

MASARYK UNIVERSITY  
FACULTY OF INFORMATICS



# Refactoring of Sequence Chart Studio

DIPLOMA THESIS

**Ondřej Bouda**

Brno, 2013



## **Declaration**

Hereby I declare, that this thesis 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.

Brno, May 27, 2013  
Ondřej Bouda

**Advisor:** RNDr. Vojtěch Řehák, Ph.D.



## **Acknowledgement**

I would like to express many thanks to my advisor, RNDr. Vojtěch Řehák, Ph.D., for his guidance throughout this work.



## Abstract

Sequence Chart Studio is a tool for formal modelling and verification of communication protocols specified in Message Sequence Chart. It features a user-friendly graphical user interface and algorithms for checking properties, computing differences between specification and actual flows, rearranging diagrams, and several other functions. All parts have been implemented as students' work.

The tool was initially designed to work with messages as the only type of events on communicating instances, and did not allow to reasonably extend it to support other types of events. The purpose of this work was to refactor the application core to enable further extensions. As a consequence, all the algorithms and other dependent code was to be adjusted without affecting their functionality. A sequence of gradual steps was performed, resulting in the core refactored successfully.

The refactoring was proven useful by actually extending Sequence Chart Studio with two new types of events: local actions and conditions. Although the tool pays attention to comply with the Message Sequence Chart recommendation, in case of conditions, the formalism was found to be rather imperfect. An extended syntax and alternative semantics were proposed for conditions. Within the work, application core code was reviewed, the object model was slightly simplified, and memory problems were greatly reduced.





## **Keywords**

Sequence Chart Studio, Message Sequence Chart, refactoring, conditions, local actions, formal methods.



## Contents

1	<b>Introduction</b>	3
1.1	<i>Sequence Chart Studio</i>	3
1.2	<i>Message Sequence Chart</i>	4
1.3	<i>The Goal of the Work</i>	5
2	<b>MSC Elements</b>	7
2.1	<i>Elements Supported in SCStudio</i>	7
2.1.1	Basic MSC	8
2.1.2	High-Level MSC	9
2.1.3	Time Restrictions	10
2.1.4	Data Languages	10
2.2	<i>Local Actions</i>	10
2.3	<i>Conditions</i>	11
2.3.1	MSC Recommendation Definition	11
2.3.2	Inconveniences of the Standard Definition	13
2.3.3	Alternative Conditions Proposal	14
3	<b>Conditions Evaluation</b>	27
3.1	<i>Restrictions over the Conditions Specification</i>	27
3.2	<i>SCStudio Algorithms for MSC Traversal</i>	27
3.3	<i>Traverser Algorithm</i>	28
3.3.1	Forward Traverser Algorithm	28
3.3.2	Backward Traverser Algorithm	30
4	<b>Internal Structure Refactoring</b>	37
4.1	<i>Initial State</i>	37
4.2	<i>Enhancement Proposals</i>	39
4.3	<i>Refactored State</i>	40
5	<b>Implementation</b>	43
5.1	<i>Local Actions</i>	43
5.2	<i>Conditions</i>	43
6	<b>Conclusions</b>	47
	<b>Bibliography</b>	49



## Chapter 1

### Introduction

The general topic addressed by this work is prevention of defects in the software development process. The motivation is fairly trivial: the less defects, the higher quality and, above all, the less money spent on development and maintenance.

Defects may arise at various stages of the development process, with various importance, however. There is a consensus that the cost of defect removal depends on the stage the defect is introduced in. E.g., [12] says: “The longer the defect stays in the software food chain, the more damage it causes further down the chain,” and concludes that early defects are more expensive. Hence, we mostly focus on the design phase.

This work specializes on the area of computer network protocols development. Particularly in this field, it is crucial to detect design shortcomings yet before the protocol gets implemented and programmed into network devices. As mentioned by [3], lots of companies use only structured English text for protocol specification, though. Compared to formal specification, usage of natural language inherently yields more flaws, as it is usually ambiguous and vague. Nonetheless, the mentioned companies regard application of formal methods as a too expensive and time-consuming process. That was the basis for creating and further development of Sequence Chart Studio.

#### 1.1 Sequence Chart Studio

Sequence Chart Studio [21] (or SCStudio, for short) is a tool for formal modelling and verification of communication protocols. It tries to remedy the aforementioned situation by helping with gradual adoption of formal methods.

First, it features a user-friendly graphical interface in which the communication gets visualized so that the diagrams may simplify and clarify the textual specification. Since the SCStudio frontend is implemented as a Microsoft Visio add-on, it not only utilizes the features of Visio, but it integrates into the whole Microsoft Office suite. On top of that, SCStudio enriches Visio with a custom set of features making communication diagrams drawing easier.

Second, if the diagrams are drawn properly, checker algorithms may formally verify that some properties are satisfied, or give a counterexample. A “proper” diagram is that which adheres to the syntactical rules required by the formalism SCStudio uses.

Third, when the modelled system gets implemented, timestamped logs of the actual communication may be imported into SCStudio, visualized, and checked whether the

logged run of the system follows the specification of the protocol. Messages violating the specification get highlighted and confronted with the expected behaviour, so it is easy to identify a problem.

There are other notable features SCStudio offers, some of which get described later in the following chapters. For a complete list, see the SCStudio documentation [1].

The formalism used by SCStudio for description of communication is Message Sequence Chart [9], referred to as MSC. It has been chosen for being powerful enough yet intuitively readable, the latter of which is a prime requirement for any specification, as emphasized, e.g., in [13].

### 1.2 Message Sequence Chart

This section briefly introduces the main features of the formalism, whereas Chapter 2 describes the MSC elements relevant for this work in more detail. For more precise introduction to MSC, see Section 1 of the ITU-T Z.120 recommendation [9].

MSC is a high-level language for specifying asynchronous message-based communication systems. The standard specifies two forms, graphical and textual, the former of which is designated for humans. Since it is visual, it gives overview of the modelled system, is easy to understand, and as such will be preferred throughout the thesis. “The textual form of MSC is mainly intended for exchange between tools and as a base for automatic formal analysis.” [9]

Let us see an example MSC diagram in Figure 1.1. The diagram describes behaviour of three communicating parties in the system: a client, a proxy and a server. The client sends a *register* message to the proxy, which resends it to the server and confirms back to the client. After receiving the confirmation, the client sends a *request* message to the proxy, which then sends a *response*.

Any such diagram—specifying the communicating parties, messages sent among them and other types of events—is referred to as *basic MSC* (BMSC). The MSC standard also specifies another type of diagrams called *high-level MSC* (HMSC), which allow for hierarchical specification of the modelled system. Using HMSC, one can decompose the system into simpler parts.

“Intuitively, HMSC can be seen as a finite automaton where every state is labelled by a BMSC<sup>1</sup>. Each run of the automaton represents a BMSC which is the concatenation of the BMSCs along the run. Hence, an HMSC specifies the set of BMSCs represented by all the (accepting) runs.” [3]

A sample HMSC diagram follows in Figure 1.2. The modelled system starts with the client registration, which is specified in the Registration MSC diagram. The specification follows with either Redirection diagram, after which the system is done, or Data Transfer diagram, which may be performed repeatedly. The data-transfer branch ends

---

1. Or HMSC—an HMSC diagram may refer to another HMSC diagram.

with the Checksum Verification. Note that it is *not* guaranteed that events from one referenced diagram may only get executed when events from the previous diagram are finished—the HMSC really expresses the concatenation of BMSCs. Chapter 2.1.2 mentions more details.

