Masarykova univerzita
Fakulta informatiky

Message Sequence Charts properties and checking algorithms

Master Thesis

Bc. Jindřich Babica

Brno, 2009

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: RNDr. Vojtěch Řehák, Ph.D.

Abstract

Message Sequence Chart (MSC) is a widely used formalism for description of telecommunication systems. A lack of tools to support MSC led us to start development of a new tool – Sequence Chart Studio (SCStudio). Using this tool one should be capable to design a system in the MSC notation, moreover, the tool will provide automatic checks of several properties of the system.

The aim of this thesis was to study MSC and according to its notation set up basis for the SCStudio. We have designed a data structures of MSC, provided functions for their processing and defined several basic properties of MSC. We also implemented checking algorithms for these properties and trace race condition property introduced in separate paper.

Keywords

Message Sequence Chart, Basic MSC, High-level MSC, Sequence Chart Studio, SCStudio, properties of MSC, automatic check

Acknowledgement

I would like to thank my supervisor Vojtěch Řehák who guided me for last terms during my work on this thesis. I really appreciate his valuable comments, constant support and his approach to students in general.

I would like to express my deep gratitude to Petr Gotthard for his countless discussions and his interest in the results of this thesis. His suggestions were very inspiring for me.

I am very grateful to both of them for their careful reading of draft of this thesis and their comments regarding my English.

I would also thank my wife Sylvie, my son and parents for their patience and support which significantly helped me to finish this thesis.

Contents

1 **Introduction** 2
2 **Message Sequence Chart Standard** 4
2.1 *Basic MSC* 5
2.2 *High-level MSC* 10
2.3 *Summary* 12
3 **Representation of BMSC and HMSC** 13
3.1 *Notation* 14
3.2 *BMsc* 15
3.3 *HMsc* 16
3.4 *Dynamic Attributes* 20
3.5 *Counter Examples* 22
3.6 *Traversers* 23
4 **Checking algorithms** 28
4.1 *Deadlock Checker* 28
4.2 *Livelock Checker* 31
4.3 *Acyclic Checker* 32
4.4 *Channel Mapping* 34
4.5 *FIFO Checker* 36
4.6 *Race Checker* 37
4.7 *Other Properties* 42
5 **Integration with SCStudio** 43
5.1 *Checking Algorithms' Interface* 43
5.2 *Build Process and Directory Structure* 44
6 **Conclusion** 46

Chapter 1

Introduction

Modeling of (distributed) systems is widely used as an approach how to get under control their complexity, comprehend their functionality and last but not least how to detect many mistakes made by designers of such systems. These mistakes are inevitable due to human fallibility.

There are many modeling techniques and languages which are able to describe a system at a particular level of abstraction. The abstraction is inseparable aspect of every modeling technique. A natural and frequently used technique of modeling complex distributed systems is based on a hierarchical division of the system into smaller subsystems–components which are more obvious.

Component based modeling is exactly the approach that is commonly used to deal with very complex system behavior. Using this approach, the basic components of the system are identified. Each of the components provides usually separable services for other parts of the system.

Furthermore, it is necessary to specify the inner behavior of these components, i.e. how the services that they offer to an environment are provided and which interfaces are used to do so.

Behavior of particular components is described by the finite state models. The states represent inner states of the components and transitions between these states determine behavior which is observable from within the components. The components are allowed to transmit information among each other. Information exchange is observable from outside of components. Depending on a used type of communication, the transmission can be provided in different ways. For example, Specification and Description Language (SDL, [12]) provides asynchronous type of communication with input queue for incoming messages, whereas UPPAAL and its description language ([8]) provide synchronous type of communication without queue of messages.

Scenario based modeling is a little bit different concept than the component base modeling. Besides the fact that it is necessary to identify the components of the system there aren't many common attributes. Instead of modeling inner behavior of the components of the system, designer of the system focuses in a message passing between these components. More precisely, designer should describe the system as a set of message passing scenarios, that corresponds to communication observable from outside of the components.

These kinds of system modeling provide a different level of abstraction and usually are used together to describe the system as unambiguously as possible.

Message Sequence Chart (MSC) is a typical representative of the scenario based modeling approach. MSC was originally described by International Telecommunication Union (ITU) in [13]. This modeling language is widely used and with few modifications (see [11]) is a part of UML.

As well as any other formalism, system described by MSC is prone to contain design defects. System design can exhibit behavior which can differ from intended behaviour of the system. E.g. the system can be able to reach a configuration from which no other action can be performed – usually, this is not the situation which the designer intended. Some of these defects can be (automatically) detected in early stage of development and so, lot of limited resources can be saved.

Unfortunately, MSC suffers from lack of professional tools which would provide not only capability for a design creation process, i.e. sketching the system in MSC notation, but design check process too.

In this thesis we will focus on MSC as modeling/design tool. We will introduce basic features of MSC. After we will familiarize with these basic features, we will separate their subset which we identify to be sufficiently strong to express quite interesting behavior of system but not to much to be unable to process the system design by automatic techniques and verify some properties of such system. Moreover, we will design a data representation of the separated subset. Besides the data structure we will describe a number of utilities which are capable to process this structure in a different way and which were implemented for these purposes.

Several properties of MSC will be defined. Together with these properties we will introduce corresponding checking algorithms which verify the system design against these properties.

Results of this thesis will serve as a basis of a Sequence Chart Studio (SCStudio) – a new tool for drawing and verification of Message Sequence Chart being developed in conjunction with ANF DATA Ltd. The current version of SCStudio can be found at [3].

Chapter 2

Message Sequence Chart Standard

MSC is introduced by Telecommunication Standardization Section of ITU (ITU-T) in [13] in the following way: *"Message Sequence Chart (MSC) is a graphical and textual language for the description and specification of the interactions between system components. Message Sequence Charts are mainly used as a specification of real-time system behavior, in particular of telecommunication switching systems. Message Sequence Charts may be used for requirement, interface and test-case specification, simulation, validation and documentation of real-time systems."* (the specification was previously discussed in [5], which this chapter is inspired with, lot of figures in this chapter are taken from [13]).

The main idea of MSC is to model the system by a set of individual **system runs** consisting of message exchanges between processes. A **run** is a explicitly specified deterministic sequence of communication acts between processes. No knowledge of inner workings of processes is needed or expected.

Core MSC language is called Basic Message Sequence Chart. A Basic Message Sequence Chart (BMSC) describes a sequence of communication exchanges between a set of processes in the system. Advanced descriptive possibilities are provided by High-level Message Sequence Charts (HMSC). In contrast to BMSC, High-level MSC is intended as a description of relations between BMSC's in the sense of particular sequence of execution of individual BMSC. Combining BMSC and HMSC, system designer provides set of possible runs of designed system.

Note that MSC contains both textual and graphical form and their expressiveness is equivalent. To ease readability of this document, we have decided to use just the graphical form. For textual form syntax or other details please follow [13].

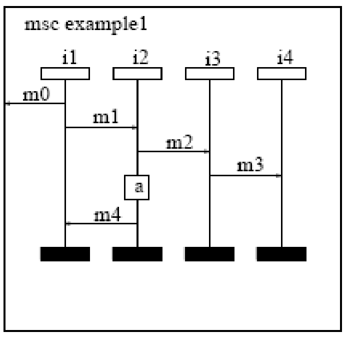

Figure 2.1: Basic MSC example

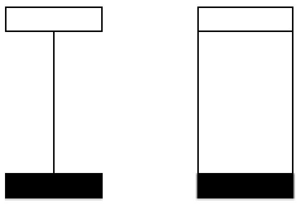

Figure 2.2: Two forms of instance

2.1 Basic MSC

Basic MSC is formed by a finite collection of process **instances** surrounded by frame with name of MSC (see Figure 2.1). An instance can be drawn in two ways shown in Figure 2.2. The reason for this duality will be explained later.

Message transmissions (possibly with a particular name) are depicted as labeled arrows. They start at sending instance (**send** or **outgoing event**) and end at receiving instance (**receive** or **incoming event**). **Local action** is denoted by rectangle with action string inside. Action string has no special semantics.

Time progress, and therefore, also an order of events on one instance is from top to down. It is assumed that all events (message output, message input, local action) consume no time but delay between two acceding events can be completely arbitrary. No global notion of time in the system is assumed.

It is possible to describe process creation and termination by Basic MSC (shown in Figure 2.3). Dashed arrow is called **create-line** symbol.

Timer handling is also very simple to describe, see Figure 2.4. The first couple in this picture shows set event of timer-horizontal or bent line to hourglass with name of timer. Parameters can be passed to timer in set event but have no semantic meaning. Second picture in Figure 2.4 depicts stop of timer and the last one shows timeout.

It is possible in Basic MSC to define a message to be **lost** or spontaneously **found** – see Figure 2.5 for syntax details.

Conditions are also supported by MSC. Generally, these conditions do not have any specific semantic meaning. They can be used as a label for a

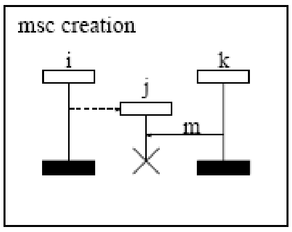

Figure 2.3: Creation and termination of process

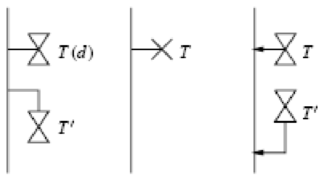

Figure 2.4: Timer handling (set, stop, timeout)

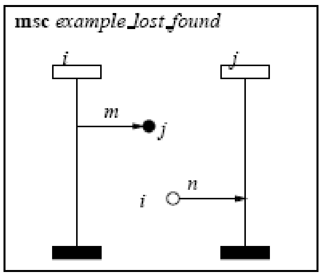

Figure 2.5: Lost/found message

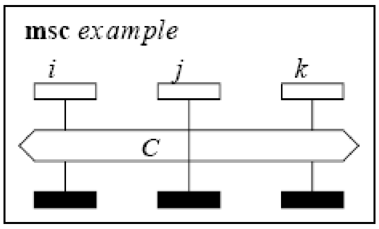

Figure 2.6: Condition example

specific situation that has occurred on the instance, and therefore, improve BMSC's readability. Example of condition which holds for more than one instance is shown in Figure 2.6. It describes a situation, when condition C holds for instances i, k but not for j (this is denoted by j's axis drawn through).

Ordering Facilities

So far, events on an instance were totally ordered in time. Sometimes it might be useful to be able to specify, that some events on the same instance may be completely unordered. This can be defined in MSC as a **coregion**. An example of a possible coregion usage is in Figure 2.7. In this figure incoming event of message m and outgoing event of message n are unordered on instance i but they are executed after output of message k and input of message l.

