but they can be constructed due the fact that the HMSC is a digraph. The meaning of the loop is the resulting of the vertical composition of the last node with the first node, creating the loop.

2.2.4.5 MSC reference

An MSC reference is the label that is found inside the box in the HMSC. This label denotes an MSC, an operations among MSC or a parameterized MSC.

2.2.5 Data

2.2.5.1 The Data approach

In the recommendation MSC2000, one main idea is the openness of the MSC language, meaning that it is not constrained to any data particular language: **The**

MSC can be parameterized to any data language . This means that any complete specification will include two different languages, one for the MSC and the other for the data.

2.2.5.2 Basic concepts

Some assumed basic elements that are included in the recommendation are:

Data type: A data type defines a (possibly infinite) set of values. E.g, the data type DAYS may contain the elements of the set {*Mon, Tue, Wed, Thu, Fri, Sat, Sun* } .

Typed variable: A typed variable is a container for a value of a specific type. A typed variable has a name, i.e., the identifier of the container, and a value, i.e, the actual contents of the container. The value is referred to by using the variable name, e.g., let the variable y have the value 3 then the expression y + 2 denotes the integer value 5. There are two types of variables, static and dynamic. The difference between them will be explained in the next sections.

Wildcard: A wildcard is a special variable used to denote a *don't care* value. The usual symbol is "_". The wildcards must be declared. A wildcard will generate a set of concrete traces corresponding to each uninterpreted trace, where each concrete trace is derived from the uninterpreted trace by substituting a different concrete value for the wildcard. If an expression contains multiple occurrences of a wildcard then each represents a different reference, so that different concrete values will, in general, be substituted for each occurrence.

Pattern: A pattern consists of either a wildcard or a dynamic variable.

Expression: An expression is a data expression which may contain wildcards, dynamic variables, and static variables.

Binding: A binding is similar to an assignment. A bind consists of an expression part and a pattern part that are connected by a bind symbol. The bind symbol is :=. The example below shows equivalent left and right binding

$$x := y + 3, \; y + 3 =: x$$

2.2.5.3 The data inside an MSC

The places where the data can be found are:

- **Data parameters:** The data can be present as data parameter in the sending, reception, timer setting and instance creation event and in the MSC references. The parameters should be valid expressions in the external data language.

- **Predicates:** The data can be presented in predicates inside conditions.

- **Action expressions:** The action expressions are used to manipulate the value of the dynamic variables. Depending on the external data language different expressions can be used as actions expressions.

- **Loop Inline expressions:** The loop inline expressions can contain static variables to specify the bounds of the iteration.

- **Time constraints:** The boundaries that can be imposed to the send and receive events, timer events and msc reference (in the HMSC).

2.2.5.4 Dynamic and static data

There are two different sort of variables: static and dynamic [2].

- A *static variable* is used to parameterize an MSC and is declared in the head of the MSC. These variables can not be modified after the instantiation of the MSC and the scope of this variable is the MSC body.

  The meaning of an MSC reference with actual parameters is *call by value* [5], in which the parameters are substituted by the actual parameters wherever they appear in its body.

  Figure 2.7 presents a parameterized MSC. The static data is declared in the head of the MSC. In the example one static variable " Door" can be used to parameterize the MSC to different users or different doors. The main idea of this scenario is the description of the exchange of messages between a locking system and the user. Using different concrete MSCs different scenarios can be described (In a *concrete MSC* concrete values are assigned to the static variables).

- A *dynamic variable* belongs to an instance and must be declared in the MSC Document. These variables can be modified using the binding mechanism by events in the owning instance. These variables can be assigned and reassigned

[2] The recommendation uses the term *static data* and *dynamic data* to denote the two types of variables

values through **action boxes, message and timer parameters, and instance creation**. The value that a dynamic variable may possess at any point in a trace will, in general, depend upon the previous events in the trace.

Figure 2.7 presents two action boxes where the value of the variable "open" is changed. In the first action box the content is changed depending on the return value of the function "valid". In the second action box the value of the variable "open" is bound to the value "false".

2.2.5.5 Data declaration

The declaration of data mostly takes place in the **MSC document**, the only exception being static variables, which are declared in the MSC head (Figure 2.7).The MSC document declarations include: messages and timers that have data parameters, dynamic variables, wildcard symbols, the data language, and data definitions. Messages that have parameters are declared so that the type and number of parameters are defined. Messages that do not have parameters need not be declared. The data definitions consist of text in the data language that, for example, defines structured types, constants, and functions signatures. It must provide all information required to type check and evaluate data expressions used in MSCs within the scope of the enclosing MSC document. Figure 2.7 presents an example of data declaration.

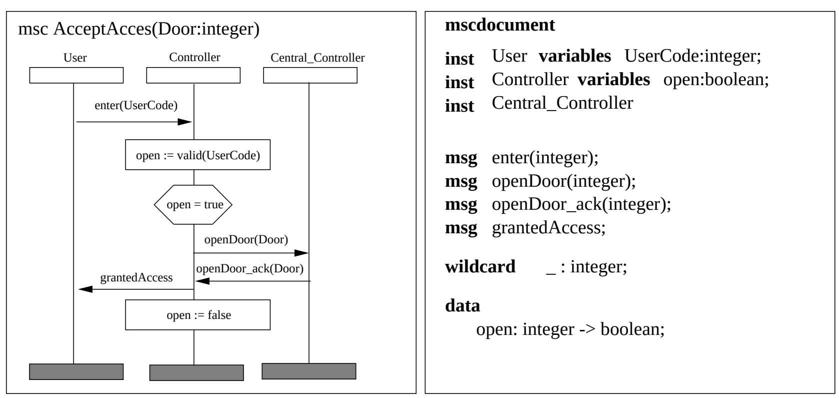


Figure 2.7: Example of data declaration inside an MSC document.

2.2.5.6 Modification of the data

The only two mechanism to manipulate the value of the variables are: *Instantiation* and *Binding*.

The *instantiation mechanism*, used only for static variables, occurs when a MSC reference contains the explicit values to be used in the static variables. The *binding mechanism*, used only for dynamic variables, occurs when an assignment is found in the parameter expression of any event such as action, receive event, etc.

2.2.5.7 Definition of values of dynamic variables

The recommendation establishes the next constraint:

> *In a defining MSC there must be no trace through an MSC in which a variable is referenced without being defined. That is, each variable appearing in an expression must be bound in the state used to compute the value of the expression. The only exception occurs in the utility MSC, references to undefined variables are permitted.*

There is a special qualifier used to denote if any variable is defined in some period of time. Figure 2.7 shows two additional action boxes, the first one contains a **def** statement which is used to indicate that a variable has been assigned some unspecified value; it is the equivalent of a binding of a variable to a wildcard. That is, **def** x is the equivalent of x := _, where _ is a wildcard. In the second action box an undef statement is used to indicate that a variable is no longer bound, i.e. that the variable cannot be legally referenced, or has moved out of scope.

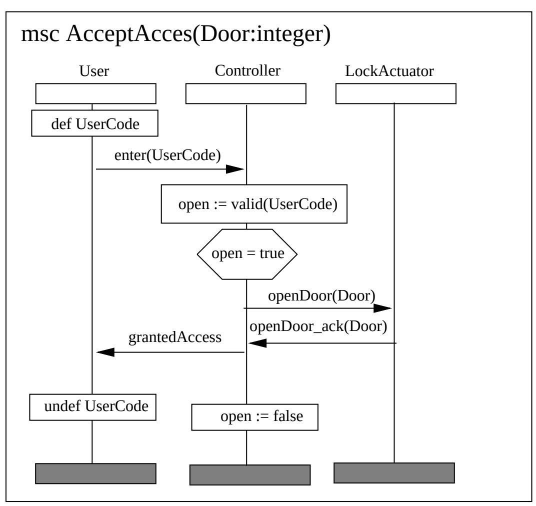


Figure 2.8: Example of **def** and **undef** qualifier.

2.2.5.8 Event State

An **Event State** is a set of bindings in the event in an execution trace. A state associated with a current event is computed from previous states together with the data content of that event. The previous states used to compute the new state depend upon the type of event, all are derived from at least the last non-creating event executed on the same instance as the current event. In addition, for message receiving events and for the first event on a created instance, the state of the corresponding send or creating events is also used in the computation. Effectively, this means that a state is maintained by each instance, and a new state is derived from the instance's previous state together with state information passed to the instance through messaging, or from the parent instance in the case of instance creation (Figure 2.9).

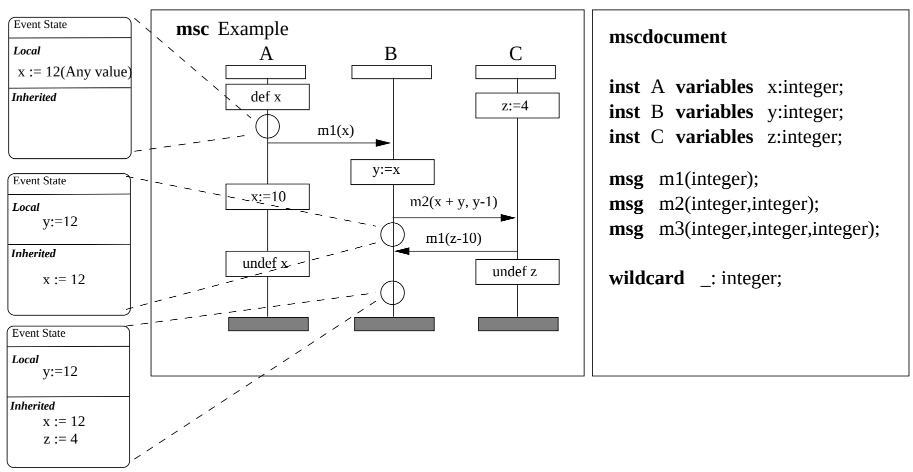


Figure 2.9: Example showing some event states.

2.2.5.9 Accessing variables

Because information is allowed to flow between instances via message passing and instance creation, the state associated with each event may contain bindings to variables not owned by the instance upon which the event occurs. The rules governing the access to the value of variables owned by foreign instances are defined in [5, pags. 58-59]. We can resume them: *If x is not bound either in the old state or in the parameter list, then the binding from the sending event can be inherited. Intuitively, the binding of a variable can be inherited from another instance only if the variable is sent to the instance by appearing in a parameter's expression. However,* *a binding cannot be inherited if the variable is owned and in scope by the receiving instance, as the local binding takes precedence. Thus, the value of a variable can be transmitted by a chain of messages to other instances, so long as each message explicitly references the variable in its parameter list [5].*

2.2.5.10 Assumed Data Types

There are three places in the standard where the MSC language assumes the existence of data types. Boolean valued expressions used in conditions, Natural number expressions used to define loop boundaries, and Time expressions used in specifying timing constraints.