### 1.3 The Goal of the Work

At the beginning of this work, there was a simple requirement: to extend Sequence Chart Studio with two new types of objects that can be modelled—*local actions* (an event representing an internal action of the instance it is attached to) and *conditions* (restricting some behaviour of the modelled system). Although the latter is a non-trivial extension, which touches all the algorithms traversing the diagrams, the main issue was that the design of the data structure used for storing the objects did not allow that (Chapter 4.1 describes the problem in detail). Hence the need for reworking, or *refactoring* the application so that it could be extended as required.

The next chapter lists the MSC elements already supported by SCStudio and describes in detail the types of elements to be added: local actions and conditions. Chapter 3 proposes the algorithm for evaluating the semantics of conditions. The main part, Chapter 4, elaborates on refactoring SCStudio to make implementation of the new elements possible. The implementation itself is documented in Chapter 5. The last chapter evaluates the accomplished work and proposes some further extensions.

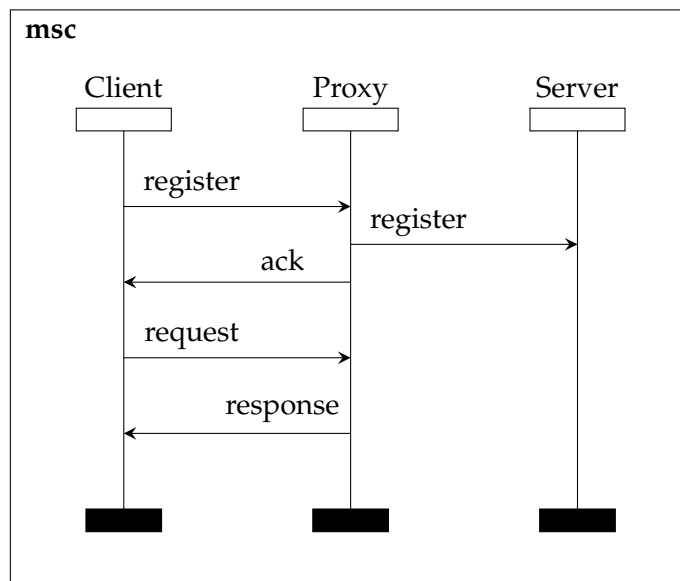


Figure 1.1: A BMSC example.

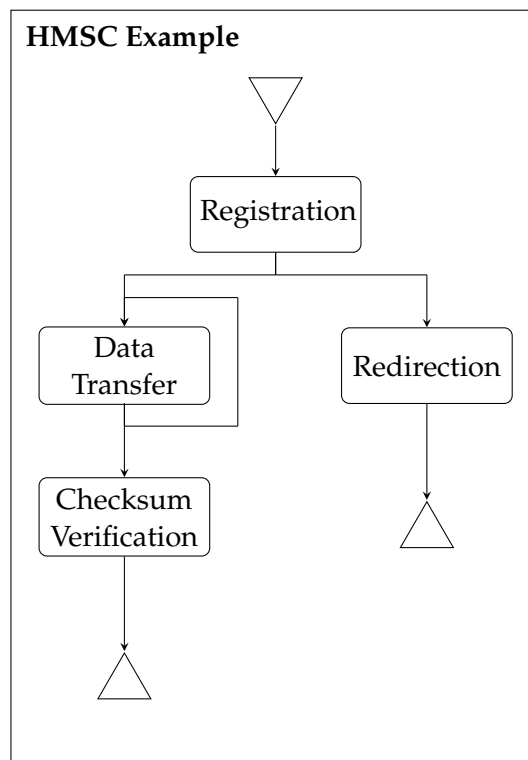


Figure 1.2: An HMSC example.



## Chapter 2

### MSC Elements

This chapter briefly describes part of the MSC recommendation relevant to this work and support of the elements in SCStudio, with the focus on local actions and conditions. A complete specification of these two types of elements is given in Sections 2.2 and 2.3.

#### 2.1 Elements Supported in SCStudio

SCStudio supports most of the elements defined in the MSC recommendation [9]. It is necessary to distinguish between support of the elements on the frontend and backend level, though. The former allows the user to draw diagrams using some elements, but the backend support enables the algorithms to actually take the elements into account when verifying, redrawing, exporting or any other algorithmic processing of a diagram.

On the frontend, all the elements relevant to this work are already supported and will be only slightly adjusted to meet the requirements. As such, frontend-level elements support will not be further discussed.

The SCStudio backend consists of:

- the *internal data structure* for representing the MSC diagrams,
- verification algorithms (e.g., race-condition checker or time-race checker),
- transformers (e.g., the *beautify* transformer able to automatically arrange the elements in the diagram, or the *tighten time* transformer—more on this in [19]),
- other algorithms (e.g., the *find flow* function able to compare MSC diagrams and show differences between them),
- import/export algorithms.

The internal data structure is in the centre of the application. All other parts depend on it, thus, with a change in the data structure, all other backend parts must be altered appropriately. As examined later, it is the data structure which will have to be extended, with all the implications regarding the other parts, for SCStudio to support local actions and conditions. There is also some “glue” code for transforming the elements between frontend and backend, which will get modified, too.

The rest of this section describes the MSC elements already supported by the SCStudio backend. Basically, it is just a brief summary of other students' work, namely [2] and further extensions [11], [19], [14], [18], [4], and [20].

### 2.1.1 Basic MSC

A BMSC diagram contains a set of *instances*, representing the communicating parties within the modelled system. An instance is depicted as a vertical line on which *events* are attached. There might be various types of events described later in this subsection. Not all instances of the system must be present in every BMSC diagram—those not mentioned are considered empty, having no events.

The events on an instance are totally ordered by their placement on the instance line from top to bottom, i.e., event  $e$  is specified to occur sooner than event  $f$  iff  $e$  is above  $f$  on the instance line. The MSC recommendation defines special instance segments, though, called *coregions* [9, Section 7.1], depicted as rectangles with dashed sides. Events within a coregion, which are attached to the rectangle sides, are generally unordered. The user may specify explicit order of any two events in the coregion by drawing an *ordering line*<sup>1</sup>. An instance may contain several areas of either form. The order within an area is given by whether it is a *strict-order* (line) area or coregion area, while two events from different areas are ordered by the relative order of the areas on the instance.

There are only two types of events supported by the SCStudio backend so far—a *message sending event* and *message receiving event*, which denote sending or receiving a *message*, respectively. There are three kinds of messages defined by the MSC recommendation, all of them supported:

**complete message** — sent and received by concrete (distinct) instances, delivered in any finite time (recall the communication model in MSC is asynchronous);

**lost message** — sent by a concrete instance, but not delivered anywhere;

**found message** — a message which “appears from nowhere” [9, p. 24], received by a concrete instance.

A complete message is a special type of element in MSC in that it adds further restrictions on order of events: a message sending event precedes the corresponding message receiving event. The (partial) order of events in a BMSC is thus defined as the reflexive and transitive closure of union of order given by instances and messages. This order is called *the visual order* of the given BMSC.<sup>2</sup>

Figure 2.1 demonstrates all the BMSC concepts supported by the SCStudio backend. There is an extra type of element not mentioned yet—a *comment*, which may be bound to any event. Comments only serve documentation purposes, and are ignored by the algorithms.