Because input of a message arises naturally after its output, we can use this information to get a partial order of all events across different instances. Ordering of events from Figure 2.7 is shown in so called connectivity graph in Figure 2.8. Of course, as all processes are not bind together by message

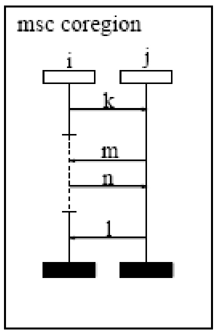

Figure 2.7: Co-region example

$$\begin{tikzcd}
\text{out(k)} \arrow[rr] \arrow[dr] \arrow[dd] & & \text{in(k)} \arrow[d] \\
& \text{in(m)} \arrow[dd] & \text{out(m)} \arrow[l] \arrow[d] \\
\text{out(n)} \arrow[rr] \arrow[dr] & & \text{in(n)} \arrow[d] \\
& \text{in(l)} & \text{out(l)} \arrow[l]
\end{tikzcd}$$

Figure 2.8: Ordering of events

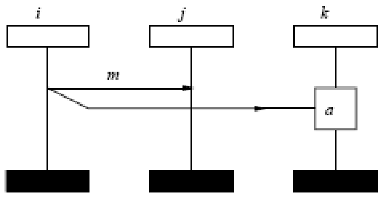

Figure 2.9: General ordering

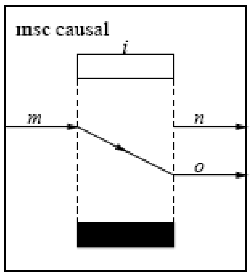

Figure 2.10: General ordering in coregion

exchange, some events may be still unordered. Above this partial order, other order relations emerging for example from unmodeled parts of system might be explicitly specified by an MSC construct called **general ordering**. An example can be seen in Figure 2.9 where the local action a at instance k occurs after output of message m (not necessarily immediately after the output). Figure 2.10 depicts use of general ordering in a coregion. It shows that input event of message m and output event of n and o are generally unordered except that o must occur after m.

Vertical, Horizontal and Alternative Composition

MSC specification provides several operators for composing MSCs (HMSC as well as BMSC). These operators are **vertical composition** (operator **seq**), **horizontal composition** (operator **par**) and **alternative composition** (operator **alt**).

Composing two MSCs using operator seq results in an MSC where all

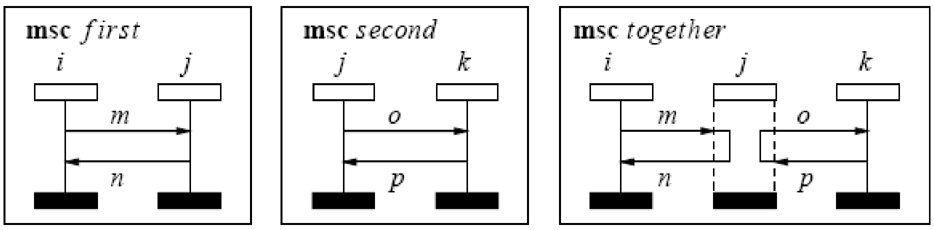

Figure 2.11: Horizontal composition

events from an instance of the second MSC have to occur after the events from the same instance of the first MSC.

Horizontal composition of two MSCs provides MSC which exhibits interleaving behavior of first MSC and the second one on common instances. Example is shown in Figure 2.11.

Alternative composition is used to describe several possible behavior of system. If there is MSC A and B an expression A alt B means that A xor B is executed.

Inline Expressions

Inline expressions are a way, how to describe nontrivial sets of system runs while keeping the MSC diagrams simple. To depict alternative or horizontal composition we use inline expressions in a way shown in Figure 2.12. The first MSC indicates that output and input of message m occurs on instances i, j or output and input of message n occurs on the same ones. The second one indicates that the two parts delimited by dashed line are interleaved when executed.

The MSC language gives us also the opportunity to describe a situation when a part of MSC is executed a specified number of times (e.g., anything from 2 to infinity, 3 to 5 times, etc.). This can be also written as an inline expression, more precisely as a **loop expression** which is followed by loop boundary. Loop boundary refers to the number of possible repeated vertical composition of content of this inline expression and indicates minimal and maximal number of repetitions. Figure 2.13 shows usage of loop inline expression and its semantically equivalent MSC.

MSC Reference Expressions

Real systems are often very complex , and therefore, it is natural requirement for a specification language to offer tools to cope this complexity. MSC

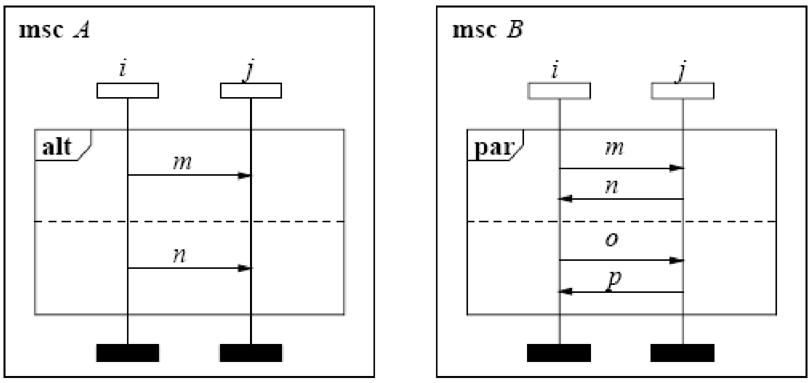

Figure 2.12: Inline expressions

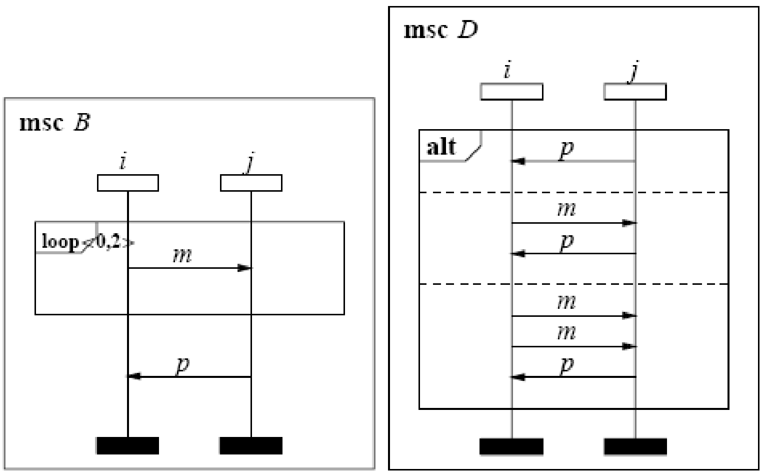

Figure 2.13: Loop inline expression

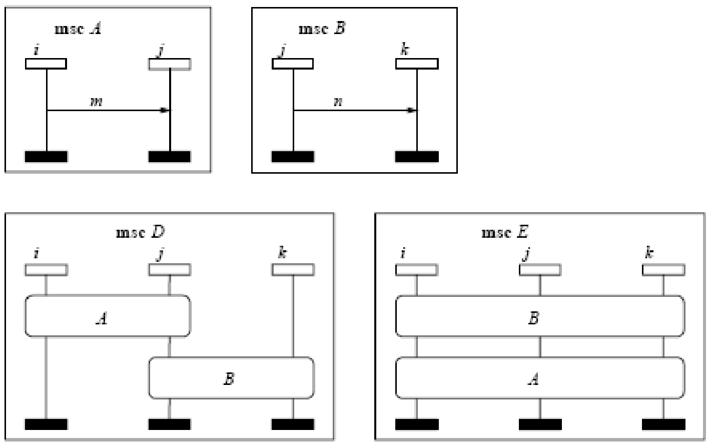

Figure 2.14: MSC reference expressions

reference expressions are one of these tools.

MSC reference expressions are used to refer to another MSC by using it's name. The basic idea is that referencing an MSC is semantically equivalent to pasting the whole MSC to the same place. It is allowed to use composition operator (alt, seq, par) inside reference expression. Figure 2.14 depicts example of MSC reference expressions' usage.

Gates

MSC recommendation describes how to define an interface of individual BMSC. For this purpose **gates** are used. There are two types of gates: **message gates** and **order gates**. Message gates are used for message events and order gates are used for ordering of events.

Input and output gates are depicted as message arrows (order arrows) connected to surrounding frame of BMSC where the name of gate is presented. Example of gates' usage is shown in Figure 2.15. Note that it is allowed to send message x to gate y (and conversely) of reference expression if and only if there is an input gate y for message x in the referred MSC.

2.2 High-level MSC

High-level MSC provide an easy and transparent way how to combine several MSC together. Syntactically, High-level MSC is a directed graph where the nodes represent other MSCs and vertices imply an order of the nodes.

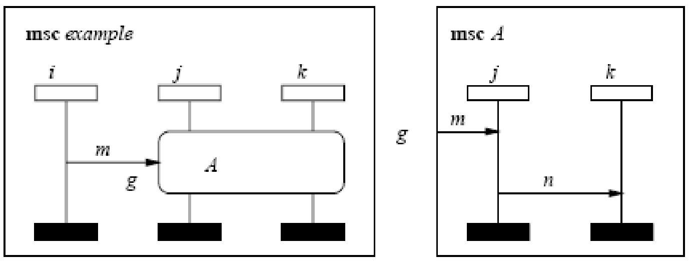

Figure 2.15: Gates' usage

Nesting of MSCs is allowed only in its finite form (recursive nesting is forbidden). One MSC can be included by more than one node of HMSC – this makes it possible for individual MSCs to be called similarly as functions in programming languages. An HMSC provides also other elements:

- **start node** ▽
- **end node** △
- **msc reference node** ▭
- **condition node** ⬡
- **connection node** ○
- **parallel frame** ▭

Start node, that is mandatory in every HMSC, determines initial point of HMSC. Analogically, end node indicates the terminal node of HMSC. MSC reference node refers to other HMSC or BMSC by its name. See Figure 2.16 for example of HMSC notation.

Semantics of HMSC is easy to explain by operators introduced so far and recursive substitution of graph vertices with corresponding MSCs. If nodes are connected via one arrow (vertex) they are exactly vertically composed (operator seq). If a node has more than one outgoing arrow, then all following nodes are alternatives (operator alt) for the vertical composition with ancestor node. Parallel frame indicates horizontal composition (operator par) of its content. Condition node has no defined semantics it has got only descriptive meaning. Connection node disambiguates crossing lines from splitting lines.

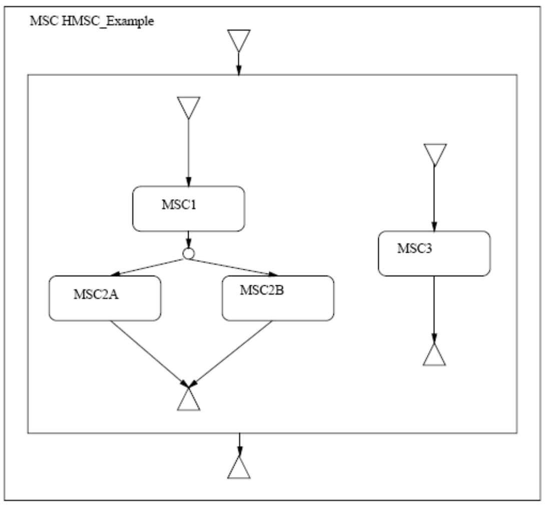

Figure 2.16: HMSC example

2.3 Summary

We have shortly introduced general facilities of MSC and informally described their semantics. For formal semantics please see ITU-T Recommendation Z.120 (Annex B) by which this text was inspired.

MSC provides wide spectrum of features for description of system behavior. Much less positive is tools' support of these features. There are only few tools (commercial mostly) that provide almost all features described in MSC formal specification.