Following the recommendation data approach, these types have to be defined as part of the user's chosen data language and not part of the MSC language.

2.2.6 Time

Time concepts are introduced into MSC to support the notion of quantified time for the description of real-time systems with a precise meaning of the sequence of events in time [5].

The timed interpretation of the MSC assumes the following:

- All events are instantaneous.

- Progress of time is explicitely represented using a special event which represents the passage of time:

  $$\{e_1, t_1, e_2, t_2, e_3, t_3, e_4..\}$$

  The triple (e_1, t_1, e_2) means that after the occurrence of event e_1 time t_1 passes until event e_2 occurs. Events with no time delay, meaning that $t_n = 0$, occur simultaneously, i.e. without any delay.

- Time progress (i.e. clocking) is equal for all instances in a MSC, a global clock is assumed [5, pag. 63].

- It is assumed that time is progressing and not stagnating. Progressing means that after each event in a trace there is eventually a time event. Non-stagnation means that there is an upper bound on the number of normal events between each pair of timed events.

2.2.6.1 The time inside the MSC

There are three main areas where time can be used:

- Time observation, such as the measurements.
- Timer events, such as the starting timer, stopping and timeout event.
- Time constrains, such as the time points and the time intervals.

2.2.6.2 Relative and absolute timing

The **relative timing** uses pairs of events - preceding and subsequent events, where the preceding event enables (directly or indirectly, i.e. via some intermediate events) the subsequent event. Relative timing can be specified by the use of arbitrary expressions of type Time, i.e. referencing parameters, wildcards and dynamic variables. The concrete value of a relative time expression is evaluated once the new state of the event relating to this relative timing has been evaluated.

The **absolute timing** is used to define occurrence of events at points in time that relate to the value of the global clock. Absolute timing can be specified by the use of arbitrary expressions of type Time, i.e. referencing parameters, wildcards and dynamic variables. The concrete values of a time constraint are evaluated at the start of a time interval once the new state of the event relating to the start of the time interval has been evaluated.

2.2.6.3 Time points

Time points are defined by expressions of type Time. The optional absolute time mark, "@" , indicates an absolute timing. The evaluation of a time point yields a concrete quantified time. An event without time constraints can occur at any time. Figure 2.10 presents an MSC where the time execution is constrained. The absolute timing constraints, represented with the "@" symbol denotes that the execution of this MSC must be start when the global clock starts, or the occurrence of this scenario restart the global clock. The total time that this scenario must consume is 498 ms. There is a relative timing constrain in the MSC, the time that the sub-MSC or MSC reference "Get_User_ID" must consume is 221 ms between the first and the last event inside the sub MSC. The only constraint is that the execution of this MSC must be in the period restricted by the global clock.

2.2.6.4 Time observations

Measurements are used to observe the delay between the enabling and occurrence of an event (for relative timing) and to measure the absolute time of the occurrence

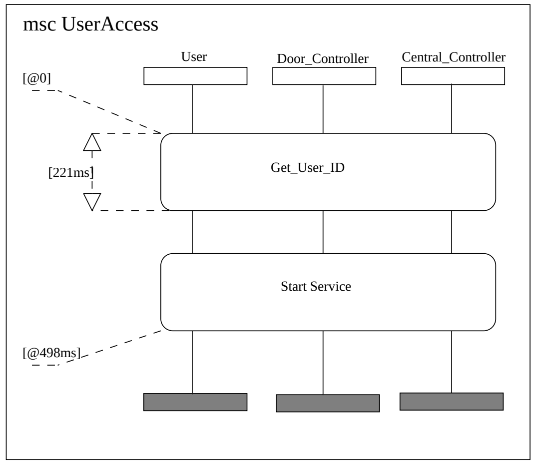


Figure 2.10: Example of relative and absolute time points.

of an event (for absolute timing). In order to distinguish between a relative from an absolute measurement, different time marks (i.e. "@" for absolute and "&" for relative) are used. Measurements can be tied to time intervals. For each measurement, a time variable has to be declared for the respective instance. Figure 2.11

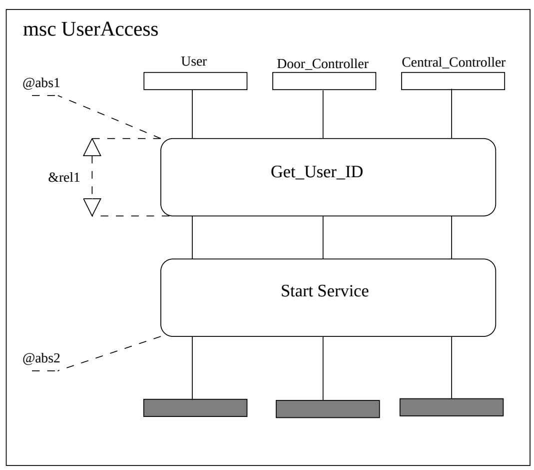


Figure 2.11: Example of relative and absolute measurements.

presents an MSC with both relative and absolute measurements. Remember that these measurements are stored in dynamic variables owned by one instance. In this case, the absolute measurements are stored in the variables **abs1** and **abs2**. The relative measurements are stored in the variable **rel1**.

2.2.6.5 The timer

The timer is a mechanism used to measure the time. A timer is a predefined counter, synchronized with the global clock. We can associate two different internal variables to the timer (these are abstract variables, different to the dynamic and static variables). The **timeout variable** and the **counter variable**, both belong to the Time domain. The default value for the timeout variable is infinite. We assume the existence of the timers in the instances. The manipulation of these variables is performed through the following events (Figure 2.12):

- **Starting timer event:** This event denotes the timer setting, i.e. set the timeout variable to any value described in its parameters.
- **Timeout event:** This event denotes the consumption of the timer signal, i.e. the counter reaches the timeout variable value.
- **Stopping timer:** This event denotes the cancelling of the timer.

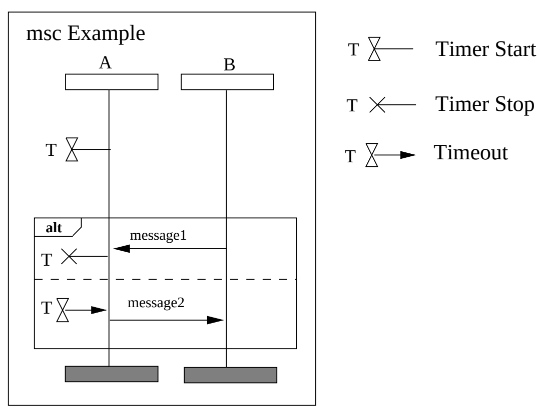


Figure 2.12: Example of a Timer.

2.2.6.6 Time interval

Time intervals are used to define constraints on the timing for the occurrence of events: the delay between a pair of events can be constrained by defining a minimal or maximal bound for the delay between the two events. A time interval does not imply that the events must occur. The fulfillment of a time constraint is validated only if the event relating to the end of that time intervals occurs in the trace. An MSC trace has to fulfill all its time constraints, i.e. if a trace violates a time constraint the trace is illegal. Time intervals can be used for relative timing as well as for absolute timing. Time intervals can be specified by the use of arbitrary expressions of type Time, i.e. referencing to parameters, wildcards, and dynamic variables. The concrete values of a time constraint imposed by a time interval are evaluated at the start of a time interval once the new state of the event relating to the start of the time interval has been evaluated. Within a time interval, either only relative time expressions or only absolute time expressions must be used. Either the minimal, the maximal bound or both bounds are given. An interval must define at least one of the two bounds. Figure 2.13 presents an MSC containing time interval constraints. The interval denotes the maximum and minimum time that constrain the pair of events. In the example the MSC is consistent with all the time constraints presented.

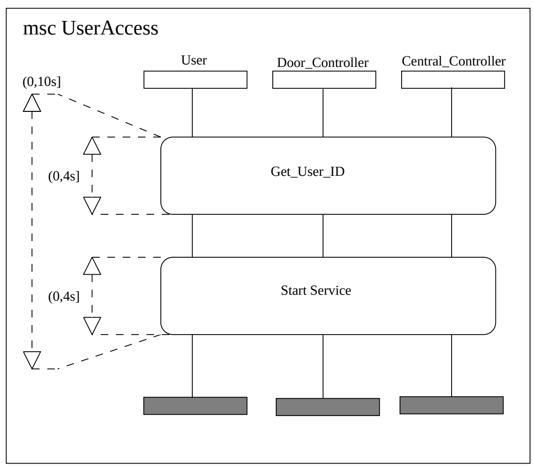


Figure 2.13: Example of a time interval.

Chapter 3

Formalization

3.1 Introduction

In this chapter a formalization based on sequences is presented.The features included in this formalization are:

- The basic Message Sequence Chart (MSC).
- The inline expression.
- The high level MSC (HMSC).
- Data

We do not formally model timing.

3.2 Previous work

There are many works related to the formalization of the MSC. For basic MSC, there are formalizations based on process algebra [3] , Petri nets [49] and an approach based on automata theory and temporal logic [7]. The most extensive semantics is based on process algebra.

In the formalization based on process algebra, it is very difficult to express what a condition is, the reason is, that a condition refers rather to states not to events. Conditions are treated as meaningless actions [3]. Petri nets are state oriented and this allows for a natural definition of conditions. Another advantage is that Petri nets provide different semantics for parallel composition and for alternative composition. The problem with this semantics is that there are no composition operators allowing to compose different MSC. Therefore, one have to specify a MSC as a closed system. Another interesting semantics is given by the automata approach

[7]. There is another semantics based on multiset algebra [46], this semantics includes the inline expressions.

In general, the existing semantics do not formalize features like actions, conditions, etc. Additionally, none of them includes the formalization of data.

3.3 Inconsistencies in the recommendation

We found one inconsistency in the recommendation Z.120 [5]: the mechanism used to modify the dynamic variables is the binding. A binding can occur in the message parameter list as is established. However, how must this binding be declared in the data declaration ? e.g., having the next message "*open(x:=4,10)*" contains one binding. The number of data parameters is two, but how should this message be declared ?

We assume that no bindings can appear in the message parameter list. And we do not formalize time.

3.4 Formalization of the basic MSC

The approach followed to formalize the MSC is based on a non-visual interpretation, as it is established in the recommendation Z.120. [5].

3.4.1 Structural elements

An MSC is formed from the following components:

- **Instances:** A finite set $\mathcal{I}$ of *instances.*The environment is modeled as another instance.
- **Timers:** A finite set $\mathcal{T}$ of *timers.*
- **Messages:** A set $\mathcal{M}$ of *messages.* A *message* may have data, i.e. as data parameters.
- **Expressions:** A set Exp of *data expressions* defined by an external data language.