---

1. Of course, to form a valid order, the reflexive and transitive closure is considered.

2. See Definition 4.7 in [2] for a more formal definition.

### 2.1.2 High-Level MSC

For high-level specification of the modelled system, HMSC diagrams are used, which relate particular BMSCs and even other HMSCs together, governing the behaviour of the (sub)system. Essentially, an HMSC diagram consists of:


- *start node*, representing the (sub)system starting point, depicted as  $\nabla$ ;
- *MSC reference nodes*, referring to other MSC diagrams, depicted as ;
- *end nodes*, denoting exit points of the described (sub)system, depicted as  $\triangle$ ;
- *flow lines*, connecting the nodes, depicted as either  $\text{—}$  (line) or  $\text{—}\rightarrow$ .

Following the introduction of HMSCs as finite automata in Section 1.2, the semantics of HMSC elements may be summed up to the following:

- the automaton states are represented by start node, reference nodes and end nodes;
- the initial state is denoted by start node;
- the final states are denoted by end nodes;
- the transitions are given by flow lines.

It is important to note that reference nodes do not work as synchronization barriers. The semantics of HMSC is *not* such that all the events would have to be executed within the automaton state, and only after that the next state could be entered. The instances do not get synchronized before or after a reference node. Instead, some events may get executed before other events, regardless of whether they are specified in a “preceding” or “following” referenced diagrams—as illustrated in Figure 2.2 and an equivalent MSC in Figure 2.3.

The SCStudio backend supports all the HMSC elements listed above. Multiple flow lines may be connected to a single point of a node. To simplify the drawing, *connection points* (depicted as  $\circ$ ) may be used in HMSC diagrams, which serve as auxiliary points for connecting multiple flow lines anywhere in the drawing. They are also supported by the SCStudio backend. Comments are supported in HMSC, too. They have the same semantics as in BMSC—they serve documentation purposes only, and are ignored by algorithms.

The remaining type of element SCStudio backend supports in HMSC is *condition node* (depicted as ). Following the introduction of HMSC semantics, conditions serve as transition guards in the corresponding finite automaton. Compared to the MSC recommendation, however, the support is very limited in SCStudio: condition nodes are stored in the internal data structure, but ignored by the algorithms (with one exception, mentioned in Section 2.3).

A summary example follows in Figure 2.4.

### 2.1.3 Time Restrictions

Besides the basic MSC elements described in the previous two subsections, SCStudio supports time restrictions (both on frontend and backend), as specified in [9, Section 11]. The purpose of the extension is to provide means to quantify time in the protocol specification instead of the implicit *any finite time*.

There may be a time interval, or a set of intervals, assigned to any pair of events or HMSC node connection points. The SCStudio documentation [1, Section Tighten Time] contains some nice examples on this. The time extension also allows for specifying timestamps for events. This is particularly useful for analysing real traffic of the implemented protocol.

### 2.1.4 Data Languages

Despite the elements described in the previous subsections, for further reference, one concept not actually supported by SCStudio has to be mentioned—the concept of data languages. Even though MSC describes communication, the actual communication data have not been mentioned yet. To stay independent of particular application, the MSC recommendation leaves a concrete data language as a parameter, and only enforces some essential requirements on it. E.g., SDL [8] or C may be used for data specification.

Data expressions may be used as contents of messages, parameters of timers, attributes of instances, etc. Unfortunately, data concepts are not supported in SCStudio.

## 2.2 Local Actions

The MSC recommendation describes *actions* as follows:

“In addition to message exchange the actions may be specified in MSCs. An action is an atomic event that can have either informal text associated with it or formal data statements. Informal text is enclosed in single quotes to distinguish it from data statements.” [9, Section 4.9]

There are two notes worth mentioning. First, an action is an event, thus, it is only relevant to BMSC diagrams. Second, the “formal data statements” refers to the concept of incorporating data in MSC—the statement should be applied to the internal state maintained with the instance during evaluation of the MSC diagram.

SCStudio frontend already supports actions—it is possible to draw diagrams with them. Regarding the backend, support for actions has to be added—a new type of event has to be defined in the internal data structure and algorithms should be altered to handle such an event. Since SCStudio has no notion of instance internal state and by no means supports the data languages, only informal text is to be considered as the action contents. Thus, an action event has the same semantics as a comment for the purpose of this work, therefore, it is to be ignored by the algorithms.

## 2.3 Conditions

*Conditions* are the other new element type to be added to SCStudio. The frontend already supports them *somehow*—it defines the condition shapes both in BMSC and HMSC, and enables the user to enter custom text in them. Their semantics, as defined by the MSC recommendation, is not implemented, though. Only HMSC conditions are present in the internal data structure and they are completely ignored by algorithms.<sup>3</sup>

In this section, the conditions get specified as defined by the MSC recommendation, the definition is confronted with some practical requirements, and the semantics is proposed for extending SCStudio.

### 2.3.1 MSC Recommendation Definition

The MSC recommendation introduces conditions as follows:

“Conditions can be used to restrict the traces<sup>4</sup> that an MSC can take, or the way in which they are composed into High-Level MSCs. There are two types of condition: *setting* and *guarding* conditions. Setting conditions set or describe the current global system state (global condition), or some non-global state (nonglobal condition). In the latter case the condition may be local, i.e., attached to just one instance.

Guarding conditions restrict the behaviour of an MSC by only allowing the execution of events in a certain part of the MSC depending on their values. This part, which is the *scope* of the condition, is either the complete MSC or the innermost operand of an inline expression<sup>5</sup> or a branch of an HMSC. The guard must be placed at the beginning of this scope. The condition is either a (global or nonglobal) state, which the system should be in (as defined by prior setting conditions), or a Boolean expression in the data language.”

The graphical syntax is presented in Figure 2.5: on the left side, a setting condition is shown, which sets condition variables *X*, *Y*, and *Z*; on the right, there is a guarding condition checking for status of condition variables *X* and *Y*.

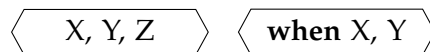


Figure 2.5: An example of syntax of setting and guarding condition, respectively.

3. The only algorithm considering the contents of condition nodes is the Monte Carlo simulation, which takes percentage values from them and traverses the diagram using random-walk weighted by these values. Unfortunately, the algorithm is documented in no printed source, only a brief tutorial is given in [1, Section Monte Carlo Simulation].

4. A *trace* is a linearization of all events in the diagram, respecting the event partial order. In other words, if one timestamped each event in a run of the system, the events ordered by the timestamps would give a trace.

5. Inline expressions allow for specifying alternative branches, loops, exception handling and other structural constructs within a BMSC, as defined in [9, Section 7.2]—which SCStudio does not support, though.

A conditions usage example follows in Figure 2.6, which specifies a connection loop. Due to the first guarding condition, the whole diagram may only be executed if condition variable *Disconnected* has previously been set. A connection request is made with one of two possible outcomes: either the connection is established and the condition variable *Connected* is set to indicate that, or the connection fails. In the latter case, two condition variables are set: *Disconnected* and *Failed*. Due to this, when the connection fails repeatedly, the system may hang up and quit. Further examples may be found in the MSC recommendation Sections 9 and 17.1.