Single BMSC or sequence of BMSC's (described by HMSC) should represent intended system behavior (its runs). MSC assumes as little as possible about inner structure of individual processes and the environment surrounding the system. It only states, that each event at each instance should be performed by target component in the same order as is specified by MSC.

System behavior described by MSC may introduce some inconsistency to implementation of the system (in many possible ways). Therefore, it is necessary to check MSC's design properties preferably through the use of automatic checking methods.

Chapter 3

Representation of BMSC and HMSC

We have designed a data structure for maintaining BMSC and HMSC. It is important to note that our aim wasn't to be able to describe all elements of ITU-T standard using this data structure. We wanted to describe those elements that are strong enough to express interesting properties of desired system but don't bring into system too expressive tool for being able to process such system automatically.

Simply said we have involved into our data representation simplified version of ITU-T standard including message exchanging, lost and found messages and coregions in BMSC, start nodes, end nodes, connection nodes and reference nodes (referencing both HMSC and BMSC) in HMSC.

Unlike the ITU-T standard we don't use the two forms of MSC representation – textual and graphical syntax of MSC, which are two inseparable parts of MSC representation. We decided to describe MSC using only one structure. The structure will cover the both textual and graphical syntax and combine their advantages together.

During designing the data structure for representation of BMSC and HMSC it was naturally necessary to take account of computational complexity of traversing such a structure as well as take account of convenient using this structure by programmers. We could define custom kind of iterator which would be hardcoded in the structure and would take care of traversing whole structure. Programmer using this iterator wouldn't have to take care of traversing the structure.

However, the kind of traversing of the structure one uses is strongly dependent on the problem one solves. Therefore, it is not very desired to limit the programmer to use only one kind of traversing inseparable from the data structure itself. We will describe so called traversers for traversing the data representation.

At first let's have a look at details of notation used in this thesis.

3.1 Notation

BMSC and HMSC are quite complex structures. In order to be able to specify properties of these structures, it would be necessary to formally define these structures. Suppose that we want to formally define BMSC. BMSC has got its events, ordering among events, labels of messages, lost messages etc. Every this part of BMSC would need its own definition – some type of relation. Due to complexity of the structures, such a great amount of different definitions wouldn't be tolerable.

Moreover, these definitions would significantly coincide with our data structure. Therefore, we have decided that the data structures will serve as formal base for our thinking about BMSC and HMSC instead of redundant formal definitions.

This approach has got several advantages besides saving reader's tolerance during reading amount of formal definitions. The data structure contains whole information (directly or indirectly) which would be involved in the definitions. In case we would have two parallel worlds – the data structures and formal definitions, the reader would have to switch between these two notations and translate each knowledge from the first one to the second one and vice versa. This process will be omitted in our case, therefore, the reader or any other person who would like to modify current implementation of checking algorithms (they will be introduced later) will be in much easier situation. The last great advantage is a graphical representation of our data structure using class diagram which is a well known data description tool. This promises fast understanding of used notation.

Elements of the class diagrams described in the following sections serve as a domain of values. Let's show it on an example. Let's suppose we have an $HMsc$ ($BMsc$) class for representation of HMSC (BMSC) and we want to talk about h as it would be an instance of $HMsc$. For this purpose we will use a notation $h \in HMsc$.

Moreover, let us suppose that there is an association class $NodeRelation$ in our class diagram which represents relation of predecessors and successors, then we will say that n is a predecessor of m in h if and only if $(n, m) \in NodeRelation_h$. Here we suppose $\forall h \in HMsc : NodeRelation_h \subseteq NodeRelation$.

It will be also necessary to specify order of events in coregion. Let's suppose there is an association class $CoregionEventRelation$ which stands for general ordering construct (Section 2.1). Let an event e be connected by this construct with an event f in $b \in BMsc$. Then we will say that

$(e, f) \in CoregionEventRelation_b \subseteq CoregionEventRelation.$

Superclasses in our diagram will be used as a union of the derived classes: if classes B and C are both derived from class A and doesn't exist any other class D derived from A we will use $A = B \cup C$.

If it isn't said otherwise, attributes defined in UML diagram which reference collection of objects are supposed to be sets: let object $a \in A$ and A is a class which has got defined attribute *friends* referencing collection of A instances in the UML diagram then $a.friends$ is supposed to be $a.friends \subseteq A$.

Value of an attribute referencing any member of the UML diagram is supposed to be member of the class: let A has an attribute *friend* referencing instance of A then for object $a \in A$ we can say that $a.friend \in A$.

In the following sections we will introduce the data structure used to represent BMSC and HMSC in details. Let's note that we will distinguish the ITU-T BMSC and HMSC from our representation by different way of typing – *BMsc* for BMSC and *HMsc* for HMSC.

3.2 *BMsc*

Figure 3.1 shows UML diagram of *BMsc* structure. The data structure as well as *HMsc* structure admits to be traversed in both direction – forward as well as backward traversing, moreover, every element of *BMsc* (the same rule holds for *HMsc*) knows (via reference) its superior element. This allows simple navigation in elements hierarchy.

BMsc is initial class for BMSC representation. It is derived from an abstract class *Msc*. Every instance (do not confuse "instance" with "*Instance*" which stands for class notation) of *BMsc* contains a list of *Instance*'s. A class *Instance* is a counterpart to ITU-T instance. An attribute *form* of *Instance* determines a form of drawn *Instance* – as specified by ITU-T this can take the value of a line form or a column form (see Section 2.1).

Instance's contain doubly-linked list of *EventArea*'s. *EventArea* is an abstract class and its derived classes are *StrictOrderArea* and *CoregionArea*. *StrictOrderArea* represents area of *Instance* where *Event*'s (*StrictEvent*'s) are linearly ordered (follow each other), unlike *CoregionArea* represents area of *Instance* where the fact that the *Event*'s (*CoregionEvent*'s) are ordered must be explicitly set. *CoregionArea* is equivalent to the ITU-T coregion. As well as *Instance*, *CoregionArea* can be of the line form or the column form specified by attribute *form*. It is assumed that the attribute of

Instance is authoritative in the case of the column form of the *Instance*.

Every *Event* references an instance of an *MscMessage*. The *MscMessage* is an abstract class with specialized types *CompleteMessage* and complementary *IncompleteMessage*. *CompleteMessage* stands for messages where both parts of communication are known (send and receive event) whereas the *IncompleteMessage* stands for lost or found message for which one of the communication sides isn't known. Direction of a message is determined by an attribute *sender* and *receiver* in case of *CompleteMessage* and by an attribute *type* in case of *IncompleteMessage*.

Horizontal position of *Event* on axis of *Instance* is provided by an attribute y (must not get over *height* of *EventArea*). Note that vertical position on axis of *Instance* in the case of the column form is determined by the direction of the message.

StrictEvent's of *StrictOrderArea* are organized as a doubly-linked list. *CoregionEvent*'s need to be able to express arbitrary ordering in coregion which is ensured by attribute *successors* of *CoregionEvent* which is a set of *CoregionEventRelation*'s (association class). This class is necessary to be able to express the ordering of events graphically – an attribute *line*. The *CoregionArea* class contains a list of all *CoregionEvent*'s in this *CoregionArea* (attribute *events*) – this simplifies memory management and allows to iterate the events in a simple way.

Example

Let's show simple but representative example of this structure to get more familiar with it. The example is shown in Figure 3.2. Figure 3.3 depicts an object diagram of this example and should clarify eventual ambiguities.

3.3 *HMsc*

UML diagram of *HMsc* structure is shown in Figure 3.4. Let's introduce its elements individually step by step.

As well as in case of *BMsc*,*Msc* represents an abstract class for *HMsc*. Purpose of this class will be clear as soon as we describe *ReferenceNode* class. *PredecessorNode* and *SuccessorNode* are abstract classes intended to specify a particular type of node in *HMsc*. *PredecessorNode* specifies predecessors and *SuccessorNode* specifies successor of another node. Relation between these two classes is shown in class *NodeRelation* which acts as an association class which is able to hold graphical representation of the relation. There is again kept a rule to be able to traverse *HMsc* in both