3.4.2 Behavioral elements

The next elements define the behavior of the MSC

- **Events :** A finite set $\mathcal{E}$ of events.

- **Inline expressions :** A finite set $\mathcal{IE}$ of inline expressions. The inline expression is a multi instance meta-event[1].

- **Send - receive bijection :** The relation described by the interpretation associated to the messages (sending - receiving events) is described by a bijective function b such that each sending event is mapped to a unique receiving event, $b : \mathcal{S} \rightarrow \mathcal{R}$, where $\mathcal{S}$ and $\mathcal{R}$ denote respectively the set of sending and receiving events. In order to handle the instance creation, we define a Creating - Created bijection, that is similar to the Send - Receive Bijection.

- **Coregions:** A finite set $\mathcal{C}$. A coregion is a multiset of events.[2]

- **Sequence of elements** Each instance is mapping to a sequence of elements, these elements can be coregions, inline expressions or both. The total order described by the sequence is denoted by $\prec_T$. The set composed by all possible sequences of elements is denoted by $\mathbb{SEQ}(\mathcal{C} \cup \mathcal{IE})$. Each instance owns a sequence of elements (coregions, inline expressions or both) describing the elements in the instance's axis time, i.e. each instance is mapped to a sequence of coregions by the function m. $m : \mathcal{I} \rightarrow \mathbb{SEQ}(\mathcal{C} \cup \mathcal{IE})$.

3.4.3 The partial order

The combination of the previous elements presented define a partial order of events:

Partial Order [3]: The local total order and the send-receive bijection, define the relation $<$, known as partial order of the elements (coregions, inline expressions or both). Let c_1 and c_2 two different elements (coregion or inline expression), we say that c_1 precedes causally c_2 , denoted by $c_1 < c_2$, if

- c_1 and c_2 belong to the same instance, and c_1 precedes in time c_2, $c_1 \prec_T c_2$.

- c_1 is the sending event of the message m, and c_2 is the respective receiving event. $b(c_1) = c_2 \wedge b^{-1}(c_2) = c_1$.

3.5 Formalization of the Inline Expressions

An inline expression can be interpreted as a multi instance meta-event. The inline expression is a sequence of MSCs. Every MSC inside the inline expression is called

[1] The inline expression can not be consider as an event, but its position in the sequence can be interpreted as a meta-event.

[2] A multiset is a set-like object in which order is ignored, but multiplicity is explicitly significant. Therefore, multisets $\{1, 2, 3\}$ and $\{2, 1, 3\}$ are equivalent, but $\{1, 1, 2, 3\}$ and $\{1, 2, 3\}$ differ.

[3] A relation r is a partial order on a set S if it has: reflexivity, antisymmetry and transitivity.

inline section, this recursive property of the inline expression describes nested inline expressions. Every inline expression is labeled with a compositional operation such as { **par**, **alt**, **loop**, ... }.

3.6 Formalization the High Level MSC

A High level MSC, denoted by HMSC [4], is a directed graph composed by

- **Nodes:** A finite set $\mathcal{N}$ of Nodes. There are different types of nodes:
  - **Control nodes :** These nodes are used to represent the initial and terminal node of the graph. The initial node is represented using the symbol $\bigtriangledown$, and the terminal using the symbol $\triangle$.
  - **Conditions:** These nodes represent global conditions of the system.
  - **Parallel frames:** These nodes represent parallel composition of elements (MSC or HMSC).
  - **Reference:** These nodes represent references to basic MSC (or instances of MSC utilities) or another HMSC.
  - **Connection points:** These nodes are used to improve the readability of the HMSC, they have no semantic interpretation.
- **Labeling node function:** A labeling function l that maps each node reference to an MSC, a parallel frame or another HMSC.
- **Edges:** A set of edges that connect nodes to nodes. $E \subseteq [\mathcal{N} \times \mathcal{N}]$.
- **Labeling operation function:** A function o that maps each edge to any operation (alternative, parallel or sequential).

There are some restrictions in the number of nodes. The number of initial nodes in the HMSC must be one. The number of incoming arrows to an initial node must be zero, and the number of outgoing arrows from a final node must be zero.

3.7 Formalization of Data

3.7.1 Basic elements

In this section we assume that all the details related to the syntactic properties of the data language have been solved. We assume a semantic domain $\mathcal{S}$ in which

[4]We change the use of some elements in the HMSC in order to make a simple formalization

the expressions will be interpreted. We assume the existence of a multiset set of variables $\mathcal{V}$ and a labeling relation loc that maps variables to instances, $loc : \mathcal{V} \rightarrow \mathcal{I}$, describing the local variables. The relation in maps variables to instances, describing the inherited variables. The definition or undefinition of variables is denoted as a binding, in the case of definition the binding is performed using a non-deterministic choice operation and the undefinition operation is represented by the use of binding to the "undefined" value $\perp$. The set of expressions Exp contains the set of *binding expressions* (B), being the strings that represent (or can represent) respectively declarations or bindings to variables.

3.7.2 State

We need to define the notion of state. A state gives a snapshot of all variables involved in the MSC. A **state** consist of:

- A set of defined variables V, $V \subseteq \mathcal{V}$.
- A valuation function $\varphi : V \rightarrow \mathcal{S}$, giving the values of the variables. The set of all valuation functions is called Φ. The interpretation of the wildcard in the context of this valuation function is defined as a non-deterministic choice, meaning that assuming that the wildcard has a specific data type, then the non-deterministic choice select any value from the values described by the corresponding data type, we denote the non-deterministic choice as $x :\in A$, where x is the value taken from the set A.

Additional to the previous elements, we need a set of functions to interpret the various elements:

- For *bindings*: A set $B' \subseteq B$ for each set of variables V, giving the set of bindings that may actually be used, given that only variables in V are defined, and state transition function $\tau : \Phi \times B \rightarrow \Phi$. $\tau(\varphi, b)$ denotes the new event state the MSC turns into when binding b is executed in the event state (V, φ). Note that $\tau(\varphi, b)$ needs only be defined when $b \in B'$, where V is the set of variables on which φ is defined.
- For *expressions*: A set $Ex \subseteq Exp$ for each set of variables V, giving the set of expressions that may actually be used, given that only variables in V are defined and an interpretation function $I_\varphi : Ex \rightarrow \mathcal{S}$, where V is the set of all variables on which φ is defined. $I_\varphi(x)$ gives the value that x is interpreted to.
- For *local variables in any expression*: $l : Exp \times \mathcal{I} \rightarrow \mathcal{P}(V)$ giving the the local variables appearing in any expression in some instance.

- For *inherited variables in any expression*: $i : Exp \times \mathcal{I} \rightarrow \mathcal{P}(V)$ giving the inherited variables appearing in any expression in some instance.

3.7.3 General semantics

In an event state (V, φ) all expressions must be in Ex an all bindings in B'. Provided one uses variables with a well-defined scope, this can be checked statically. The types of events that can change the state are: The action, receiving, setting timer, timeout and instance creation event. The semantics associated to the modification of the event state is similar for all of them. If the MSC is in a state (V, φ), all events such as $action(a, i)$ or $receive(m, i, j)$ have in the "label"(i.e. a for action and m for a message) part expressions, which must be in Ex. The semantics for such events are equal to the semantics of $(i, \mathbf{receive}, I_\varphi(m), i, j)$ and $(i, \mathbf{action}, I_\varphi(a))$. However, if there is any explicit binding, such that $m \in B$ or $a \in B$, this causes that the event state changes from (V, φ) to $(V, \tau(\varphi, a))$ or $(V, \tau(\varphi, m))$.

Chapter 4

Execution Model for the basic MSC

4.1 Introduction

In this chapter the execution model for the basic MSC is presented. The approach followed to model the execution of the MSC was the utilization of an *Abstract Execution Machine* (AEM). This AEM can be used in two different ways: as an *Acceptor* or *Generator*. The AEM works as an *Acceptor* when is used to verify if a set of traces met the corresponding specification (MSCs). The AEM works as a *Generator* when the AEM is used to generate set of traces based on a specification.

4.2 Basic concepts

4.2.1 Event Structure

The Event Structure $\mathcal{E}_{\mathcal{S}}$ is a vector of sequences. The elements of the sequences are coregions and inline expressions. This structure represents the set of sequences of coregions as is presented in Figure 4.14.

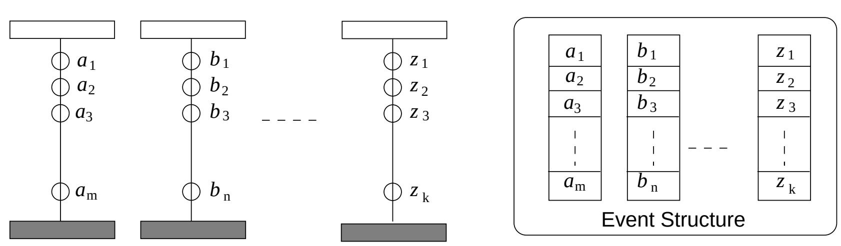


Figure 4.14: Visualization of the Event Structure.

4.2.2 Instance Reference

An *instance reference* is a structure composed by a pointer and a multiset of events. The *instance reference* is used as a pointer over the sequence of elements in the Event Structure (MSC). Every instance reference is associated to any instance. The instance reference works as a head over the sequence and reads the content of the current element in the sequence and keep it.

4.3 Abstract Execution Machine

The *Abstract Execution Machine* is a set of structures and rules used to generate or accept traces defined by an MSC. The AEM is composed by the next components:

- **Control reference:** A set $Cref$ of *instance references*, this set is similar to the set of heads used in a multi-tape Turing Machine.
- **Event memory:** A set $\mathcal{M}_{\mathcal{E}}$ of events. This set is used to keep some historic knowledge of the execution.
- **Data space:** A set D_{space} of *MSC variables.* This component will be explained in the next sections.
- **Operational rules:** A set O of operational rules that define the AEM behavior.

A graphical representation of the AEM and the event structure is presented in Figure 4.15.

4.3.1 Event Memory

An Event Memory, $\mathcal{M}_{\mathcal{E}}$ is a multiset of events. The Event Memory is used to record some historical information about the execution of the AEM.

4.3.2 The *enabled* predicate

The next table presents the description of the *enabled* predicate.