According to the recommendation, conditions have several characteristics:

**type** — whether it is a setting or a guarding condition; there is a special type of guarding condition: *otherwise*;

**content** — what is the content of the condition: either a set of *condition variable* names (those conditions are referred to as *system-state conditions* in the rest of the text), or a boolean expression in the data language, with the exception of the *otherwise* condition, which has no content; formal syntax follows later in this subsection;

**range** — set of instances the condition affects;

**scope** — within what scope of the diagram the condition has effect (see the definition in the quotation above).

### Conditions Placement

The placement of conditions is restricted by the following rules in [9, Section 4.7]:

- “If instance *a* and instance *b* share the same <condition> then for each message exchanged between these instances, the <message input> and <message output> must be placed either both before or both after the <condition>.”
- “A guarding condition must be placed at the beginning of its scope, where a scope is either a whole MSC, an operand of an inline expression, or a branch of an HMSC.”
- “A guarding condition should always cover all ready instances of its scope, where an instance is ready if it contains an event that may be executed before any other event in the scope.”
- “*Otherwise* can only occur as the guard of exactly one operand of an alternative expression.”

There are some more, mainly syntax-oriented restrictions in the recommendation, which are not that relevant to be listed here, though.

## Conditions Range

Each condition specifies the set of instances it affects (the condition is *shared* among these instances). Any non-empty set of instances from the whole modelled system may be used. Even instances not present in a BMSC diagram may share a condition. Speaking about syntax, this is achieved by the *shared* keyword, following a list of instance names, both in textual and graphical form. There is a special *shared all* syntax for referring to all instances from the whole system, specifying a *global* condition.

## Conditions Semantics

As the MSC recommendation says, “setting conditions define the actual system state of the instance(s) that share the condition.” A single setting condition may list multiple condition variables which are set to *true* by that condition.

Regarding the guarding conditions: “The events in the scope guarded by a guarding condition can only be executed if the guarding condition is true at the time the first such event is executed. [...] If the guard is a set of condition names, the *last* setting condition on that set of instances must have a non-empty intersection with the guard. Only setting conditions on exactly the same instances are checked; conditions that set the state of a subset or superset of these instances do not. [...] The *otherwise* guard is true if guards of all other operands of the alternative expression are false.” [9]

According to the quoted paragraphs, the traces with unsatisfied guarding condition are excluded from the MSC. A system-state guarding condition is satisfied (evaluated to *true*) iff at least one of the listed condition variables is set in the last setting condition on the exactly same set of instances.

### 2.3.2 Inconveniences of the Standard Definition

Prior to implementing the conditions in SCStudio, let us first see some inconveniences the recommendation proposes. The problematic part, regarding the system-state conditions, is that when evaluating a guard, just the last setting condition sharing the same set of instances is considered.

Let us assume the following use case: first, a client connects to a server and initializes some settings for the session. Then, the client proceeds, however, in the middle of the session, the client decides to change one of the settings. Later on, some behavior of the system depends on the configuration set up by the client.

When modelling the system, guarding conditions shall be used to cut off the behavior which the system should not exhibit due to the client’s configuration (e.g., the system shall not send the client a notification if the client did not enable notifications). When the guarding condition refers only to the last setting condition, that setting condition overrides any previous settings—it may not only add some state. Thus, to represent the client changing a setting in the middle of the session, the according setting condition must repeat all the system states that were set previously—which is possible

to express using the standard conditions semantics, with several problems, though:

- preserving some part of the system state must be handled by a diagram of size exponential to the number of preceding setting conditions which set the condition variables constituting the system state;
- guarding conditions cannot specify conjunction of several condition variables—together with the restriction that a guard may only be present at the beginning of its scope, it implies that to make a branch executable only when two particular condition variables are set, a new condition variable must be used, representing the conjunction;
- any further modification (i.e., addition) of a system state results in major changes in the model.

See Figure 2.7 for an example in which just two condition variables shall be preserved and another one added.

Besides the problem with preserving part of the system state and representing a combination of condition variables, there is another (although much less serious) inconvenience—possible usage of *otherwise* conditions. The MSC recommendation permits an *otherwise* condition only in alternative expressions (as quoted in Chapter 2.3.1). It could be convenient to allow *otherwise* conditions also in HMSC branches.

To sum up, conditions are quite limited in the MSC recommendation, and are useful mainly for describing the system state using only a single condition variable at a time. Any combinations of several condition variables are overcomplicated, as demonstrated by Figure 2.7, which is due to limitation to only last setting condition and only to disjunctions of several condition variables.

One could argue that data language expressions are supported by the MSC recommendation, which were completely omitted by the aforementioned critics. These are quite different, though, since only local actions may set a state which may be used by a subsequent data-language-expression guarding condition—and local actions only define the state for a single instance, not a set of instances like setting conditions. Hence, we have a tool which is more powerful at guarding a scope, but less powerful in defining the state. Although both kinds of conditions could be combined together.

The following subsection proposes an alternative syntax and semantics for MSC conditions, which is a conservative extension in the sense it does not enhance MSC expressability, but rather simplifies its usage. The proposed conditions will then be implemented in SCStudio.

### 2.3.3 Alternative Conditions Proposal

With the background of the previous subsection, alternative conditions are proposed for MSC. Even those aspects that will not get implemented by this work are proposed to give a complete specification so that SCStudio might be extended in the future.



The base of the conditions specification remains the same as defined in the MSC recommendation. The alternative proposal differs (only) in:

- a new type of *documentation* conditions;
- a new type of *unsetting* conditions;
- extended syntax and different semantics of system-state guarding conditions;
- extended *otherwise* conditions placement.

First, the extended syntax is presented in the following text, the semantics is then specified for the according types of conditions.

### Extended Syntax

The conditions syntax is extended over the MSC recommendation [9, p. 42] as follows:

```

1  <condition text> ::=
2      <condition name list>
3      |   unset <condition name list>
4      |   when {<condition name list> |
5              <left open> <expression> <right open> |
6              <condition name expr>}
7      |   otherwise
8      |   "<any text without quotes>"
9      |   when "<any text without quotes>"
10
11 <condition name list> ::=
12     <condition name> {, <condition name>}*
13
14 <condition name expr> ::=
15     <condition name term>
16     | <condition name expr> {OR | <bool or>} <condition name term>
17
18 <condition name term> ::=
19     <condition name factor>
20     | <condition name term> {AND | &&} <condition name factor>
21
22 <condition name factor> ::=
23     [{NOT | !}] <condition name primary>
24
25 <condition name primary> ::=
26     <condition name>
27     | <left open> <condition name expr> <right open>
28
29 <left open> ::= (
30
31 <right open> ::= )
32
33 <bool or> ::= ||

```

where <condition name> refers to name of a condition variable, <expression> is an expression in a data language.

### Documentation Conditions

The *documentation conditions* serve for free-form documentation purposes<sup>6</sup>. A system state is described by the documentation condition using natural language. The documentation condition might be viewed as a mere comment using the conditions syntax. Both setting and guarding documentation condition may be specified, using the syntax at lines 8 and 9 in the syntax listing above. The semantics is the same as with comments—the documentation conditions, both setting and guarding, are ignored by algorithms.

### Unsetting Conditions

The purpose of unsetting conditions is explained together with semantics of system-state guarding conditions proposed below. Both the graphical and textual syntax is similar to setting conditions, with the difference that the keyword **unset** is used in front of `<condition name list>`, as denoted at line 3 in the syntax listing above. The restrictions on using unsetting conditions are the same as for setting conditions.

### System-State Guarding Conditions

Both the graphical and textual syntax of system-state guarding conditions is extended as specified at line 6 in the syntax listing above. Besides condition variable name lists, it is possible to specify expressions on condition variables.

The semantics of guarding conditions differs from the MSC recommendation. The events in the scope guarded by a guarding condition can still only be executed if the guarding condition is true at the time the first such event is executed. If the guard is a set of condition variable names or an expression on condition variable, however, it is not evaluated according to the last setting condition on that set of instances, but rather according to the *current system state* defined on that set of instances. The notion of system states is introduced together with the *condition effect* of HMSC nodes on the system state, which actually defines operational semantics of HMSC nodes. Guarding conditions may then be evaluated based on the system states.