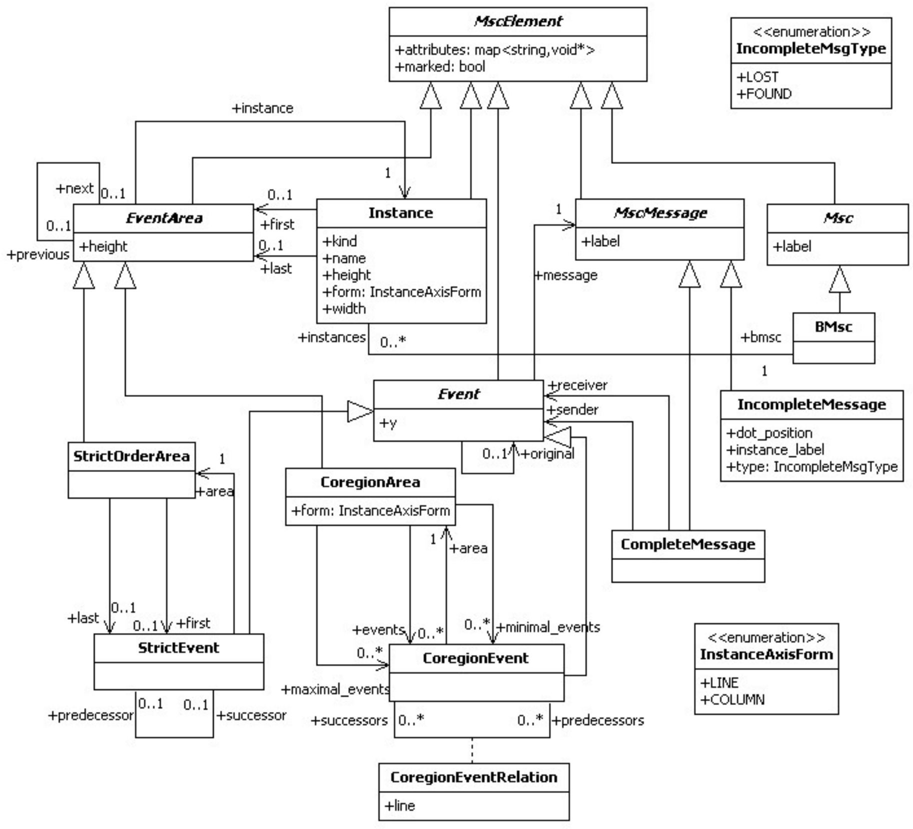

Figure 3.1: $BMsc$ UML diagram

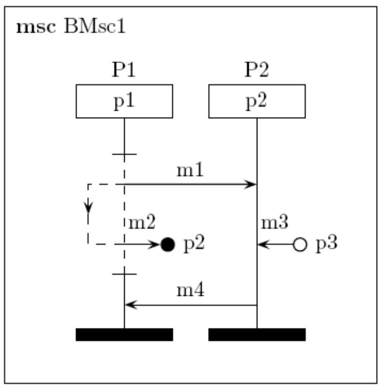

Figure 3.2: Example of BMSC – BMsc1

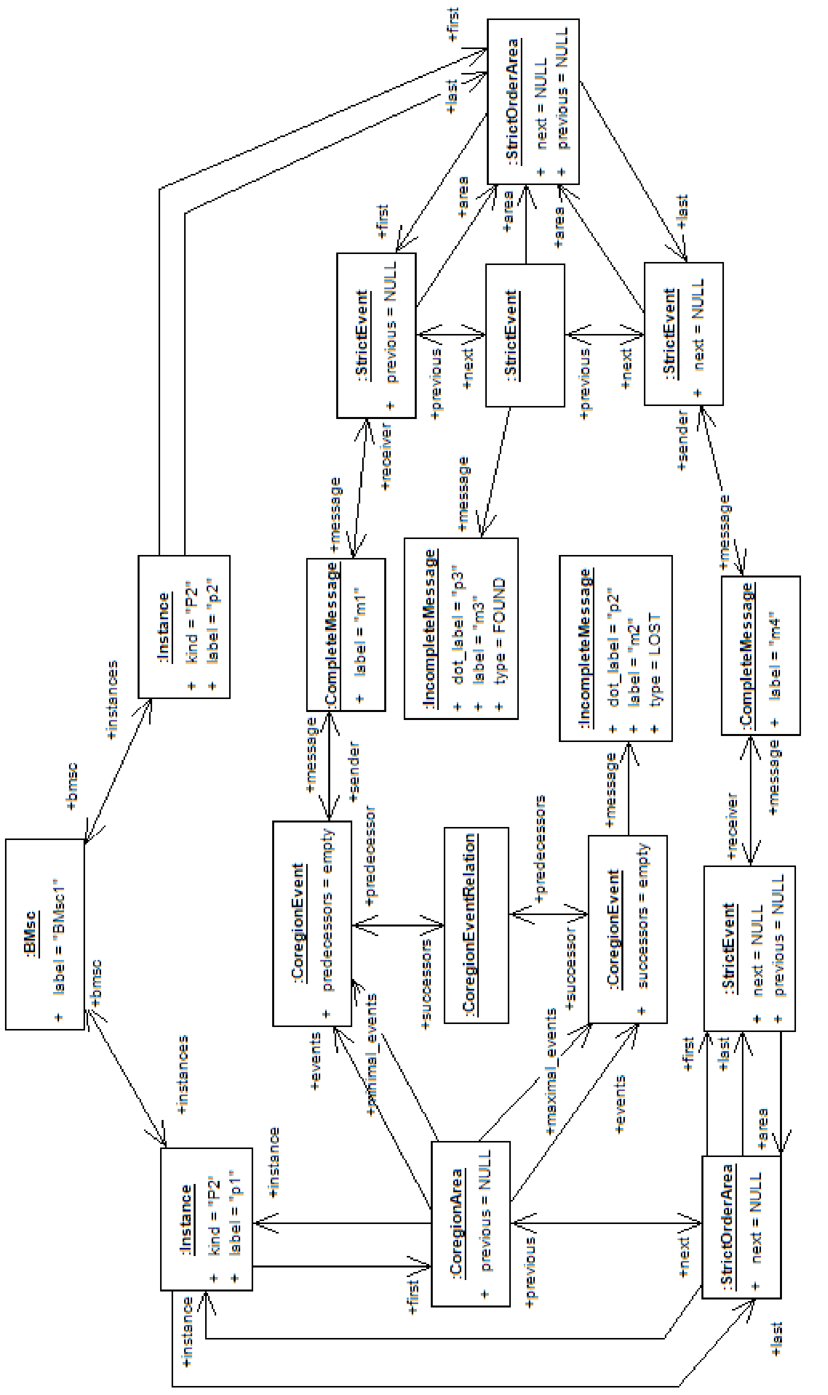

Figure 3.3: Object diagram of BMsc1

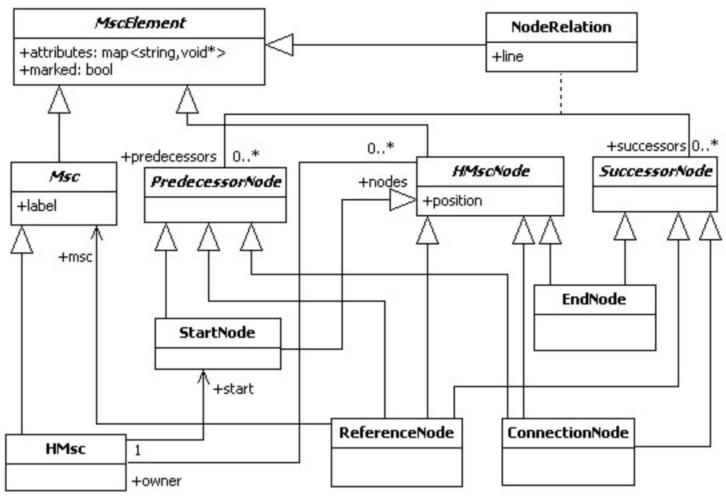

Figure 3.4: *HMsc* UML diagram

directions, therefore, *PredecessorNode* has references to its successors and *SuccessorNode* has references to its predecessors.

StartNode, *EndNode*, *ConnectionNode* and *ReferenceNode* are equivalents to similarly named elements of HMSC, they are derived from common superclass *HMscNode*. *StartNode* is a starting point of *HMsc*, whereas *EndNode* its a terminating point, *ConnectionNode* helps with drawing of transitions and *ReferenceNode* references *BMsc* or *HMsc* by an attribute *msc*. Every *HMsc* contains exactly one *StartNode* referenced by an attribute *start*. Other nodes of the *HMsc* are referenced by an attribute *nodes* of the *HMsc*.

Derived classes from the *PredecessorNode* – *ConnectionNode*, *StartNode* and *ReferenceNode* are the only classes which are able to act as predecessors in *NodeRelation*. *EndNode*,*ConnectionNode* and *ReferenceNode* are only successors. The *StartNode*, *EndNode*, *ConnectionNode* and *ReferenceNode* are subclasses of *HMscNode* which specifies position of node in *HMsc*.

Example

Figure 3.5 depicts a simple HMSC with three reference nodes. We provide its object diagram for better understanding relationships between individual elements of its *HMsc*. The diagram is shown in Figure 3.6. Note that there aren't details for referenced HMSC and BMSC with labels "HMsc2"

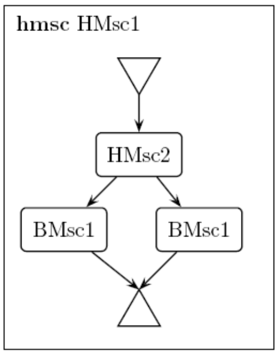

Figure 3.5: Example of HMSC– HMsc1

and "BMsc1" in the diagram.

3.4 Dynamic Attributes

During processing of *BMsc*/*HMsc* it is necessary to store extra information about its elements. In order to retrieve and set the information to arbitrary element of *BMsc* or *HMsc* for checking algorithms purposes we could declare every one attribute in the intended element. However, our aim was to separate data (*BMsc*,*HMsc*) and checking algorithms into interconnected but standalone part of the system where one is able to provide new checking algorithms without need of change in the data part.

For this purposes there are among others these methods of *MscElement* handling this requirement:

```
template<class T>
T& get_attribute(const std::string& name,
const T& def)
```

```
template<class T>
void remove_attribute(const std::string& name)
```

These methods are well documented in source code, let's note only few details. As can be seen from the declarations of the methods, dynamic attributes of *MscElement* are accessible via their name. The `get_attribute` method retrieves dynamic attribute of *MscElement* in case it is set otherwise `def` value is copied into attribute and returned.

The attributes are held in `std::map<std::string,void*>` collection. Access time to an attribute isn't, therefore, constant but logarithmic depending on number of dynamic attributes of particular *MscElement*. Note

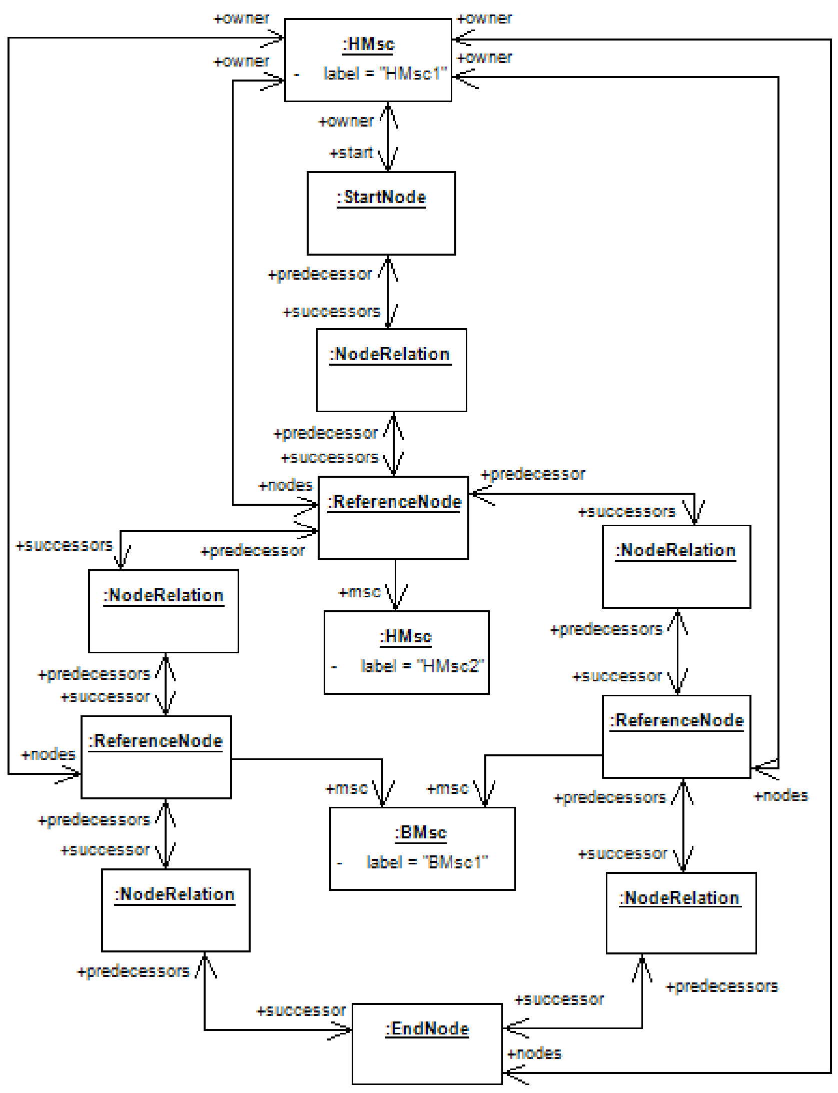

Figure 3.6: Object diagram of HMsc1

that the collection of dynamic attributes shouldn't be used to hold more than constant amount of the attributes because of possible negative side effect on a time complexity of provided tools for processing *BMsc*/*HMsc* (see Section 3.6).

Type of held attributes is an universal type – `void*`, therefore, a developer is fully responsible for deallocation of memory used by concrete attribute using method `remove_attribute`.

Dynamic attributes as they were suggested are trade-off among few requirements: as mentioned above implementation of a checking algorithm shouldn't need change to the data structure, the dynamic attribute should be accessible among more than one manipulator of the attribute and of course access time complexity to such attribute must be suitable. These requirements led us to dynamic attributes as were described in this section.

3.5 Counter Examples

If one wants to check BMSC or HMSC to satisfy particular property it is necessary to be capable to express eventual counter example if the property is not satisfied. We have identified two kinds of errors which can occur in BMSC/HMSC – static errors and dynamic errors.