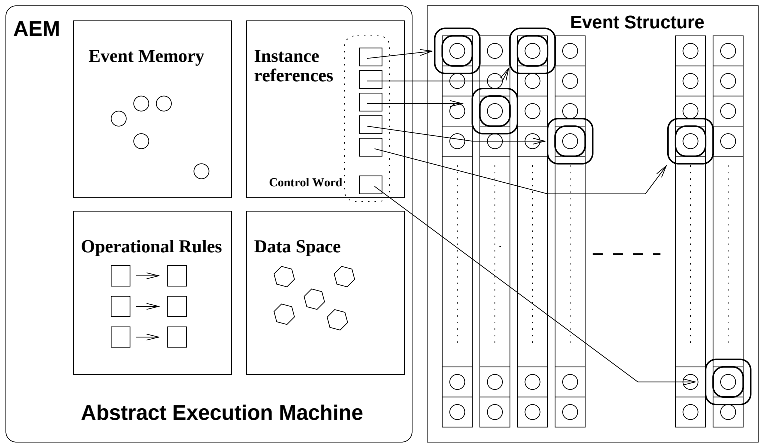


Figure 4.15: The AEM and the event structure.

Event type	**Condition**	**Description**
Receiving	$b^{-1}(ev) \in \mathcal{M}_{\mathcal{E}}$	The corresponding sending event is located in the Event Memory. i.e., the corresponding sending event has occurred.
Time out	$time(Timer(ev))$	The timer counter has reached the timeout value. The *time* predicate is associated to the timed semantics of the MSCs.
Created instance	$b^{-1}(ev) \in \mathcal{M}_{\mathcal{E}}$	The corresponding creating instance event is located in the Event Memory
Creating instance	$true$	
Sending	$true$	
Stopping Instance	$true$	
Condition	$true$	
Action	$true$	
Setting timer	$true$	
Stopping timer	$true$	

Table 4.1: Description of the *enabled* predicate .

4.3.3 The *action* operation

The *action* operation defines the actions that are performed when an event occurs:

Event	**Actions**	**Description**
Sending	$\mathcal{M}_{\mathcal{E}} \cup \{generate(ev)\}$	Copy the event in the Event Memory.
Receiving	$\mathcal{M}_{\mathcal{E}} - \{b^{-1}(ev)\}$ and $update(ev)$	Remove the event from the Event Memory and update the set of variables
Action	$update(generate(ev))$	Update the local set of variables
Setting timer	$update(generate(ev))$ and $start(ev)$	Start the timer and update the set of variables
Stopping timer	$stop(ev)$	Stop the timer
Timeout	$update(generate(ev))$	Update the local bindings
Creating instance	$\mathcal{M}_{\mathcal{E}} \cup \{generate(ev)\}$	Copy the event in the Event Memory.
Created instance	$\mathcal{M}_{\mathcal{E}} - \{b^{-1}(ev)\}$ and $Cref \cup \{ref(ev)\}$ and $update(ev)$	Remove the corresponding creating event from the Event Memory and update the set of variables
Stopping instance	$Cref - \{(ref(ev))\}$	Remove the event reference [1]
Condition	if $evaluate(ev)$ = **false** then $stopExec(ev)$.	If the condition evaluation is false, then stop the the execution[2].

Table 4.2: Description of the *action* operation.

4.3.4 Operational rules

Let be ev be any event in any instance reference (meaning in the multiset of events) and let be ref be any reference in the control word, then

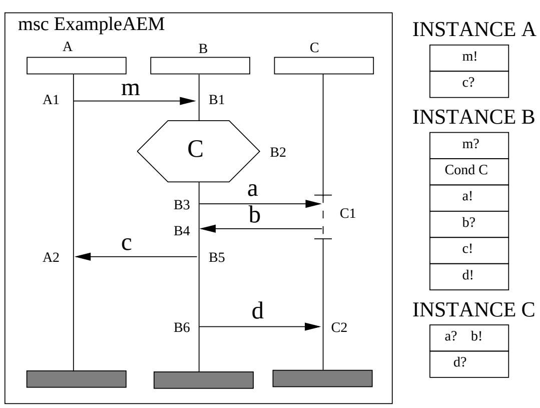
Rule	**Rule**	**Description**
Enabling event	$enabled(ev) \longrightarrow action(ev)$	If an event is enabled, then perform the corresponding action.
Computing progress	$ref = \emptyset \longrightarrow next(ref)$	If there is any event reference that is empty, then move to the next element in the corresponding sequence.

Table 4.3: The AEM operational rules.

4.4 Example: The operation of the AEM without data

Figure 4.16 presents an example of execution using the AEM. The header of the table has the following columns:

- A, B and C: denote the *instance reference*in the MSC.
- R(A), R(B) and R(C): Denote the elements in the *instance references*.
- Enabled : denotes the set of enabled events.
- Selected: denotes the event which actually occurs.



	A	B	C	R(A)	R(B)	R(C)	Enabled	Selected
0	A1	B1	C1	m!	m?	a? b!	m! b!	m!
1	A2	B1	C1	c?	m?	a? b!	m? b!	m?
2	A2	B2	C1	c?	C	a? b!	C b!	b!
3	A2	B2	C1	c?	C	a?	C	C
4	A2	B3	C1	c?	a!	a?	a!	a!
5	A2	B4	C1	c?	b?	a?	a?	a?
6	A2	B4	C2	c?	b?	d?	b?	b?
7	A2	B5	C2	c?	c!	d?	c!	c!
8	A2	B6	C2	c?	d!	d?	c? d!	c?
9		B6	C2		d!	d?	d!	d!
10			C2			d?	d?	d?
11								

Figure 4.16: MSC Example.

4.4.1 Data space

The *data space* is a set of tuples $< v, i, j, s >$ called *MSC variables*, where v is the variable, i is the owner instance, j is the instance which manipulate the variable and s is the state of the variable (defined or undefined). The two instances are used to distinguish between a local or inherited variable,i.e the owning and the manipulating instance.

4.4.2 The *generate/accept* function

If the approach selected is as an Acceptor then the *accept* function fill the wildcards with the values provided in the event. If the approach selected is as a Generator, the function will select any random (non-deterministic choice) value from the corresponding data type.

4.4.3 Snapshot

A Snapshot is a set of *MSC variables*. A snapshot is used to copy the variables (bindings) that are explicitly or implicitly referenced in the parameter expressions of some events. This snapshot is associated to the sending and creating event when is located in the Event Memory. The other events do not require temporal storage due to the fact that the snapshot is not required when the data modification is performed.

4.4.4 The *update* action

The *update*(ev) action performs the actions related to the updating of the data space.

1. Extract the explicit bindings (local variables).

2. Update all local local bindings.

3. If the event is a creating event, then update the creating and the created event state.

4. If the event is a sending event then create a snapshot and store it in the event memory.

5. Add or update the inherited variables referenced in the parameter expression.

In this section an example of execution using an MSC with data is presented. The headers of tables 4.4 and 4.5 have the following columns:

- A, B and C: denote *instance references*.
- R(A), R(B) and R(C): denote the elements in the *instance reference*.
- Enabled : denotes the set of enabled events.
- Selected: denotes the "real" event.
- Loc(A), Loc(B) and Loc(C) denotes the local variables of each instance.
- In(A), In(B) and In(C) denotes the inherited variables of each instance.

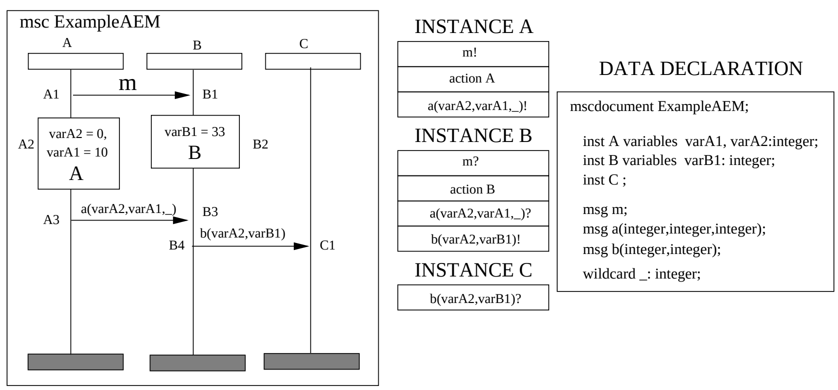


Figure 4.17: MSC Example with data.

Step	A	B	C	R(A)	R(B)	R(C)	Enabled	Selected
0	A1	B1	C1	m!	m?	b?	m!	m!
1	A2	B1	C1	A	m?	b?	m?,A	m?
2	A2	B2	C1	A	B	b?	B,A	A
3	A3	B2	C1	a!	B	b?	B, a!	a!
4		B2	C1		B	b?	B	B
5		B3	C1		a?	b?	a?	a?
6		B4	C1		b!	b?	b!	b!
7			C1			b?	b?	b?
8								

Table 4.4: Execution of the bAEM using the MSC with data.

T	Loc(A)	Loc(B)	Loc(C)	In(A)	In(B)	In(C)
0	varA1=⊥, varA2=⊥	varB1=⊥				
1	varA1=⊥, varA2=⊥	varB1=⊥				
2	varA1=10, varA2=0	varB1=⊥				
3	varA1=10, varA2=0	varB1=⊥				
4	varA1=10, varA2=0	varB1 = 33				
5	varA1=10, varA2=0	varB1 = 33			varA1=10, varA2=0	
6	varA1=10, varA2=0	varB1 = 33			varA1=10, varA2=0	
7	varA1=10, varA2=0	varB1=33			varA1=10, varA2=0	varA2=0, varB1=33
8	varA1=10, varA2=0	varB1=33			varA1=10, varA2=0	varA2=0, varB1=33

Table 4.5: Execution of the bAEM using the MSC with data, this table only describes the event states.

Chapter 5

Execution model for the basic MSC with inline expressions

5.1 Introduction

Using the formalization presented in Chapter 3, the inline expression is another element in the sequence that defines the instance behavior. The inline expression is composed by sections, and every section is a sequence of elements. Every section is interpreted as a new *instance reference*(thread) in the AEM. However, we need to include more information in order to describe the semantic behavior of every inline expression type. The Figure 5.18 shows a representation of this approach.

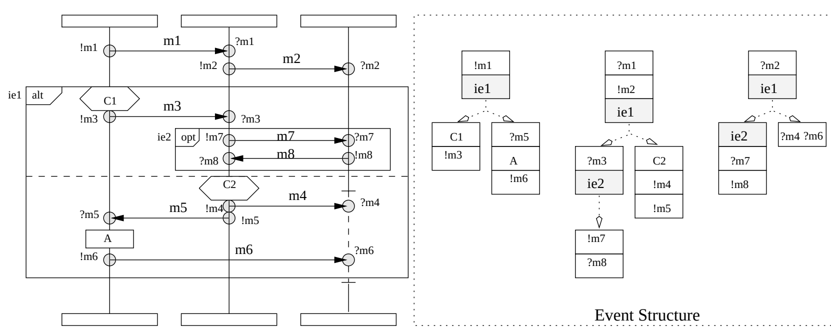


Figure 5.18: The interpretation of the inline expression in the AEM.