**Definition 1.** Let **CondVar** be the set of all condition variables. Any  $\sigma \subseteq 2^{\text{CondVar}}$  is called a system state. The set of all system states is denoted by  $\Sigma$ .

A system state expresses the set of condition variables which have been set. On the system start, no condition variables are set, thus, the initial state is empty. With each HMSC node  $n$  passed in the system run, the system state may be changed: one or more condition variables may be set using a setting condition, or unset by an unsetting condition. Both may be present in either BMSC or HMSC. To define that formally, the condition effect of a node is defined on HMSC, which specifies transformation of the system state caused by the HMSC node.

---

6. Free-form conditions were suggested within SCStudio feature requests by the project partners.

**Definition 2.** Let  $E_b$  denote the set of events in BMSC  $b$ ,  $S_b, U_b \subseteq E_b$  denote the setting, or unsetting condition events in  $b$ , respectively. Let  $<_b$  be the visual order on  $E_b$ . Let  $V(c)$  denote the set of condition variables listed by (BMSC or HMSC) condition  $c$ .

Let  $N$  denote the set of HMSC nodes. The condition effect is defined as relation  $\rightarrow \subseteq \Sigma \times N \times \Sigma$  which contains the following elements, written as  $\sigma \xrightarrow{n} \sigma'$ :

- $\sigma \xrightarrow{n} \sigma$  for any  $n$  except reference node and setting or unsetting condition node;
- $\sigma \xrightarrow{n} (\sigma \cup V(n))$  if  $n$  is a setting condition node;
- $\sigma \xrightarrow{n} (\sigma \setminus V(n))$  if  $n$  is an unsetting condition node;
- $\sigma \xrightarrow{n} (\sigma \cup X \setminus Y)$  if  $n$  is a BMSC reference node, where  $X$  and  $Y$  are sets of condition variables which have been set (or unset, respectively) by the referenced BMSC, i.e.:

$$\begin{aligned} X &= \{C \in \mathbf{CondVar} \mid \exists e \in S_b . C \in V(e) \wedge (\nexists f \in U_b . e <_b f \wedge C \in V(f))\} \\ Y &= \{C \in \mathbf{CondVar} \mid \exists e \in U_b . C \in V(e) \wedge (\nexists f \in S_b . e <_b f \wedge C \in V(f))\} \end{aligned}$$

- $\sigma \xrightarrow{n} \sigma'$  if  $n$  is an HMSC reference node,  $n_1, n_2, \dots, n_k$  is the part of the system run executed within the referenced HMSC, and  $\sigma \xrightarrow{n_1} \sigma_1 \xrightarrow{n_2} \dots \xrightarrow{n_k} \sigma'$ .

Now, in a system run  $n_1, n_2, \dots$ , the system state for any executed HMSC node  $n_i$  is  $\sigma_i$ , where  $\sigma_1 \xrightarrow{n_1} \sigma_2 \xrightarrow{n_2} \dots \xrightarrow{n_{i-1}} \sigma_i$ ,  $\sigma_1 = \emptyset$ . Therefore, every guarding condition node within a system run has a defined system state on which it may be evaluated. The following definition gives the formal semantics of guarding conditions.

**Definition 3.** On the set  $\Gamma$  of all condition expressions conforming to the `<condition name expr>` syntax, and the set  $\Sigma$  of all system states, the condition valuation  $\nu : \Gamma \times \Sigma \rightarrow \{\text{true}, \text{false}\}$  is inductively defined as follows:

- $\nu(\sigma, C) = \begin{cases} \text{true} & \text{if } C \in \sigma \\ \text{false} & \text{otherwise} \end{cases}$
- $\nu(\sigma, \text{NOT } \phi) = \nu(\sigma, !\phi) = \begin{cases} \text{true} & \text{if } \nu(\sigma, \phi) = \text{false} \\ \text{false} & \text{otherwise} \end{cases}$
- $\nu(\sigma, \phi_1 \text{ AND } \phi_2) = \nu(\sigma, \phi_1 \ \&\& \ \phi_2) = \begin{cases} \text{true} & \text{if } \nu(\sigma, \phi_1) = \text{true} \text{ and } \nu(\sigma, \phi_2) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$
- $\nu(\sigma, \phi_1 \text{ OR } \phi_2) = \nu(\sigma, \phi_1 \ || \ \phi_2) = \begin{cases} \text{true} & \text{if } \nu(\sigma, \phi_1) = \text{true} \text{ or } \nu(\sigma, \phi_2) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$

A guarding condition is in system state  $\sigma$  evaluated to true if:

- it contains a condition name expression  $\phi$  and  $\nu(\sigma, \phi) = \text{true}$ , or
- it contains a condition name list  $C_1, \dots, C_n$  and  $\nu(\sigma, C_1 \text{ OR } \dots \text{ OR } C_n) = \text{true}$ .

Note the evaluation of a condition name list is consistent with the MSC recommendation in that any non-empty intersection with a prior setting condition satisfies the guard. Explicit disjunction expressions shall be preferred, though, to prevent any confusion.

### Extended *Otherwise* Conditions Placement

Besides the possible placement of *otherwise* conditions specified by the MSC recommendation (which is only in alternative expressions, which are not supported by SC-Studio), an *otherwise* condition may also guard an HMSC branch. Before we proceed, let us define HMSC branches first, as they are not defined by the MSC recommendation.

**Definition 4.** *A branch head is such an HMSC node which is either the start node or has more than one outgoing flow (connection) line. A branch is an HMSC path starting at a branch head. Two branches are alternative if they have the same head. Branch  $B$  is said to be guarded by condition  $c$  (or  $c$ -guarded) if  $c$  is the first node on  $B$  following the head of  $B$ .*

The restrictions and semantics of the *otherwise* conditions in HMSC are similar as specified by the MSC recommendation for expressions: an *otherwise* condition can only occur as the guard of exactly one of alternative branches. The *otherwise* condition evaluates to *true* iff all branches alternative to the *otherwise*-guarded branch are guarded by a condition evaluated to *false*.

To demonstrate the benefits of the alternative conditions proposal, the use case from Section 2.3.2 is redrawn in Figure 2.8.



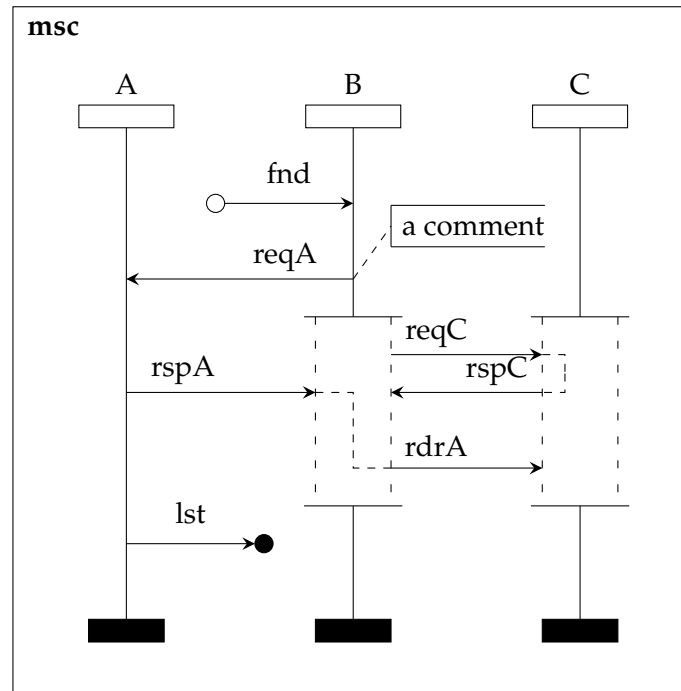


Figure 2.1: First, instance *B* finds message *fnd*. Then, it sends *reqA* to *A*. Independently of time of *reqA* reception by *A*, *B* sends *reqC* to *C*. After either of the instances receives the request message, it sends a response back to *B*. Messages *rspA* and *rspC* may arrive at *B* in any order. *B* redirects *rspA* to *C* as message *rdrA*—note that it still may arrive at *C* sooner than *reqC*. After *A* has responded to *B*, it sends *lst*, which gets lost, though.