Static error is an error which violates a property (we can talk about a static property too) which doesn't depend on a way HMSC is gone through. Example of such a property will be described in Section 4.3.

Dynamic error is an error violating a property (a dynamic property) which depends on a way HMSC is gone through. Example of this kind of error/property is described in Section 4.6.

We have decided to provide hierarchical information (as well as HMSC is hierarchically organized) about both types of errors to users of our system. If it is necessary to represent static error, hierarchical path in HMSC to a violating phenomenon is constructed as well as hierarchical run through HMSC in case of dynamic error. This will allow user to navigate to the error in a hierarchical way which is essence of HMSC expressivity.

Because it is assumed that our system will be able to display *BMsc* and *HMsc*, and another structure used to describe these errors would be only additional complication which should be processed, using *HMsc* and *BMsc* turned out to be the best way how to express the kinds of errors.

For these purpose an attribute *original* of *MscElement* (basic class of all *BMsc*/*HMsc* elements) was introduced. This attribute is used to reference original elements from erroneous counter example. The counter example may contain many elements which don't represent the error itself, but may be necessary to describe the error in more precise way. To distinguish these auxiliary elements and the elements which actually lead to the error we have suggested an attribute *marked* of *MscElement*. This attribute should be used to mark (highlight) the erroneous elements.

3.6 Traversers

The previous sections described the data structure used to represent BMSC and HMSC and among others explained why it is not desired to join together data representation and utilities for their iteration. In this chapter we will describe so-called *traversers* and their general usage pattern which are suggested to iterate *BMsc* and *HMsc*.

Our *traversers* are based upon Depth First Search (DFS) algorithm and notation used in them is adopted from [7]. Although we suggested *traversers* with only once type of iteration through the structure (which is not desired as explained above) the used pattern has proven to be very robust and may be very useful for different types of iterations for future.

HMsc Traversers

Base class for *HMsc* traversers is *DFSBMscGraphTraverser* (abbreviation of Depth First Search BMsc Graph Traverser). Instance of this class is capable to traverse *HMsc* like it would be a *BMsc*-graph – flattened version of *HMsc* where every *ReferenceNode* references only *BMsc* not *HMsc*. If there is some *HMsc* referenced from more than one *ReferenceNode* its nodes are traversed as many times as the *HMsc* is referenced. Traversing starts in *StartNode* of the *HMsc*. Unreachable nodes from the *StartNode* aren't traversed.

As well as used in [7] there are several states/colors which *HMscNode* of *HMsc* can occur in during traversing:

- **white** - *HMscNode* wasn't visited yet – default color
- **gray** - *HMscNode* was already visited and there exists any successor (not necessarily direct) of the *HMscNode* which wasn't finished yet (in another way said the *HMscNode* is currently on stack)
- **black** - *HMscNode* and every its successor was visited

According to these states and situation when gray *HMscNode* is colored to black one *DFSBMscGraphTraverser* contains four lists of „listeners" containing items of these types:

- *WhiteNodeFoundListener*
- *GrayNodeFoundListener*
- *BlackNodeFoundListener*
- *NodeFinishedListener*

These types are abstract classes with specified necessary method to be implemented. The methods are called when desired situation occurs:

- `on_white_node_found(HMscNode* n)` – white $n \in$ *HMscNode* was just found (method of *WhiteNodeFoundListener*)
- `on_gray_node_found(HMscNode* n)` – gray $n \in$ *HMscNode* was just found (method of *GrayNodeFoundListener*)
- `on_black_node_found(HMscNode* n)` – black $n \in$ *HMscNode* was just found (method of *BlackNodeFoundListener*)
- `on_node_finished(HMscNode* n)` – all successors of *HMscNode* and the *HMscNode* itself too are marked as black (method of the last *NodeFinishedListener*)

Imagine one wants to count number of reachable *HMscNode*'s in whole *HMsc*. The following example shows usage of *DFSBMscGraphTraverser* and *WhiteNodeFoundListener* to fulfill this requirement.

```
class NodeCounter : public WhiteNodeFoundListener{
public:
  unsigned int count;

  NodeCounter(){
    count = 0;
  }

  void on_white_node_found(HMscNode* n){
    count++;
  }
};
```

```
unsigned int count_nodes(HMsc *hmsc){
  NodeCounter counter;
  DFSBMscGraphTraverser traverser;
  traverser.add_white_node_found_listener(&counter);
  traverser.traverse(hmsc);
  return counter.count;
}
```

One can see that without traversers and listeners it would be necessary to implement iteration of *HMsc* in some way for any similar simple task. Using traverser and listener this task is much easier. Moreover, traversers holds currently reached path in *HMsc* in public read-only attribute *reached_elements* which holds references to traversed *HMscNode*'s and *NodeRelation*'s as they were in sequence visited. The attribute is useful for path reconstruction in the case we want to show user some interesting place in his design which was found by the traverser.

Note that the traversers use dynamic attributes of *HMscNode*'s to remember their color. A name of this attribute is set to be default if not specified in constructor of the traverser, however, one can change it (and it is probably intended behaviour) to avoid different traversing share the same attribute.

Traversers have public method `cleanup_traversing_attributes`. This method cleans up dynamic attributes necessary for traversing – state of node during traversing. These attributes need to be cleaned up after traversing as well as general dynamic attribute does. The cleaning is automatically performed when traversing is finished or in a destructor of the traverser. But there may be still some situations when it is necessary to call the method explicitly.

The only way how to terminate traversing is to throw exception from a listener. In this case the traversing isn't finished and there are still allocated dynamic attributes by the traverser. If one wants to reuse this traverser (not to destruct the traverser and use another one) in exception handling it is necessary to call `cleanup_traversing_attributes`.

Depth first search algorithm has got very useful feature. It is capable to identify a cycle in a graph in simple way. Paper [7] describes that whenever graph contains a cycle DFS algorithm detects it by found gray node. This feature of DFS is frequently used in this thesis.

This section described basic concept of a traverser. Besides the basic mentioned *DFSBMscGraphTraverser* there are also other "DFS" traversers.

The following list summarizes their usage:

- *DFSBMscGraphTraverser* – Traverses *HMsc* like it would be *BMsc*-graph, each *HMscNode* is traversed as many times as its *HMsc* (attribute *owner*) is referenced by any *ReferenceNode*. Moreover, successor of *ReferenceNode* isn't traversed if the *ReferenceNode* references any *HMsc* which hasn't got reachable *EndNode*.

- *DFSHMscTraverser* – Traverses referenced *HMsc*'s and their nodes only one time. Moreover, doesn't care about reachable *EndNode* in referenced *HMsc* if is going to traverse successor of *ReferenceNode*.

- *DFSBHMscTraverser* – Traverses *HMsc* in backward manner.

- *DFSRefNodeHMscTraverser* – Traverses *HMsc* in the same way how *DFSBMscGraphTraverser* does with exception that *ConnectionNode*'s are supposed to be only edges of graph and aren't involved in traversed nodes.

The traversers are well commented in source code. For details follow the comments.

BMsc **Traversers**

Traversers of *BMsc* use the similar concept as *HMsc* traversers. There are again particular listeners whose methods are called at a specified situation during traversing *BMsc*:

- *WhiteEventFoundListener*

- *GrayEventFoundListener*

- *BlackEventFoundListener*

- *EventFinishedListener*

In contrast to *HMsc* traversers where *HMscNode*'s are considered to be graph nodes there are *Event*'s in *BMsc*. Usage of *BMsc* traversers is similar to usage of *HMsc* traversers. Same rule about dynamic attribute used for traversing holds for them too.

The following list summarizes different kinds of *BMsc*-traversers and their usage:

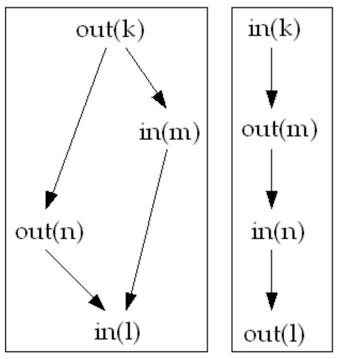

Figure 3.7: Instances' graph

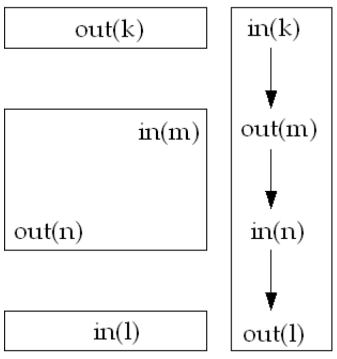

Figure 3.8: Areas' graph

- *DFSEventsTraverser* – Traverses all events of *BMsc* and follows receive opposites of send events. Traversed graph of this traverser for BMSC in Figure 2.7 can be displayed as the one in Figure 2.8.
- *DFSInstanceEventsTraverser* – Traverses separately events of individual instances, i.e. it doesn't follow matching events. Traversed graph for Figure 2.7 is shown in Figure 3.7. The rectangles enclose subgraphs which are traversed separately in sequence.
- *DFSAreaTraverser* – Traverses events of individual areas. These areas are traversed sequentially with respect to their occurrence on instance. This traverser doesn't follow any matching event. Traversed graph for Figure 2.7 is shown in Figure 3.8. The subgraphs (enclosed by rectangles) are traversed in up-to-down manner.

For details about *BMsc*-traversers see comments in source code.

Summary

A pattern of the traversers has shown during work that it is very useful and robust concept that simplifies elementary operations on *BMsc*/*HMsc* in great way. Moreover, using listeners and traversers creates logical layer between the structures (their inner representation and dependencies) and functional layer which ensures computation over these structures. The middle layer may be even more helpful in case that structure of *BMsc*/*HMsc* will change. When strictly using traversers in a code it would be necessary to change mainly only inner implementation of the traversers (in perfect world of course).

Chapter 4

Checking algorithms

This chapter defines particular simple and more complex properties of BMSC/HMSC and shows concrete implementations of checking algorithms which verify whether the BMSC/HMSC satisfies the properties using previously described data structure for description and tools for maintaining $BMsc$ and $HMsc$.

There is a section called "Implementation" for every checking algorithm which should serve as a high level description of particular implementation, and section "Counter example" which is intended to be a description how to provide a feedback to a user in the case of found error.

4.1 Deadlock Checker

Deadlock property as defined in this section is little bit different from the deadlock property as is usually described in an environment of parallel systems. Usually deadlock is defined as cyclic dependence of components on resources (or events) which are exclusively owned by different components.

To explain our deadlock property and many other features, it will be very helpful to introduce path notion in $HMsc$. Before that it is necessary to define a referenced set of $h' \subseteq HMsc$.

Definition 4.1. *Subset of HMsc referenced by object $h \in HMsc$ denoted as $Referenced_h$ is defined recursively in the following way:*

1. $h \in Referenced_h$
2. $h' \in Referenced_h \Rightarrow (\forall n \in h'.nodes \cap ReferenceNode : (n.msc \in HMsc \Rightarrow n.msc \in Referenced_h))$
3. *$Referenced_h$ doesn't contain anything else*

With this definition we can exactly define $BMsc$-graph which will be useful in future.