Figure 5.18 shows an MSC containing two inline expressions. The corresponding send and receive events of each message are denoted by circles. Next to the MSC the sequence intepretation is presented. The first instance contains just two events, the sending event (!m1) and the inline expression (ie1). The inline expression ie1 contains two sections, and each section contains two and three evens respectivelly. Using this approach nested inline expression can be described.

5.2 The extended *instance reference*

5.2.1 The *instance reference*state

The *instance reference*may be in one of the following states (Figure 5.19):

- **Sleeping.** The *instance reference*is created, but it can not run.
- **Running.** The *instance reference*is being executed.
- **Waiting.** The *instance reference*is waiting to be awake.
- **Terminated.** The *instance reference*has finished execution.

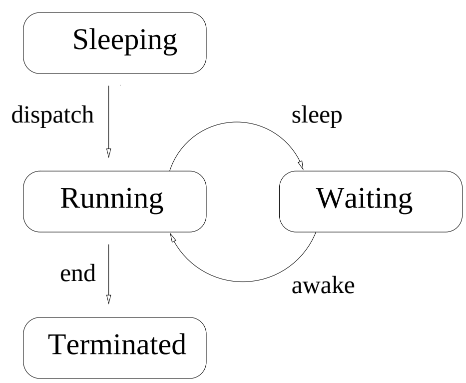


Figure 5.19: The *instance reference*states.

5.2.2 The *instance reference*counter

In order to handle the loop inline expression all *instance references*have an counter, i.e. an integer variable associate to them.

5.2.3 Decision set

The common behavior (Figure 5.20) in the alternative inline expressions is handled by the AEM using a decision set. The decision set is composed by *instance references*, this set denotes disjoint alternatives among *instance references*. The set of all decision sets is called *Decision Memory*.

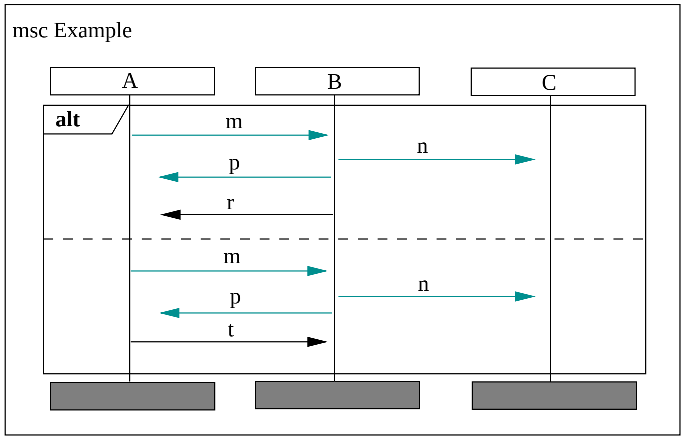


Figure 5.20: Example of common preamble between two alternatives in the MSC.

5.2.4 The inline expression activation

The inline expression activation is similar to the procedures used to create new threads in the programming languages. The Figure 5.21 presents the approach used to represent the different "threads" (*instance references*) that are activated in the inline expressions.

This activation is denoted by the operation *activate* and is described by the next steps:

If the *instance reference*is the first referenced instance that reaches the inline expression:

1. For every section in every instance in the inline expression a new *instance reference*is created, the initial state of these *instance reference*is **Ready**.
2. The new *instance reference*created that has the same instance as the current *instance reference*is dispatched.
3. The current instance reference state changes to **waiting**.
4. Depending on the inline expression label, the next set of actions is performed:

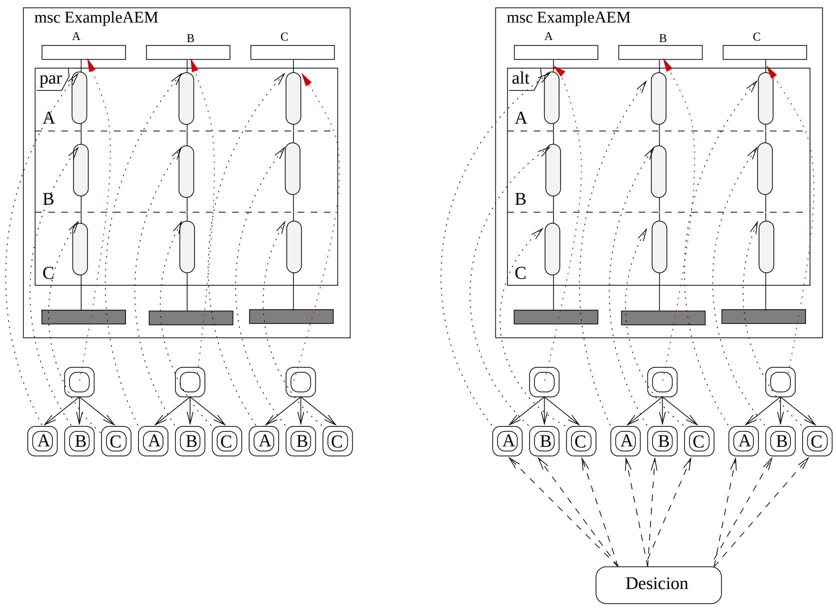


Figure 5.21: The inline expression activation in the AEM.

Label	**Actions**
Alternative	Add a new decision set using them.
Parallel	
Loop	
Optional	Add a new decision set using.
Exception	Add a new decision set using them and the current *instance reference*.

Otherwise:

1. The *instance references*that already exists (children) are dispatched.
2. The current instance reference state changes to **waiting**.
3. Depending on the inline expression label, the next set of actions is performed:

Label	**Actions**
Alternative	Add a new decision set using them.
Parallel	
Loop	
Optional	Add a new decision set using them and the current *instance reference*.
Exception	Add a new decision set using them and the current *instance reference*.

5.2.5 The *instance reference* relationships

To organize the set of *instance references* we define a set of additional concepts that are interpreted as relations.

- A set of *instance references* are **brothers** if they are at the same level. For example, the initial *instance reference* in any MSC are brothers. The set of new *instance references* that are created when the AEM find an inline expression are brothers among them.
- A set of *instance references* are **children** if they have the same "father", meaning the same *instance reference* that dispatches them.

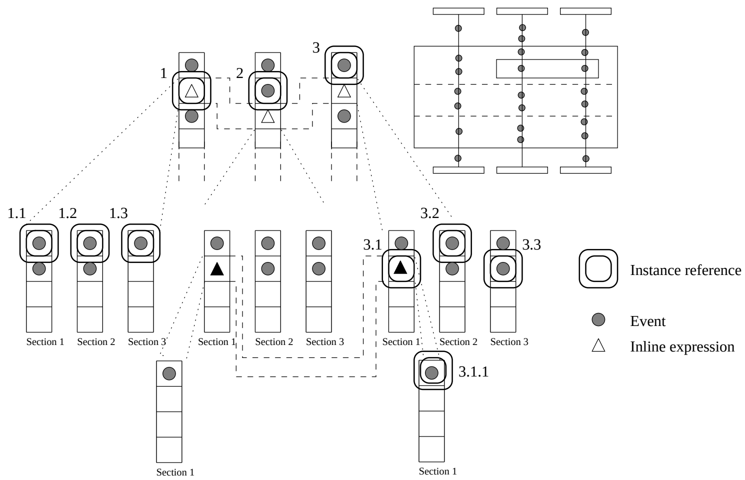
Figure 5.22 shows an example of this concepts.

5.2.6 Inline expression counter set

In order to control the execution of the loop, we define a tuple called *inline expression counter*, $< ie, v, s >$, where ie denotes inline expression, v the value of the maximum counter and s denotes the state { free, locked }. The set of inline expression counters is called *inline expression counter set.*

5.2.7 The operation *clean*

This operation removes elements from the Decision set and terminates *instance references*. This operation is used to allow the execution of *instance references* when they are selected by the event (in the case of alternative inline expression for example). Let be ev any event that occurs:



The instance references 1,2,3 are brothers.
The instance references 1.1, 1.2, 1.3 are brothers.
The instance references 3.1, 3.2, 3.3 are brothers.

The instance reference 1 is the father of 1.1, 1.2 and 1.3 .
The instance reference 3 is the father of 3.1, 3.2 and 3.3 .
The instance reference 3.1 is the father of 3.1.1 .

Figure 5.22: The relations among *instance references*.

1. Get all *instance references*that own this event.
2. Get all decision sets that have any of the *instance references*computed in step 1.
3. Every *instance reference*and its corresponding brothers that are in any set found in step 2 and do not own the event *ev* must be terminated. (This step removes the unselected alternatives).
4. Update the corresponding decision set. If there are any set containing only one element, then this set must be removed, otherwise just remove the corresponding decision relation.
5. If there is any loop involved in the decision set, then lock the corresponding inline expression counter.

5.2.8 The operation *stopExec*

This operation stops (removes) the *instance references*that are related to any event in a condition. It means that the *instance reference*related with a guard is stopped.

This operation terminates all related *instance reference*to any condition that evaluates to false (The brothers are terminated).

5.2.9 The operation *evalLoop*

This operation perform the following actions: Let be *top* the maximum value associated to the corresponding inline expression, *max* and *min* the corresponding loop bounds and c the *instance reference*counter.

1. Increment the *instance reference*counter.
2. If the operation is restricted, $(c < min \wedge c < top)$ then activate(ev).
3. If the operation is not restricted, $(c > min \wedge c < max \wedge topisnotlock) \vee$ then create a new friend *instance reference*and add a new decision set.

5.2.10 The extended *action* operation

The *action* operation defines the actions that are performed when an event happens, or is selected.

5.2.11 The extended operational rules

Let *ev* be any event in any instance reference (meaning in the multiset of events) and let *ref* be any reference in the control word, then

5.3 Example: The operation of the AEM with inline expressions

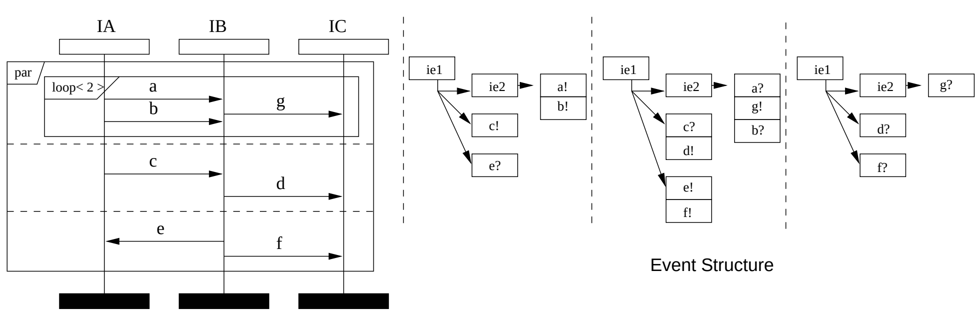


Figure 5.23: Example MSC with inline expressions and the corresponding Event Structure .