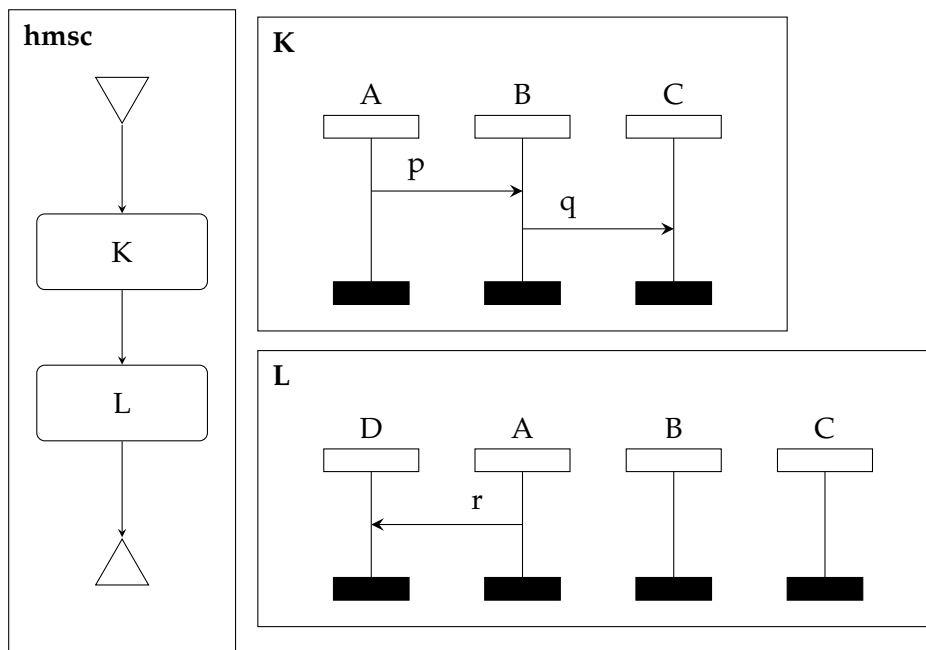


Figure 2.2: An example of asynchronous behaviour. Messages *q* and *r* may be sent or received independently of each other, i.e., *r* might be sent after delivery of *q*, but *r* might, alternatively, be sent right after *p*, even before *p* is received by *B*.

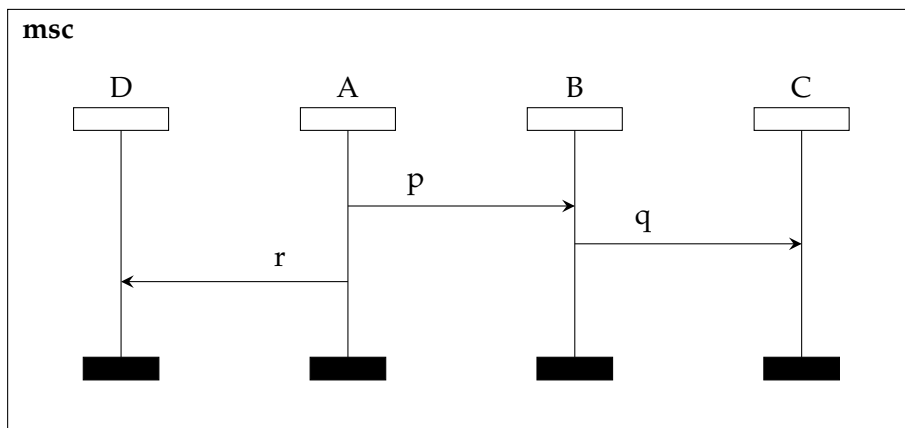


Figure 2.3: An MSC equivalent to that in Figure 2.2.