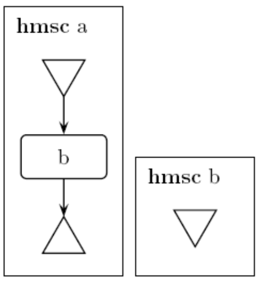

Figure 4.1: Wrong path example

Definition 4.2. *Let object $h \in$ HMsc and its $Referenced_h = \{h\}$ then h is called BMsc-graph.*

Since now we will use *h.snodes* to denote $\{h.start\} \cup h.nodes$ for $h \in$ *HMsc*.

Definition 4.3. *Path in $h \in$ HMsc is defined as a sequence $n_1, n_2, ..., n_k$ where $\forall n_i, 1 \leq i \leq k : n_i \in \bigcup\{h'.snodes | h' \in Referenced_h\}$, and $\forall(n_j, n_{j+1}), 1 \leq j < k$ hold:*

- $n_j \in StartNode \Rightarrow (n_j, n_{j+1}) \in NodeRelation_{n_j.owner}$
- $(n_j \in ReferenceNode \wedge n_j.msc \in HMsc) \Rightarrow n_{j+1} = n_j.msc.start$
- $(n_j \in ConnectionNode) \vee (n_j \in ReferenceNode \wedge n_j.msc \in BMsc) \Rightarrow (n_j, n_{j+1}) \in NodeRelation_{n_j.owner}$
- $n_j \in EndNode \Rightarrow \exists n \in ReferenceNode$ *such that* $n.msc = n_j.owner \wedge (n, n_{j+1}) \in NodeRelation_{n.owner}$

Definition 4.4. *Node $n \in$ HMscNode is called reachable in $h \in$ HMsc if there exist a path $n_1, ..., n$ in h where $n_1 = h.start$.*

Figure 4.1 shows $a, b \in$ *HMsc*. a references b via its reference node. If we denote *StartNode*, *ReferenceNode* and *EndNode* of a as s_a, r_a and e_a and *StartNode* of b as s_b then s_a, r_a or s_a, r_a, s_a are paths in a but neither s_a, r_a, e_a nor s_a, r_a, s_b, e_a aren't. Note that *DFSBMscGraphTraverser* with an appropriate listener is able to generate one type of path. Now we are ready to define deadlock of *HMsc*.

Definition 4.5. *Let $h \in$ HMsc, $h' \in Referenced_h$ and $n \in h'.snodes \setminus ConnectionNode \setminus (h.nodes \cap EndNode)$ is reachable in h. Node n is called deadlock if there doesn't exist any path $n, ..., m$ where $m \notin ConnectionNode$.*

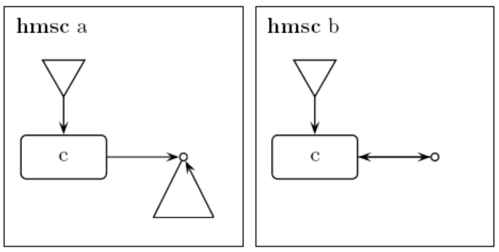

Figure 4.2: Deadlock example

Deadlock is a simple property which denotes that there doesn't exist any path from node n to any other node except $\mathit{ConnectionNode}$. This reflects the fact that $\mathit{ConnectionNode}$ is semantically empty element – it only interconnects other elements.

It is simple to show that if node n is deadlock in $h \in \mathit{HMsc}$ and there exists any $e \in h.nodes \cap \mathit{EndNode}$ then there isn't any path $n, .., e$. Opposite implication doesn't hold because of possible cycles in h (this will be called as livelock 4.2).

Figure 4.2 shows $a, b \in \mathit{HMsc}$ both referencing any $c \in \mathit{BMsc}$ via their reference nodes r_a, r_b. An r_a is deadlock in a but r_b is not deadlock in b. b is deadlock free.

Implementation

Idea of checking algorithm is to find any $h' \in Referenced_h$ and reachable node $n \in h'.snodes \cap (\mathit{StartNode} \cup \mathit{ReferenceNode})$ which doesn't occur on a cycle (otherwise at least n is reachable from n) and there isn't any path $n, ..., e$ such that $e \in \mathit{EndNode}$.

We use $\mathit{DFSBMscGraphTraverser}$ for traversing $h \in \mathit{HMsc}$. Note that $\mathit{DFSBMscGraphTraverser}$ traverses h in depth first search manner, i.e. if $n \in h.nodes \cap \mathit{ReferenceNode} \wedge n.msc = h'$ then h' and its nodes will be traversed before successors of n.

Concrete implementation is introduced in $\mathit{DeadlockChecker}$ class. To test whether node $n \in h'.snodes \cap (\mathit{StartNode} \cup \mathit{ReferenceNode})$ lies on a cycle or not we use a stack (depths) of numbers representing depth of nodes of this type during traversing. These depths are kept by nodes in dynamic attribute depth.

Whenever gray node n' is found (cycle is detected) it is checked whether $depths.top \geq n'.depth$. If this condition holds it means that there is any node $m \in (\mathit{StartNode} \cup \mathit{ReferenceNode})$ on the cycle and, therefore, each predecessor of n' is marked as deadlock free. Moreover, any predecessor of any $e \in \mathit{EndNode}$ is marked as deadlock free.

Whenever any node from $(\mathit{StartNode} \cup \mathit{ReferenceNode})$ is going to be finished (method of $\mathit{NodeFinishedListener}$) and the node is not marked as deadlock free it is found that this node is deadlock and appropriate exception is thrown.

Let suppose we will flatten $h \in \mathit{HMsc}$ into BMsc-graph h'. Complexity of deadlock checking algorithm for h' is linear with respect to number of nodes in $h'.snodes$ and number of edges in this BMsc-graph.

Counter example

Let n be a deadlock in $h \in \mathit{HMsc}$. To show to user most suitable node – or more precisely path to this node – as deadlock we will distinguish few examples:

1. $n \in \mathit{ReferenceNode}$ – It is sure that $n.msc \in \mathit{BMsc}$, node n should be shown.
2. $n \in \mathit{StartNode}$ – Node n should be shown.
3. $n \in \mathit{EndNode}$ – It is sure that there is any $m \in \mathit{ReferenceNode}$ such that $m.msc = n.owner$. Because the problem of the incomplete path from n isn't obvious in $n.owner$, node m should be shown.

4.2 Livelock Checker

Deadlock defined property of HMSC which contains a reachable node from which there isn't any path leading to some end node. Deadlock isn't the only case which can show such type of behavior. An other case is reachable cycle of nodes in HMSC from which there isn't any path to an end node. We will call this case of never ending path livelock. Unlike deadlock there exists path to another node of the HMSC in the case of livelock, but similarly as in deadlock we can never reach an end node from it.

Definition 4.6. *Let $h \in \mathit{HMsc}$ and $n_0, ..., n_k \in \mathit{HMscNode}$ reachable nodes in h such that $\exists i, 0 \leq i \leq k : n_i \in \mathit{ReferenceNode}$ and $n_0, ..., n_k, n_0$ make a cycle and there doesn't exist path $n_i, ..., e$, $e \in \mathit{EndNode}$. Cycle $n_0, ..., n_k, n_0$ is called livelock.*

Figure 4.3 shows simple HMsc with livelock which is made of node referencing $MSC2$.

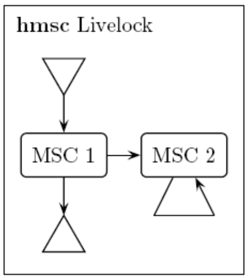

Figure 4.3: Livelock example

Implementation

Livelock checker implementation is introduced in *LivelockChecker*. It uses *DFSRefNodeHMscTraverser* to traverse given *HMsc* – this traverser doesn't take account with *ConnectionNode*'s as nodes of graph, let's denote the traverser as t_1. Whenever t_1 finds white node which is end node second traverser t_2 is triggered which is instance of *DFSBHMscTraverser*.

This kind of traverser traversers *HMsc* in backward manner. The t_2 is, therefore, able to find (with appropriate marking listener) nodes which path to the end node exists from.

So whenever t_1 finds gray node which wasn't marked by t_2 and its listener, t_1 can be sure that cycle (containing *ReferenceNode*) was found and there doesn't exist any path from the found node to any end node.

Complexity of this algorithm is same as complexity of deadlock checking algorithm, i.e. linear with respect to size of the flattened version of $h \in HMsc$.

Counter example

Livelock is made by a specific cycle. Therefore, path to the cycle and the highlighted cycle itself should be represented to the user of SCStudio in case of found livelock.

4.3 Acyclic Checker

Previous section defined and described simple property of $h \in HMsc$ which doesn't depend on any $b \in BMsc$ referenced from any $n \in ReferenceNode$. Presence of deadlock depends strictly only on graph structure of h and doesn't take account of structure of b. Following acyclic property will make almost opposite approach.

To be able to define acyclic property it is helpful to introduce visual order of events in $b \in BMsc$. Note that we use notation $Event_b = \{e \in Event | e.area.instance.bmsc = b\}$ for $b \in BMsc$. Moreover $e.is\_send$ is assumed to be $true$ if $(e.message \in CompleteMessage \wedge e.message.sender = e) \vee (e.message \in IncompleteMessage \wedge e.message.type = LOST)$ otherwise it is assumed to be $false$.

Definition 4.7. *Let $b \in BMsc$. Visual order $<$ of b is binary relation on $Event_b$ defined as follows:*

- $e_1.area \in StrictOrderArea \wedge e_1.next = e_2 \Rightarrow (e_1, e_2) \in <$
- $(e_1, e_2) \in CoregionEventRelation_b \Rightarrow (e_1, e_2) \in <$
- $e_1.message \in CompleteMessage \wedge e_1.is\_send \wedge e_1.message.receiver = e_2 \Rightarrow (e_1, e_2) \in <$
- $e_1.area.next = e_2.area \Rightarrow (e_1, e_2) \in <$
- $<$ *is reflexive and transitive*
- $<$ *doesn't contain anything else*

See again Figure 2.8 and its origin – Figure 2.7. Relation $<$ is exactly induced by the arrows in Figure 2.8.

Note that visual order ("visual" means that it is exactly what one can see in BMSC) isn't said to be order relation in general. It is caused that our design of $BMsc$ doesn't ensure visual order to be antisymmetric relation. Standard syntax of MSC [13] proscribes send event to be causally dependent on receive event of the same message. It is natural requirement but without any other constraints on our definition of $BMsc$ we aren't able to comply with this requirement.

Way how the standard syntax ensures this requirement is simple syntactic constraint that eliminates such causal dependency (shown in [6]): „An arrow representing a message may be horizontal or with downward slope." To comply with this requirement we suggest *acyclic* property.

Definition 4.8. *A $b \in BMsc$ is said to be acyclic if visual order relation $<$ of b is antisymmetric.*

Figure 4.4 shows $b \in BMsc$ which isn't acyclic. Little augmentation of this definition specifies acyclic property of $h \in HMsc$, i.e. h is acyclic if every referenced $b \in BMsc$ is acyclic.