Event	**Actions**	**Description**
Sending	$\mathcal{M}_{\mathcal{E}} \cup \{generate(ev)\}$	Take a copy of the associated bindings in the sending instance and store it in the Event Memory
Receiving	$\mathcal{M}_{\mathcal{E}} - \{b^{-1}(ev)\}$ and $update(ev)$	Remove the event from the Event Memory and update the set of variables
Action	$update(generate(ev))$	Update the local set of variables
Setting timer	$update(generate(ev))$ and $start(ev)$	Start the timer and update the set of variables
Stopping timer	$stop(ev)$	Stop the timer
Timeout	$update(generate(ev))$	Update the local bindings
Creating instance	$\mathcal{M}_{\mathcal{E}} \cup \{generate(ev)\}$	Take a copy of the associated bindings in the sending instance and store it in the Event Memory
Created instance	$\mathcal{M}_{\mathcal{E}} - \{b^{-1}(ev)\}$ and $Cref \cup \{ref(ev)\}$ and $update(ev)$	Remove the corresponding creating event from the Event Memory and update the set of variables
Stopping instance	$Cref - \{(ref(ev))\}$	Remove the event reference [1]
Condition	if $evaluate(ev) = $ **false** then $stopExec(ev)$.	If the condition evaluation is false, then stop the the execution[2].
Inline Expression	$activate(ev)$	Activate the inline expression

Table 5.6: Description of the extended *action* operation.

Rule	**Description**
Event Execution	If an event is enabled, then perform the operations *action* and *clean*.
Computing progress	If there is any *instance reference* that is empty and running, then move to the next element in the corresponding sequence.
Awakening *instance references*	If there is any *instance reference* that has no children and the corresponding inline expression is not a loop then the *instance reference* is awake If there is any *instance reference* that has no children and the corresponding inline expression is a loop then perform the operation *evalLoop*
Terminating *instance reference*	If the sequence has no elements then the *instance reference* is terminated.

Table 5.7: The AEM operational rules.

Figure 5.24 presents the corresponding *instance references* and its execution.

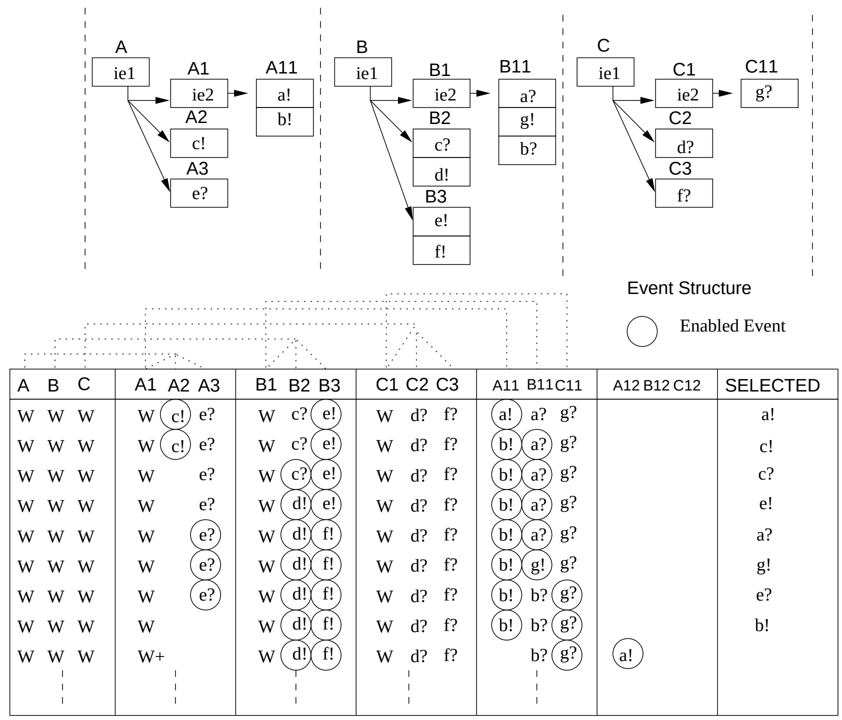


Figure 5.24: Execution example.

Chapter 6

Execution Model of the HMSC

6.1 Introduction

In this chapter the execution of the High Level Message Sequence Chart (HMSC) is presented. The execution model is based on an extension of the previous Abstract Execution Machine. The approach is similar to the one used to handle the inline expressions.

6.2 Approach

The approach used to handle the HMSC is assuming a **strong** vertical composition. We need to use this approach since the weak vertical composition can lead to undesired results. An example of a HMSC is presented in Figure 6.25. The instances involved in each node are presented. Assuming the weak vertical composition, there is not an initial scenario, the initial scenario is split in many "initial scenarios". The example shows the "real" initial set of scenarios (MSCs A, B and E, the reason is based on the meaning of the weak vertical composition.

This situation can be handled, in some way, using the strong vertical composition, where all events in the first MSC precede the events in the second MSC. Assuming this composition the execution is isolated in just one node (with exception of the alternative and parallel operations).

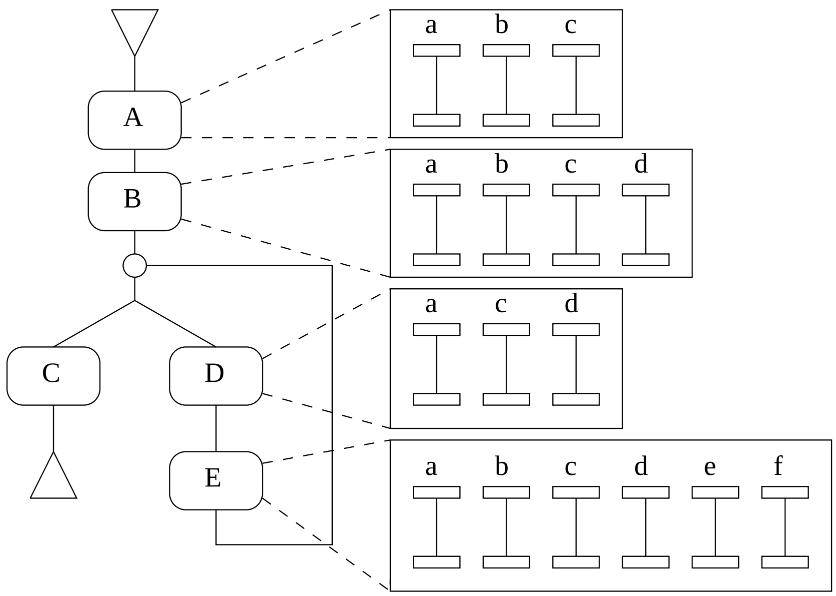


Figure 6.25: A HMSC and the instances presented in some nodes.

6.3 The new elements in the AEM

6.3.1 The *node reference*

A *node reference*is similar to an *instance reference*. The set of possible states is the same. The only difference is the object that is referenced: the *instance reference*points to elements in any sequence and the *node reference*points to nodes in the digraph.

The relations that the *node reference*can have are: fatherhood and childhood. The fatherhood relation is extended to either *node reference*or *instance reference*. The only one constraint is that an *instance reference*cannot be father of a *node reference*.

6.3.2 The extended *clean* operation

This operation removes elements from the Decision set and terminates *instance reference*. This operation is used to allow the execution of *instance references*when they are selected by the event (in the case of alternative inline expression for example). Let *ev* be any event that really happens:

1. Get all *instance references*that own this event.

2. Get all decision tuples that have any of the *instance references*computed in

step 1.

3. Every *instance reference*and its corresponding brothers that are in any tuple found in step 2 and do not own the event *ev* must be terminated. (This step removes the unselected alternatives).

4. Update the corresponding decision set. If there is any tuple containing only one element, then this tuple must be removed, otherwise just remove the corresponding decision relation.

5. If there is any loop involved in the decision set, then *lock* the corresponding inline expression counter. Meaning that the *instance references*related in the inline expression must perform this number of loops. In some sense, this counter denotes the compromised iterations.

6.3.3 The extended operation rules

Let *ev* be any event in any instance reference (meaning in the multiset of events) and let *ref* be any reference in the control word:

Rule	**Description**
Event Execution	If an event is enabled, then perform the operations *action* and *clean*.
Computing progress	If there is any *instance reference*that is empty and running, then move to the next element in the corresponding sequence.
Awakening *instance references*	If there is any *instance reference*that has no children and the corresponding inline expression is not a loop then the *instance reference*is awake If there is any *instance reference*that has no children and the corresponding inline expression is a loop then perform the operation *evalLoop*
Terminating *instance reference*	If the sequence has no elements then the *instance reference*is terminated.
Executing *node reference*	If there is no child *instance references*then move to the next node in the digraph and create all the corresponding children.

Table 6.8: The AEM operation rules.

6.4 The operation of the AEM with HMSC

Figure 6.26 and 6.27 present how the *node reference*and *instance reference*are visualized in the execution of the HMSC. Figure 6.26 presents the case where a node is reached and the *node reference*activates the *instance reference*corresponding to the instances referenced in the MSCs node. Figure 6.27 presents the case where an alternative is reached. The way to handle the decision is similar to the one used to handle alternative inline expressions.

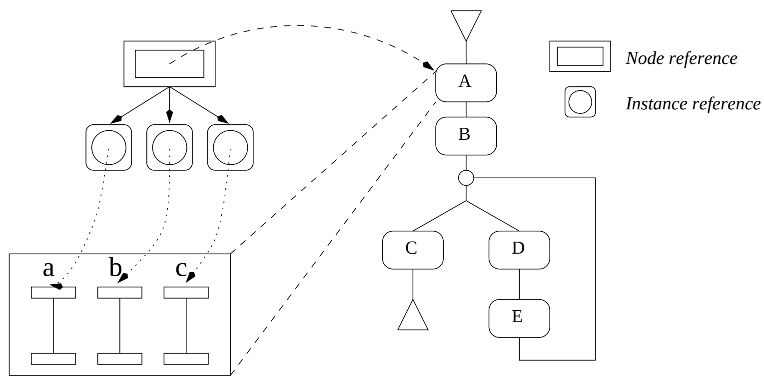


Figure 6.26: Example of the AEM and HMSC (initial step).