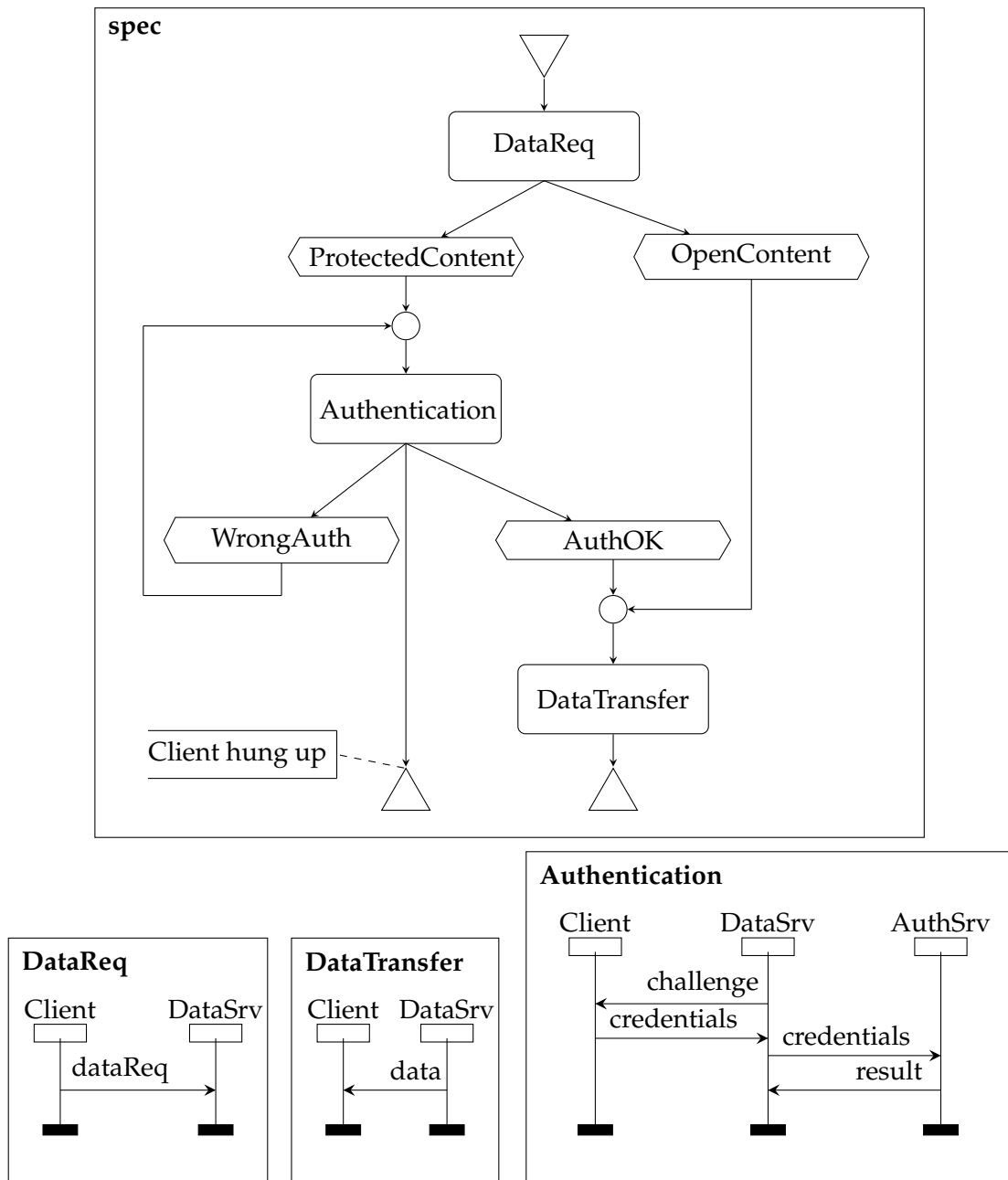


Figure 2.4: An example of HMSC support in SCStudio. First, the *DataReq* node gets executed, i.e., *Client* sends *dataReq* to *DataSrv*. If the *OpenContent* condition is satisfied, the *DataTransfer* node is executed. Otherwise, if *ProtectedContent* condition is met, the system continues with *Authentication*. Depending on the result, *Authentication* is executed again, or the *DataTransfer* begins. For those familiar with Subversion, that could be a (simplified) description of a request for versioned data.



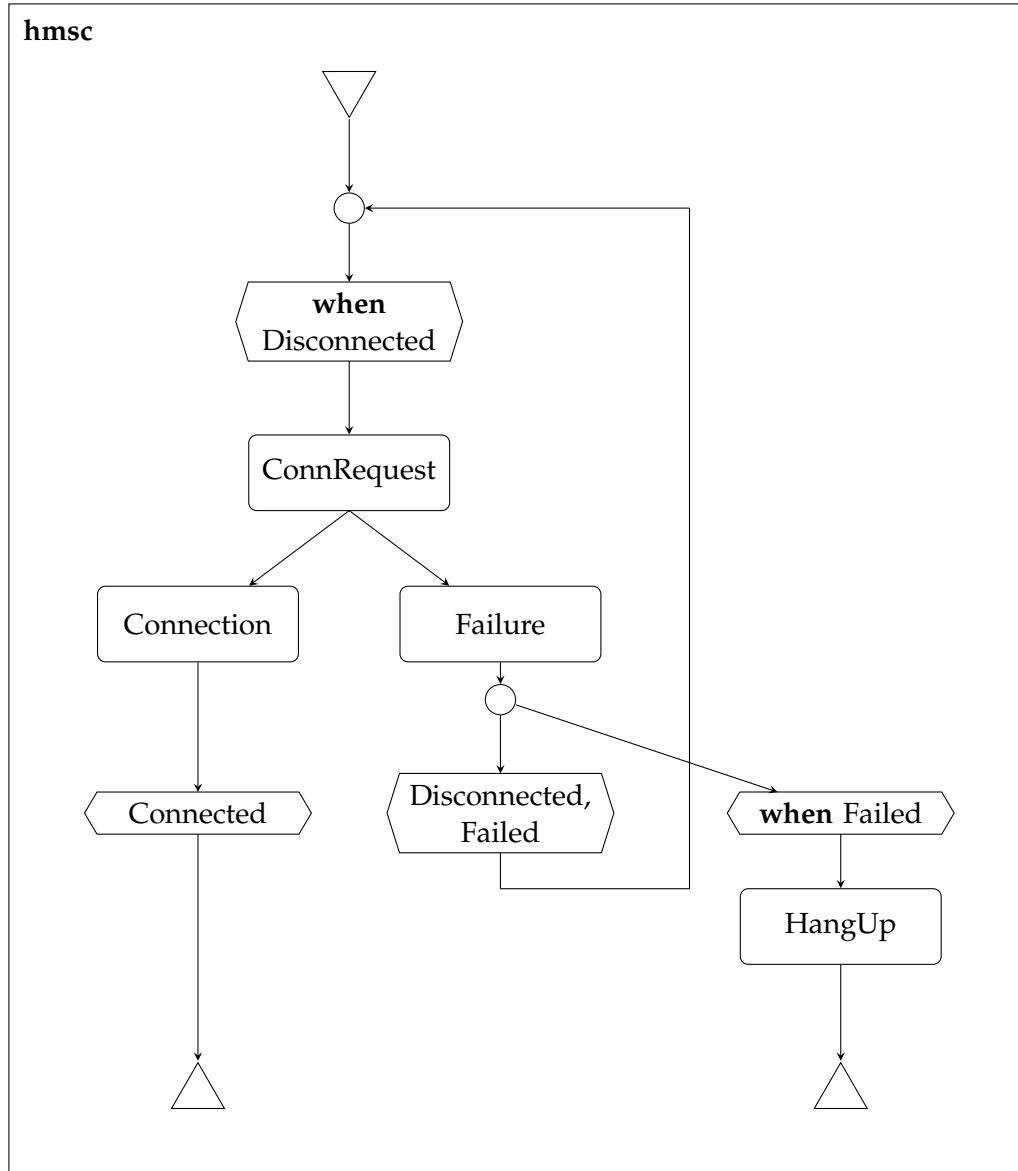


Figure 2.6: An example of conditions usage.