Definition 4.9. *An $h \in HMsc$ is said to be acyclic if $\forall n \in \bigcup\{h'.nodes | h' \in Referenced_h\} \cap ReferenceNode : n.msc \in BMsc \Rightarrow n.msc$ is acyclic.*

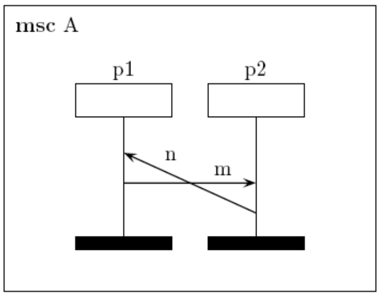

Figure 4.4: Non-acyclic example

Implementation

Implementation of checking algorithm for this property (introduced in the *AcyclicChecker*) is quite simple using *DFSEventsTraverser* because the traverser follows $<$ relation while traversing events. Cycle is detected by gray node – event in our case. Aim was to define some *GrayEventFoundListener* which throws an appropriate exception in case of gray event is found.

Complexity of this algorithm is linear with respect to size of connectivity graph which is traversed by *DFSEventsTraverser*.

Counter example

To present a cycle in $b \in BMsc$ to user it is convenient to display whole *BMsc* with marked events in the cycle. If user is checking $h \in HMsc$, $h' \in HMsc$ representing path in h to a violating $b \in BMsc$ should be displayed.

4.4 Channel Mapping

Let's turn out little bit from checking algorithms. In this section we will introduce notion of *channel mapping* (delivery semantics in another way) which in shortcut means classification of events in BMSC into relevant FIFO (First In First Out) channels – similar concept as defined in [4]. This notion will be frequently used in next sections. Let's introduce the notion deeper.

Communication systems are based on different architectures which provides various number of properties which designer of the system can rely on. Among these properties message order reliability belongs too, i.e. it is ensured that message x sent before message y via same channel will be delivered before message y.

Let's assume that we intend to describe communication between two

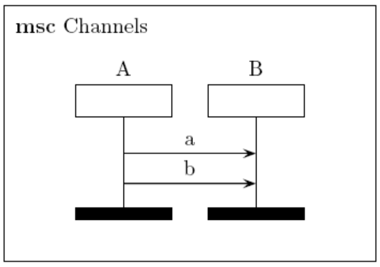

Figure 4.5: Channel mapping example

processes A and B using BMSC depicted in Figure 4.5. In case there is one FIFO channel between all pairs of communicating processes message a is ensured to be delivered before message b. In case there is separated channel for each message type and communicating pair (in this case labels of messages denotes the message type) it can happen that b will overtake a. Decision strategy about classification of events to certain FIFO channel will be called as channel mapping.

Checking algorithms should be as independent on used channel mapping as possible. Otherwise it would be necessary to implement different algorithms for different type of channel mapping. Therefore, we have already involved notion of channel mapping into our system from beginning and suggested interface (abstract class) *ChannelMapper* which should be implemented by each channel mapping strategy.

The interface specifies several methods which decides whether two different *Event*'s – their messages belong to the same channel or not. There are listed the most important of them.

```
size_t channel(const Event* event)

bool same_channel(const Event* e1,const Event* e2)
```

The first one returns identifier of channel which message of $event$ belongs to. The second one resolves whether $e1$ and $e2$ belong to the same channel or not. Implementation of these two methods is first step for new kind of channel mapping. Appreciative reader can remark, that condition `same_channel(e1,e2)` ⇔ `channel(e1)=channel(e2)` holds. Why to keep then two methods? The `same_channel` is supposed to be able to be more effective than comparison of two results of `channel`, because it is not needed to compute channels identifier.

We have suggested two implementation of this interface too: the first one is *SRChannelMapper* and the second one is *SRMChannelMapper*. The

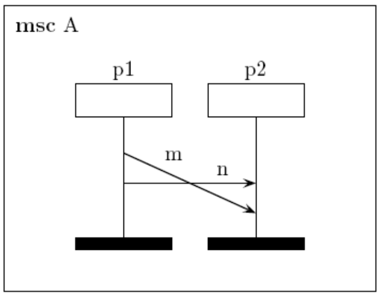

Figure 4.6: FIFO property

SRChannelMapper corresponds to single FIFO channel per communicating pair (SR=Sender Receiver), while the *SRMChannelMapper* corresponds to single FIFO channel per communicating pair and message name (SRM = Sender Receiver Message).

4.5 FIFO Checker

Many interesting properties of *HMsc* (not only those introduced in this thesis) depends on simple property of visual order relation $<$. This simple property ensures that events don't "visually" overtakes each other.

Note that we use notation $e.matching\_even$ for $e \in Event, e.message \in CompleteMessage$ to denote event $f \in Event : e \neq f \wedge (e.message.sender = f \vee e.message.receiver = f)$

Definition 4.10. *Let $b \in BMsc$ be acyclic, relation $<$ its visual order and $m \in ChannelMapper$ arbitrary channel mapper. A b is said to be FIFO with respect to channel mapper m if for all $e_1, e_2 \in Event_b : (m.same\_channel(e_1, e_2) \wedge e_1 < e_2 \wedge e_1.message, e_2.message \in CompleteMessage \Rightarrow e_1.matching\_event < e_2.matching\_event)$.*

Definition 4.11. *An $h \in HMsc$ is said to be FIFO if $\forall n \in \bigcup\{h'.nodes | h' \in Referenced_h\} \cap ReferenceNode : n.msc \in BMsc \Rightarrow n.msc$ is FIFO.*

The above definition forbids FIFO $b \in BMsc$ to contain events–messages from same channel which overtake each other. Figure 4.6 shows non-FIFO *BMsc*.

Implementation

Schema of our implementation (introduced in *FifoChecker*) can be summarized into two steps. At first compute relation $<$, at second checkout whether events' messages from same channel don't overtakes each other.

Base of relation $<$ consists of send and corresponding receive events and successive events at single $Instance$. To compute $<$ it is necessary to augment this base to its transitive closure.

Let suppose that $b \in BMsc$ is acyclic (requirement in definition of FIFO). Therefore, we are able to order events of b in topological manner, i.e. we are able to make sequence of its events $e_1, ..., e_n$ such that $\forall i, j, 1 \leq i < j \leq n$: $e_j < e_i \vee (e_j \not< e_i \wedge e_i \not< e_j)$.

Due to this structure of the problem it is possible to compute the transitive closure in more effective way than standard Floyd-Warshall algorithm as was shown in [4]. It is done by $VisualClosureInitiator$. Complexity of introduced computation of transitive closure in [4] is $\mathcal{O}(n^2)$ where n is number of events in b whereas Floyd-Warshall algorithm complexity is $\mathcal{O}(n^3)$.

The second part of the problem – verification of order of events from same channel is simple and we can use for it topological order of events created for computation of transitive closure. I.e. in sequence verify for $i, j, 1 \leq i < j \leq n$ whether $e_i.matching\_event < e_j.matching\_event$ if $e_i < e_j$ and e_1, e_2 are from the same channel. Complexity of this check is $\mathcal{O}(n^2)$.

Counter example

User should be able to see violating events in context of whole $BMsc$, therefore, We suggest to display whole $BMsc$ which violated the property with marked the violating events.

4.6 Race Checker

In this section we will little bit change our understanding of delivering of messages in BMSC. Imagine situation described by Figure 4.7. Strictly following the semantics defined by ITU-T, then delivery of message a whenever occurs before delivery of message b. However, is the ordering of the events implicit or do $p1$ and $p2$ have to ensure the ordering?

Because instances $p1$ and $p3$ don't know anything about themselves (there isn't any message interchange between them) and we don't know anything about time which take message a and b during transmition and time they were sent it can happen that message b will arrive before message a. But our figure indicates that this never occurs, it describes that message a arrives before b.

In fact the figure doesn't describe that the message a arrives every time

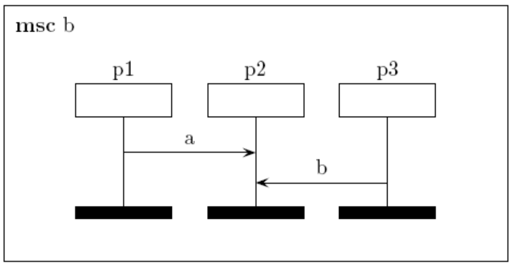

Figure 4.7: Race introduction

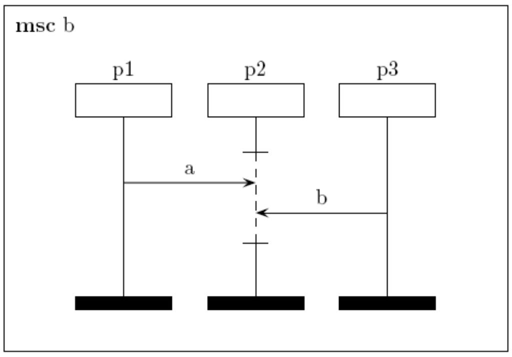

Figure 4.8: Arbitrary order of delivering

before the message b – it is impossible as we have shown, but that the message a is processed before the message b by instance $p2$. If we suppose that system exhibits exactly the described ordering of the events, there must be some kind of mechanism which ensures that a is processed before b. E.g. it can be some kind of buffer of messages in which message b is never processed before message a in particular state of instance $p2$. There was said that no knowledge of inner workings of processes was needed or expected in Chapter 2. This is not actually right because the processes need to be capable to execute proper event at specific time at least.

But what happen if we can not rely on such property of the system? Communication among these three instances should not be described as shown in Figure 4.7 but Figure 4.8 describes the situation in more proper way. The figure indicates (using coregion) that messages a, b can be delivered in arbitrary order.

Thus let's suppose for now that there isn't any such mechanism which ensures that messages are delivered in order that is described by BMSC. BMSC will describe intended order of messages delivering which can differ from implicit behavior. To distinguish between intended and implicit order, we will introduce notion of causal order. Intended order is described by visual order defined in 4.3. Note that our definition of causal order is adopted from [14] and modified for purposes of our notation.

Definition 4.12. *Let $m \in$ ChannelMapper and $b \in$ BMsc be FIFO with respect to m. Causal order $\ll$ of events in $Event_b$ with respect to m is defined as the least partial ordering on b's events such that $e \ll f$, if at least one of the following items hold:*

- $e.message \in CompleteMessage \wedge e.is\_send \wedge e.matching\_event = f$
- $e.instance = f.instance \wedge f.is\_send \wedge e < f$
- $e.message, f.message \in$ *CompleteMessage,*e, f *are receive events and there exist events* $e', f' : (e.matching\_event = e' \wedge f.matching\_event = f' \wedge m.same\_channel(e, f) \wedge e' < f')$

The first one condition says that sending of message precedes receiving of message. The second one says that each send event at any instance waits until there isn't any other event waiting to be processed before. The third one condition takes account of same FIFO channels of messages, i.e. if any message is sent before other message through a same channel, the first one message will be surely delivered before the second one.

Race in BMSC Checker

If we come back to our example in Figure 4.7 we can demonstrate on it what it is race in BMSC. Simply said the race in BMSC is difference between visual order $<$ and $\ll$.

Definition 4.13. *Let $m \in ChannelMapper$ and $b \in$ BMsc be a FIFO with respect to m and $<$ visual order and $\ll$ causal order with respect to m. Two events $e, f \in Event_b$ are said to be in race with respect to m if $(e, f) \in <$ but $(e, f) \notin \ll$.*

Let $m \in$ *SRChannelMapper* (4.4), Figure 4.7 is typical example of race because events of delivering of message a and delivering of message b are in race. Note that the events are not in race in Figure 4.8.