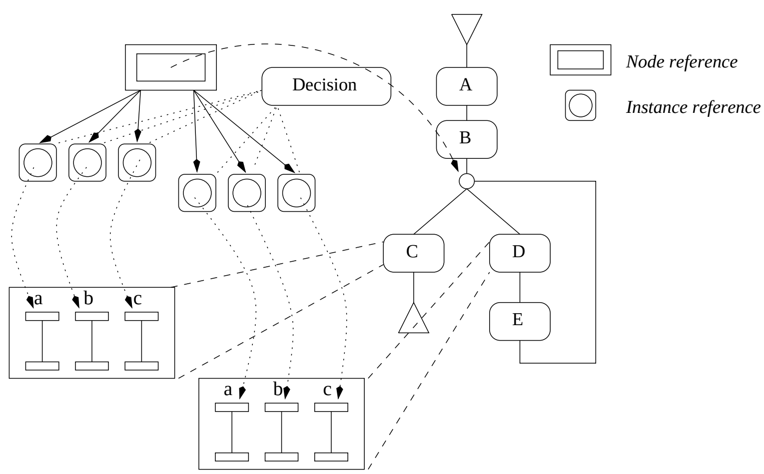


Figure 6.27: Example of the AEM and HMSC (alternative)

Chapter 7

Applications

7.1 Introduction

Testing is one of the most popular methods to protect users and customers against insecure, inappropriate or even erroneous soft- and hardware products. Furthermore, a thorough and comprehensive test gives an indication about the quality of a product.

One application of the AEM is the test generation. This chapter presents an initial approach towards the test generation using MSC2000. The goal is the generation of TTCN test cases.

7.2 Previous work

The idea of using the MSC as source for the automatic test generation is not new. There is one paper describing the direct generation of test cases from MSCs [62]. The MSCs version that was utilized in this project was the MSC'92, neither inline expression nor HMSC were included.

The approach followed in this project was based on the analysis of all possible traces that can be generated by the MSCs.The analysis starts removing the internal events (events that are inside the system under test) following by a reduction of traces that are "equivalent". The algorithm proposed worked effectively due the fact that the MSCs presented can only generate a finite number of traces. However the new features in the MSC2000 need to be handled carefully.

The algorithm proposed by [62] is explained as follows:

1. An MSC describes a partial ordered set of actions. The partial order is defined by the messages and by the order of actions along the instance axes. Based on this information we calculate the sequences of actions which include the

actions of the MSC and which are consistent with the partial order defined by the MSC.

2. For the test case description only the actions of the testers are of interest. Therefore in the second step we remove all actions which are not performed by the testers from each sequence.

3. MSC and TTCN are different languages with different semantics. For TTCN some of the sequences which we generated in step 2 are redundant. During a test run they can not be distinguished. In other words, for TTCN several sequences are in the same equivalence class. In the third step we select one sequence of each equivalence class.

4. In the fourth step the selected sequences are transformed into the TTCN notation.

7.3 The TTCN language

A TTCN description specifies a whole test suite. It consists of

- a *test suite* overview which is mainly a contents list of the test suite,
- a *declarations part* which includes the message and data type definitions,
- a *constraint part* which consists of conditions on message parameters, i.e. default values or value ranges which should be tested, and
- a *dynamic part* which for each test case describes the sequence of exchanged messages.

TTCN has two syntactical forms: TTCN/MP (TTCN Machine Processible form) as pure textual representation and TTCN/GR (TTCN Graphical form) which is a graphical representation. Both forms are equivalent and can be translated into each other. We only consider the dynamic part, focus on the generation of the tree structure.

7.3.1 The declarations and constraints part

TTCN has its own data type and value assignment concept. It includes very powerful operators to express conditions on parameter values. For practical purposes TTCN also allows to use ASN.1 in the declarations and constraints part.

7.3.2 The dynamic part

A TTCN test case describes the sequences of events which should be performed by the testers. In general, these are send and receive events. The event sequence is specified by means of a tree notation. Figure 7.27 shows an example with the corresponding MSC (Only the tree is presented).

The tree structure is determined by the ordering and the indentation of the specified events. In general, the same indentation denotes a branching (i.e. alternative events, e.g. lines No. 2 and 4) and the next larger indentation denotes a succeeding event (e.g. lines No. 1 and 2). Events are characterized by the involved entities (i.e. A and B), by its kind (i.e. "!" denotes a send event and "?" describes a receive event) and by the message which should be sent or received. An example may clarify the notation. The statement B?Disconnect denotes the reception of the message Disconnect by the entity B.

1	A!ReleaseComplete	
2	B?Disconnect	
3	B!ReleaseComplete	
4	B?ReleaseComplete	

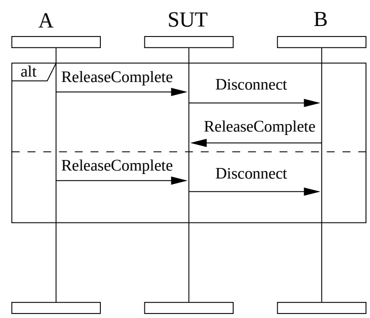


Figure 7.28: A TTCN tree and its corresponding MSC.

Chapter 8

Concluding remarks

In this thesis we have given an introduction of the new features presented in MSC2000. The most important features of the MSC2000 have been explained in Chapter 1 and 2. This chapter also contains an informal explanation of the meaning of the MSC.

In Chapter 4, a formal description of the MSC is introduced, based on sequences and bijective functions. In this chapter oner inconsistency in the new Recommendation is presented.

In Chapter 5, the execution model for the basic MSC is presented. The model was described using an*Abstract Execution Machine*. This model includes the data concepts proposed in [5] with some restrictions. Two examples are presented.

In Chapter 6, the AEM is extended to handle inline expressions. Chapter 7 extends the AEM to handle HMSC, handling strong vertical composition instead of the weak vertical composition.

There are two possible ways to use the AEM, as an Acceptor or Generator. In Chapter 8 a discussion of the applications of the AEM is presented.

The AEM constitutes an executable interpretation of the MSC, allowing the implementation of MSC based tools.

References

[1] R. Alur and M. Yannakakis. Model checking of message sequence charts, Proceedings of the Tenth International Conference on Concurrency Theory, Springer Verlag, 1999

[2] Damm, W, and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems, (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, pp. 293-312, 1999.

[3] S. Mauw and M.A. Reniers. Operational semantics for MSC'96. In A. Cavalli and D. Vincent, editors, SDL'97: Time for Testing - SDL, MSC and Trends, Tutorials of the Eighth SDL Forum, pages 135-152, Evry, France, September 1997.

[4] Ekkart Rudolph, Peter Graubmann, Jens Grabowski. Tutorial on Message Sequence Charts. In: Tutorials of the 7th SDL Forum, Oslo, Norway, September 25-29, 1995.

[5] Message Sequence Charts. (MSC2000). ITU-T, 1999.

[6] Padilla, Gerardo. Understanding MSC2000 Data and Time Concepts. Project Report. Jul 2000.

[7] Ladkin, P.B., Leue, S. What Do Message Sequence Charts Mean? In R.L. Tenney, P.D. Amer, M.U. Uyar (eds.), Formal Description Techniques VI, IFIP Transactions C, Proceedings of the 6th International Conference on Formal Description Techniques, North-Holland, p. 301 - 316, 1994.

[8] Grabowski, J., An Automata based Model for the Implementation of a TTCN Simulator. Technical Report IAM-96-006, University of Berne, Institute for Informatics, Berne, Switzerland, February 1996.

[9] S. Leue, L. Mehrmann and M. Rezai, Synthesizing ROOM Models from Message

Sequence Chart Specifications, 13th IEEE Conference on Automated Software Engineering, Honolulu, Hawaii, October 1998. Based on Technical Report 98-06.

[10] A.G. Engels, L.M.G. Feijs, and S. Mauw. MSC and data: dynamic variables. In R. Dssouli, G.v. Bochmann,and Y. Lahav, editors, SDL'99, Proceedings of the Ninth SDL Forum, pages 105-120, Montreal, June 21-25 1999. Elsevier Science Publishers B.V.

[11] G.J. Holzmann, Early Fault Detection Tools, Software Concepts and Tools, Vol. 17, 2, pp. 63-69, 1996, (also: Proc. TACAS95 Passau Germany LNCS 1055 pp. 1-13) .

[12] G.J. Holzmann, Formal Methods for Early Fault Detection, Proc. FTRTFT Confs. on Formal Techniques for Real-Time and Fault Tolerant Systems, LNCS, Vol. 1135, pp. 40–54, Uppsala Sweden, September 1996, ((invited paper)) .

[13] G.J. Holzmann D.A. Peled and M.H. Redberg, Design Tools for Requirements Engineering, Bell Labs Technical Journal, pp. 86-95, Winter 1997

[14] R. Alur G.J. Holzmann and D. Peled, An Analyzer for Message Sequence Charts, Software Concepts and Tools, Vol. 17, 2, pp. 70-77, 1996, (also: Proc. TACAS95 Passau Germany LNCS 1055 pp. 35–48) .

[15] ITU-TS. ITU-TS Recommendation Z.120 Annex B: Algebraic semantics of Message Sequence Charts. ITU-TS, Geneva, 1995.

[16] S. Mauw and M.A. Reniers. An algebraic semantics of Basic Message Sequence Charts. The computer journal, 37(4):269-277, 1994.

[17] S. Mauw. The formalization of Message Sequence Charts. Computer Networks and ISDN Systems, 28(12):1643-1657, 1996.

[18] S. Mauw and M.A. Reniers. Operational semantics for MSC'96. Computer Networks and ISDN Systems, 31(17):1785-1799, 1999.

[19] J.C.M. Baeten and S. Mauw. Delayed choice: an operator for joining Message Sequence Charts. In D. Hogrefe and S. Leue, editors, Formal Description Techniques, VII, pages 340-354. Chapman & Hall, 1995.

[20] S. Mauw and E.A. van der Meulen. Specification of tools for Message Sequence Charts. In ASF+SDF '95. Amsterdam, 1995.

[21] S. Mauw and M.A. Reniers. High-level Message Sequence Charts. In A. Cavalli and A. Sarma, editors, SDL'97: Time for Testing - SDL, MSC and Trends, Proceedings of the Eighth SDL Forum, pages 291-306, Evry, France, September 1997.

[22] A. Engels, S. Mauw, and M.A. Reniers. A hierarchy of communication models for Message Sequence Charts. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, FORTE/PSTV'97, pages 75-90, Osaka, Japan, november 1997. Chapman & Hall.

[23] L.M.G. Feijs and S. Mauw. MSC and data. In Yair Lahav, Adam Wolisz, Joachim Fischer, and Ec khardt Holz, editors, SAM98 - 1st Workshop on SDL and MSC. Proceedings of t he SDL Forum Society on SDL and MSC, number 104 in Informatikberichte, pages 85-96. Humboldt-Universitt Berlin, 1998.

[24] S. Mauw, M.A. Reniers, and T.A.C. Willemse. Message Sequence Charts in the software engineering process. Report 00/12, Department of Computer Science, Eindhoven University of Technology, 2000.

[25] Ekkart Rudolph, Peter Graubmann, Jens Grabowski. Tutorial on Message Sequence Charts. In: Tutorials of the 7th SDL Forum, Oslo, Norway, September 25-29, 1995. Message Sequence Charts. (MSC2000). ITU-T, 1999.

[26] S. Leue, L. Mehrmann and M. Rezai, Synthesizing ROOM Models from Message Sequence Chart Specifications, 13th IEEE Conference on Automated Software Engineering, Honolulu, Hawaii, October 1998. Based on Technical Report 98-06.

[27] H. Ben-Abdallah and S. Leue, MESA: Support for Scenario-Based Design of Concurrent Systems, in: B. Steffen (ed.), Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98, Lisbon, Portugal, March/April 1998, Vol. 1384 of Lecture Notes in Computer Science, p. 118 - 135, Springer Verlag, 1998.

[28] H. Ben-Abdallah and S. Leue, Timing Constraints in Message Sequence Chart Specifications, in: Formal Description Techniques X, Proceedings of the Tenth International Conference on Formal Description Techniques FORTE/PSTV'97, Osaka, Japan, November 1997, Chapman & Hall, to appear.

[29] H. Ben-Abdallah and S. Leue, Expressing and Analyzing Timing Constraints in Message Sequence Chart Specifications, Technical Report 97-04, Dept. of Electrical and Computer Engineering, University of Waterloo, April 1997.

[30] H. Ben-Abdallah and S. Leue, Syntactic Detection of Process Divergence and non-Local Choice in Message Sequence Charts, in: E. Brinksma (ed.), Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems TACAS'97, Enschede, The Netherlands, April 1997, Lecture Notes in Computer Science, Vol. 1217, p. 259-274, Springer-Verlag, 1997.

[31] H. Ben-Abdallah and S. Leue. Syntactic Analysis of Message Sequence Chart Specifications, Technical Report 96-12, Dept. of Electrical and Computer Engineering, University of Waterloo, November1996.

[32] P.B. Ladkin and S.Leue. Interpreting Message Flow Graphs, Formal Aspects of Computing 7(5), p. 473 - 509, Sept./Oct. 1995.

[33] H. Ben-Abdallah and S. Leue. Architecture of a Requirements and Design Tool Based on Message Sequence Charts, Technical Report 96-13, Dept. of Electrical and Computer Engineering, University of Waterloo, October 1996.

[34] Ekkart Rudolph, Jens Grabowski, Peter Graubmann.Towards a Harmonization of UML-Sequence Diagrams and MSC. In: 'SDL'99 - The next Millenium' (Editors: R. Dssouli, G. v. Bochmann, Y.Lahav), Elsevier, June 1999.

[35] Jens Grabowski, Ekkart Rudolph, Peter Graubmann. The Standardization of Message Sequence Charts. In: Proceedings of the IEEE Software Engineering Standards Symposium 1993 (SESS'93), Brighton, UK, September 1993.

[36] Loc Hlout,Claude Jard, Benot Caillaud, An Effective equivalence for sets of scenarios represented by HMSCs, Rapport de recherche INRIA n0 3499, Septembre 1998.

[37] Hubert Canon, Loc Hlout, From High-level Message Sequence Charts to BDL specifications, Research report, March 1998.

[38] A. Engels: Design Decisions on Data and Guards in MSC2000. In: S. Graf, C. Jard and Y. Lahav (editors): SAM2000. 2nd Workshop on SDL and MSC, pages 33-46. Col de Porte, Grenoble, June 2000.

[39] F. Khendek, G. Robert, G. Butler and P. Grogono,"Implementability of Message Sequence Charts", Proceedings of the SDL Forum Society International Workshop on SDL and MSC (SAM'98), Berlin, Germany, June 29 - July 01, 1998.

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

[41] D.Harel and H. Kungler, Synthesizing State-Based Object system from LSC Specifications, Proc. Fifth Int. conf on Implementation and Application of Automata (CIAA 2000), Lecture Notes in Computer Science, Springer-Verlag, July 2000, to appear.

[42] Ingolf Krger, Radu Grosu, Peter Scholz, Manfred Broy, From MSCs to Statecharts, Kluwer Academic Publishers, 1999

[43] Stefan Loidl, Ekkart Rudolph, Ursula Hinkel, MSC'96 and Beyond - a Critical Look, Elsevier, 1997

[44] A. Engels: Message Refinement. Describing Multi-Level Protocols in Message Sequence Charts. In: Y. Lahav, A. Wolisz, J. Fischer and E. Holz (editors): SAM'98. 1st Workshop on SDL and MSC. Proceedings fo the 1st Workshop of the SDL Forum Society on SDL and MSC. Berlin, Germany, 29th June - 1st July 1998. Humboldt-Universitt zu Berlin, 1998. Informatikberichte Number 104.

[45] A. Engels, Th. Cobben: Interrupt and Disrupt in MSC. Possibilities and Problems. In: Y. Lahav, A. Wolisz, J. Fischer and E. Holz (editors): SAM'98. 1st Workshop on SDL and MSC. Proceedings fo the 1st Workshop of the SDL Forum Society on SDL and MSC. Berlin, Germany, 29th June - 1st July 1998. Humboldt-Universitt zu Berlin, 1998. Informatikberichte Number 104.

[46] P. Kosiuczenko. Time in Message Sequence Charts: A Formal Approach. Technical Report Nr. 9703, Ludwig-Maximilians-Universitt Mnchen, Institut fr Informatik, January 1997.

[47] P. Kosiuczenko. Formalizing MSC'96: Inline Expressions. Technical Report Nr. 9705, Ludwig-Maximilians-Universitt Mnchen, Institut fr Informatik, January 1997.

[48] P. Kosiuczenko, M. Wirsing. On the Semantics of Message Sequence Charts: an Algebraic Approach. To appear in Science of Computer Programming, Elsevier 2000, 32 pages.

[49] Stefan Heymer. A Semantics for MSC based on Petri-Net Components. Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM'2000), Grenoble (France), June, 26 - 28, 2000.

[50] Stefan Heymer. A Non-Interleaving Semantics for MSC. In: Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC - SAM'98 (Editors: Y. Lahav, A. Wolisz, J. Fischer, E. Holz). Informatik-Berichte Humboldt-Universitt zu Berlin, Juni 1998.

[51] Radu Grosu, Ingolf Krger, Thomas Stauner,Hybrid Sequence Charts, Technische Univerit at Munchen, TUM-I9914,1999

[52] Radu Grosu, Thomas Stauner, Modular and Visual Specification of Hybrid Systems – An Introduction to HyCharts, Technische Univerit at Munchen, TUM-I9801, 1998

[53] Prof. Manfed Broy, Technical University Munich : "On the Meaning of Message Sequence Charts",SAM'98 (Editors: Y. Lahav, A. Wolisz, J. Fischer, E. Holz). Informatik-Berichte Humboldt-Universitt zu Berlin, Juni 1998.

[54] J-P. Katoen, L. Lambert., Pomsets for message sequence charts, Proceedings 8th German Workshop on Formal Description Techniques for Distributed Systems (FBT'98), pages 197-208, Shaker Verlag, 1998. P. Graubmann, E. Rudolph, J. Grabowski: Towards a Petri Net Based Semantics Definition for Message Sequence Charts. In: SDL 93 - Using Objects; Proceedings of the 6th SDL Forum Darmstadt. Elsevier, 1993.

[55] Robert Nahm, Jens Grabowski, Dieter Hogrefe. Test Case Generation for Temporal Properties. Technical Report IAM-93-013, University of Berne, Institute for Informatics, Berne, Switzerland, June 1993.

[56] Michael Schmitt, Anders Ek, Jens Grabowski, Dieter Hogrefe, Beat Koch. Autolink - Putting SDL-based test generation into practice. In: Testing of Communicating Systems (Editors: A. Petrenko,N . Yevtuschenko), volume 11, Kluwer Academic Publishers, 1998.

[57] Anders Ek, Jens Grabowski, Dieter Hogrefe, Richard Jerome, Beat Koch, Michael Schmitt. Towards the Industrial Use of Validation Techniques and Automatic Test Generation Methods for SDL Specifications. In: SDL'97 - Time for Testing - SDL, MSC and Trends (Editors: A. Cavalli, A. Sarma), S. 245-259, Elsevier, September 1997.

[58] Ekkart Rudolph, Ina Schieferdecker, Jens Grabowski. HyperMSC - a Graphical Representation of TTCN. Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM'2000), Grenoble (France), June, 26 - 28, 2000.

[59] Ekkart Rudolph, Ina Schieferdecker, Jens Grabowski. Development of an MSC/UML Test Format. FBT'2000 - Formale Beschreibungstechniken fr verteilte Systeme (Editors: J. Grabowski, S. Heymer), Shaker Verlag, Aachen, June 2000.

[60] Jens Grabowski, Dieter Hogrefe. TTCN SDL- and MSC-based specification and automated test case generation for INAP. Proceedings of the "8th International Conference on Telecommunication Systems (ICTS'2000) - Modeling and Analysis", Nashville, March 2000.

[61] Jens Grabowski, Thomas Walter.Visualisation of TTCN test cases by MSCs. In: Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC - SAM'98 (Editors: Y. Lahav, A. Wolisz, J. Fischer, E. Holz), Informatik-Berichte Humboldt-Universitt zu Berlin, June 1998.

[62] Jens Grabowski. The Generation of TTCN Test Cases from MSCs. Technical Report IAM-94-004, University of Berne, Institute for Informatics, Berne, Switzerland, May 1994. Abstract , Postscript-File (33 p.)

[63] Jens Grabowski, Dieter Hogrefe, Robert Nahm. A Method for the Generation of Test Cases Based on SDL and MSCs. Technical Report IAM-93-010, University of Berne, Institute for Informatics, Berne, Switzerland, April 1993.

[64] Jens Grabowski, Beat Koch, Michael Schmitt, Dieter Hogrefe. SDL and MSC Based Test Generation for Distributed Test Architectures. In: 'SDL'99 - The next Millenium' (Editors: R. Dssouli, G. v. Bochmann, Y. Lahav), Elsevier, June 1999.

[65] Zhen Ru Dai. TTCN Test Suite Generation with Autolink - Applied to a 3rd Generation Mobile Network Protocol. Diploma.

[66] Jens Grabowski, Thomas Walter. Towards an Integrated Test Methodology for Advanced Communication Systems. Proceedings of the "16th International Conference and Exposition on Testing Computer Software (TCS'99), Theme: Future Trends in Testing", Washington D.C., June 1999.