Implementation

The main problem in effective implementation of BMSC race checker (common *RaceChecker* class for *BMsc* and *HMsc*) was to compute relation $\ll$.

Base of relation $\ll$ is given by the three conditions in definition of $\ll$. It can be computed directly from visual order $<$ and given $ChannelMapper$.

To compute transitive closure of $\ll$ we use again algorithm based upon topological sort with complexity $\mathcal{O}(n^2)$ (n is number of events in $BMsc$) from [4]. Test for equivalence of $<$ and $\ll$ takes $\mathcal{O}(n^2)$, therefore, whole complexity of this algorithms is $\mathcal{O}(n^2)$.

Counter Example

User should be able to see possible race in $BMsc$ in context, and therefore, it is necessary to display to him/her whole original $BMsc$ with marked pair of events and their messages which violates the property.

Race in HMSC Checker

There are several works which try to maintain notion of race in HMSC. Authors of [15] introduced definition of problem of race occurrence in HMSC which was proven to be undecidable. Authors of [14] provided seemingly similar notion of trace race which was proven to be decidable to check whether HMSC contains trace race. For details about exact difference between these two notions, please, refer to [14].

For obvious reason we will follow the second one concept – trace race. To avoid repetition of notions introduced in the paper we will discuss only implementation related features of this checking algorithm. Details about background of this algorithm are left upon reader and his interest in theoretical results introduced in the paper.

Implementation

Algorithm described by authors of paper [14] works with BMSC-graph, therefore, it was necessary to transform $HMsc$ into $BMsc$-graph. This task is performed by $BMscGraphDuplicator$ and $BMscDuplicator$. This duplicator transforms $HMsc$ into $BMsc$-graph in such way that there isn't lost any information about original structure of $HMsc$ – for this purpose attribute $original$ is used. This is very useful in case we want to reconstruct path in original $HMsc$ from path in transformed $BMsc$-graph.

Informally said $h \in HMsc$ is transformed into $h' \in HMsc$ which contains duplicates of start node and end nodes of h. If h contains reference node which references $BMsc$, then the node is duplicated and its $BMsc$ too in natural way into h'. If h contains reference node which references any other *HMsc* the reference node is transformed into connection node which is connected to successors of start node of the referenced *HMsc*. End nodes of the referenced *HMsc* are transformed into connection nodes and are connected to successors of the original reference node. This type of construction is hierarchical of course.

We are aware of this approach to be memory consuming but it seems to be more effective than handle many identifiers of each nodes depending on current path to reference node which its *HMsc* is referenced from. Moreover, the implementation of this approach is much more clear and understandable.

It was necessary to represent $MaxP$, $MinP$ and footprint too. While trying to find efficient implementation of footprint$f$, we have found that it is not necessary to keep relation $\ll_f$ but it is enough to keep for each event e from footprint f set of instances which contain any greater event with respect to relation $\ll$. This can be held as simple bitfield and simplifies time and space complexity of algorithm as well as maintaining itself.

Footprint is represented by class *Footprint* which is descendant of general *ExtremeEvents*. *ExtremeEvents* holds maximal/minimal events (implemented by *EventDependentInstances*) indexed by precomputed identifiers of instances. It can be confusing that the *ExtremeEvents* holds maximal or minimal events but the reason is simple. This class serves as representation of $MaxP$ and $MinP$, therefore, maximal or minimal depends on chosen semantics which it is used for. *Footprint* differs from *ExtremeEvents* by its augmented members. Because it is necessary to keep path which led to current footprint, *Footprint* contains previous footprint and path from the previous which created the new one.

To traverse *BMsc*-graph special kind of traverser – *FootprintTraverser* is provided. This traverser is derived from *ReferenceNodeFinder* which traverses *HMsc* from given node to the nearest reference nodes and doesn't continue behind them. Such a way of traversing with appropriate listener can simulate that *HMsc* consists only of reference nodes.

Complexity of this algorithm is discussed in paper [14].

Counter Example

If there is any trace race in *HMsc* violating path in the *HMsc* should be provided to user. This path should contain duplicated *BMsc*'s referenced from the original *HMsc* with highlighted violating events.

4.7 Other Properties

There are other interesting properties which would deserve to be mentioned in this thesis or even to be implemented in SCStudio, e.g. safety ([10]), local choice ([9]) and regularity ([16]). Few of them are surely being to be implemented in future versions of SCStudio and was mentioned in [5].

Unfortunately there isn't enough of space in this thesis to describe all of them but I hope that this thesis will be helpful in future implementation of checking algorithms of these properties. Note that basic concepts and utilities were not designed only for introduced checking algorithms but we had a respect to possible usage in future algorithms.

Chapter 5

Integration with SCStudio

This chapter describes how the checking algorithms are integrated into SC-Studio. We will describe common interface of checking algorithms. Every part of the interface will be discussed with motivation to its implementation. Build process and directory structure of SCStudio will be described too.

5.1 Checking Algorithms' Interface

Checking algorithms are thought to be modules of future SCStudio. Natural architectural requirement for such modules whose implementation is hidden from environment is interface which the module is controlled through and which is common to every module. This will ensure unified way of modules handling.

We have suggested common interface for the checking algorithms. The interface is in form of abstract classes *Checker*, *BMscChecker* for checking *BMsc* and *HMscChecker* for checking *HMsc*. *Checker* interface specifies common methods for all checkers, *BMscChecker* and *HMscChecker* specifies particular methods for their domain.

There is public pure virtual method `cleanup_attributes` specified for *Checker* to clean up dynamic attributes allocated by the checker. This method should be called by *Checker* itself after each checking process but it may be useful to call this method outside from the checker in case some unhandled exception inside checker occurs – otherwise it could lead to memory leaks.

BMscChecker and *HMscChecker* in sequence prescribe these pure virtual methods:

```
BMsc* check(BMsc* bmsc,ChannelMapper* mapper)
```

```
HMsc* check(HMsc* hmsc,ChannelMapper* mapper)
```

Both return counter example in case *BMsc*/*HMsc* does not satisfies property which is checked by subclasses of them or null pointer otherwise.

5.2 Build Process and Directory Structure

Intention of SCStudio is to be as platform independent as possible. Therefore, CMake – cross-platform, open-source build system – is used for build process. CMake is introduced at [1] in the following way: *"CMake is a family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice."*

Source files of SCStudio are organized in the following directory structure:

```
/doc

/src

    /data

    /check
        /liveness
        /order
        /pseudocode
        /race
    /view

/tests
```

The `doc` directory will contain Doxygen documentation (see [2]). The `src` involves implementation of all introduced features. Subdirectory `data` contains definition of the data structures *BMsc*/*HMsc* and their traversers described in Chapter 3. Directory `check` involves all checking algorithms in its subdirectories: `liveness` contains *DeadlockChecker* (described in 4.1) and *LivelockChecker* (4.2), directory `order` contains *FifoChecker* (4.5) and *AcyclicChecker* (4.3), `pseudocode` contains the common auxiliary utilities (e.g. for computation of $<$ or $\ll$) and `race` includes *RaceChecker* and *Footprint* implementation (4.6). The `view` directory contains visualization tools of SCStudio (out of scope of this thesis).

Together with checking algorithms there are also provided basic tests of these algorithms. These tests are situated in directory `tests`. To be conform to test framework of CMake the tests are built into executable files returning non-zero value in case of found error. Inspection of the source files of these tests is recommended way how to get familiarized with $BMsc$ and $HMsc$.

Chapter 6

Conclusion

The main aim of this thesis was to set up basis of future tool – Sequence Chart Studio which will serve for designing and verification of communication systems using Message Sequence Chart. At first it was necessary to study MSC standard introduced by Telecommunication Standardization Sector in Recommendation Z.120.

This recommendation has become sample for the data structures which are able to describe chosen subset of features of MSC. The data structures are capable to supply role of the textual and the graphical syntax of original MSC, moreover, they contain utilities simplifying processing of the structures. The data structure isn't used only for a representation of intended design of system but for a possible error reporting too. The structures are described using UML class diagram. Besides a graphical description of the structure, the UML diagram serves as a base for thinking about $BMsc/HMsc$ as a formal structure.

Processing of $BMsc$ and $HMsc$ by itself is performed by the traversers and their listeners. A concept of the traversers and listeners is capable to be used in many cases and simplifies singnificantly processing of the structures. Moreover, as it turned out, the concept is very flexible and can be used for different kind of traversing than the one shown in this thesis (Depth First Search).

Using the introduced notation over UML diagram there were described several properties of $BMsc$ and $HMsc$. These properties are checked by checking algorithms which fully derived benefit from the provided traversers. The first four of the algorithms check static properties of BMSC or HMSC and can be used as a sample example how to use the developed utilities. The last one of the algorithms is little bit more complex. An origin of this algorithm is in the paper [14].

We hope this thesis will be helpful for future work on Sequence Chart Studio and that the designed data structure and utilities will simplify the work in significant way.

Bibliography

[1] CMake, The cross-platform, open-source build system. `http://www.cmake.org/`.

[2] Doxygen, Source code documentation generator tool. `http://www.stack.nl/~dimitri/doxygen`.

[3] Sequence Chart Studio. Sequence chart drawing and verification tool. `http://scstudio.sourceforge.net/`.

[4] R. Alur, G.J. Holzmann, and D. Peled. An Analyzer for Message Sequence Charts. *Tools and Algorithms for the Construction and Analysis of Systems: Second International Workshop, TACAS'96, Passau, Germany, March 27-29, 1996: Proceedings*, 1996.

[5] J. Babica, V. Řehák, P. Slovák, P. Troubil, and M. Zavadil. Formalisms and Tools for Design and Specification of Network Protocols, 2007.

[6] H. Ben-Abdallah, S. Leue, University of Waterloo, Dept. of Electrical, and Computer Engineering. *Syntactic Analysis of Message Sequence Chart Specifications*. Dept. of Electrical and Computer Engineering, University of Waterloo, 1996.

[7] T.T. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. MIT Press Cambridge, MA, USA, 1990.

[8] Department of Information Technology at Uppsala University and Department of Computer Science at Aalborg University. UPPAAL. `http://www.uppaal.com/`.

[9] B. Genest and A. Muscholl. The Structure of Local-Choice in High-Level Message Sequence Charts (HMSC). 2002.

[10] E.L. Gunter, A. Muscholl, and D. Peled. Compositional message sequence charts. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(1):78–89, 2003.

[11] O. Haugen. Comparing UML 2.0 Interactions and MSC-2000. *Proceedings of SAM 2004: SDL and MSC fourth International Workshop*.

[12] R.Z. ITU-T and Z. Recommendation. 100: Specification and Description Language (SDL). *International Telecommunication Union*, 2000.

[13] ITU Telecommunication Standardization Sector - Study group 17. ITU recommendation Z.120, Message Sequence Charts (MSC), 2004.

[14] Jan Fousek, Vojtěch Řehák, Petr Slovák, and Jan Strejček. Decidable race condition in high-level message sequence charts. 2008.

[15] A. Muscholl and D. Peled. Message sequence graphs and decision problems on Mazurkiewicz traces. *Proc. of MFCS*, 99:81–91, 1999.

[16] A. Muscholl, D. Peled, and Z. Su. Deciding Properties for Message Sequence Charts. *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, pages 226–242, 1998.