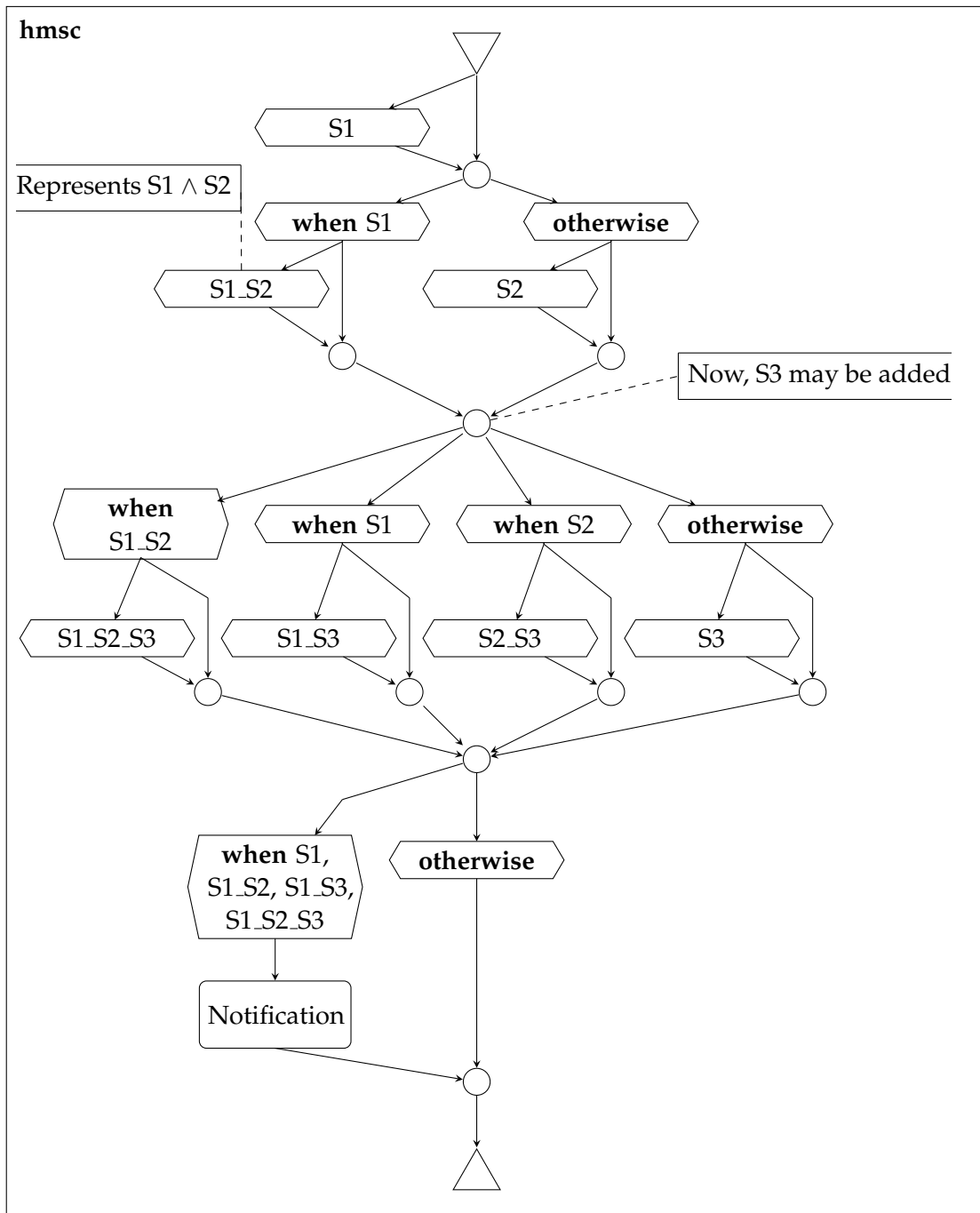


Figure 2.7: Preserving a part of the system state with conditions as specified by the MSC recommendation. Each of the condition variables to be potentially preserved may or may not have been set. To set another condition variable, it is necessary to examine all possibilities.

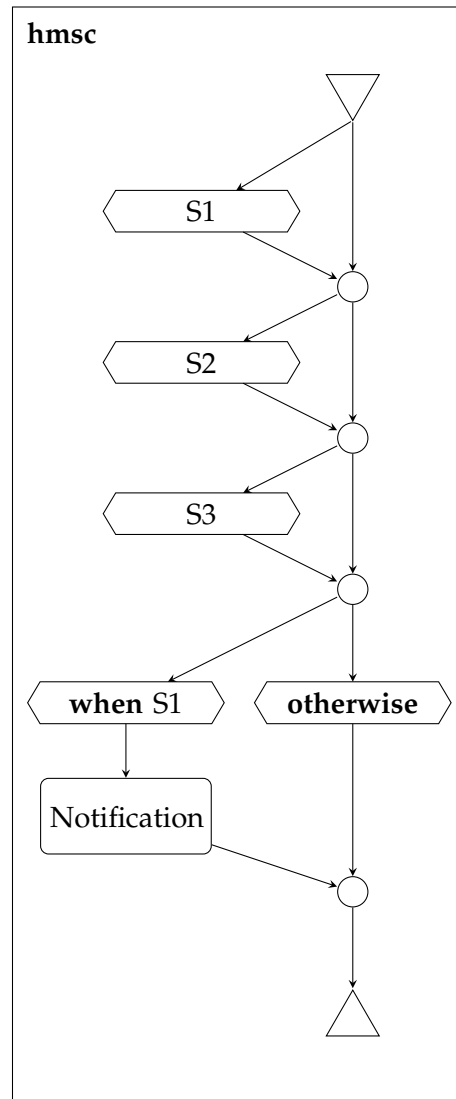


Figure 2.8: Diagram of the same use case as in Figure 2.7, using the proposed alternative conditions.



## Chapter 3

### Conditions Evaluation

This chapter proposes an algorithm for evaluating the alternative conditions, as specified in Section 2.3.3. There are some restrictions over the specification, mentioned in Section 3.1. As conditions evaluation is to be added to SCStudio, the algorithms currently being used for traversing MSC diagrams are proposed to be modified rather than introducing a completely new algorithm. Hence, Section 3.2 goes through the methods MSC diagrams are traversed in SCStudio. Finally, Section 3.3 proposes the modifications enabling conditions evaluation.

#### 3.1 Restrictions over the Conditions Specification

Among all the practice-oriented extensions proposed in previous chapter, the implementation of conditions in this thesis will be restricted in two aspects: the data-language-expression guarding conditions and condition ranges.

The data-language-expression guarding conditions will be omitted for obvious reasons: SCStudio has no support for data language within MSC diagrams, and adding such capabilities is beyond the scope of this thesis. Therefore, evaluation of guarding conditions containing data language expressions may not be implemented.

As for the latter restriction, only global conditions will be implemented for simplicity reasons. Although both the MSC recommendation and the proposition in the previous chapter allows for specifying subset of instances which a condition affects, the practical impact of this possibility is considered too low, compared to the complications it introduces. The data structure will contain the information, though, for future extensions.

#### 3.2 SCStudio Algorithms for MSC Traversal

Since the initial implementation, SCStudio core has implemented a very generic way of traversing MSC diagrams. The aim was to encapsulate the data structure, and thus separate it from algorithms traversing it. Thanks to this, to adjust the traversal, it is only necessary to change the traverser algorithms, while checkers and other parts of SCStudio remain untouched. Thus, the main part of integrating conditions evaluation is a modification of traversers.

The concept of traversers, including examples, is explained in [2]. Speaking in terms of design patterns (see, e.g., [7]), the algorithms act as observers, while the traverser actually accessing the data structure is an observable subject. An algorithm first registers itself (or an auxiliary object) for some types of traversal events at the traverser. Then, the traverser starts walking through a given diagram, notifying the observers about reached nodes.

There are several kinds of traversers, each iterating the diagrams slightly differently. Most of them are based on the depth-first search (DFS), as presented in [5]—with nodes colored white, grey, or black, according to their actual state in the search.<sup>1</sup> Upon traversing, when reached a node, the traverser notifies all the algorithms which registered for receiving events of reaching (or leaving) a node of a specific color.

By the combination of particular traverser and types of traversal events listened to, various algorithms may be implemented without actually touching the data structure. For instance, a cycle-detecting checker algorithm might use a DFS traverser and register to receive events on reaching a grey node—the reaching of which implies a cycle.

An overview of all traverses used in SCStudio is given in Table 3.1 and Table 3.2.

### 3.3 Traverser Algorithm

Implementation of conditions semantics in SCStudio will be performed as a modification of traversers, introduced in the previous section. The key aspect of conditions is that they exclude those system behaviours for which the guarding condition (if any) is not satisfied. In SCStudio, only HMSC can be used for branching the specification to several alternative system behaviours.<sup>2</sup> Hence, it is sufficient to modify only HMSC traversers.

There are two problems, though. First, some algorithms actually bypass traversers and access the internal data structure directly. Those will have to be adapted on an individual basis. Second, some traversers move backwards. That will have to be treated separately, and is left out to Section 3.3.2. For now, let us consider just forward traversers.

#### 3.3.1 Forward Traverser Algorithm

All the forward HMSC traversers are modified in a similar way. There are two fundamental modifications.

As the first necessary modification, branches guarded by unsatisfied conditions are not reported to the listeners. The algorithms implemented in SCStudio are supposed to rely only on information retrieved by the listeners they employ, so this modification

---

1. A node is white if it has not been reached yet, grey if already reached but DFS is still running for some of its successors, or black when all of its successors have been processed.

2. The MSC recommendation also specifies alternative inline expressions, which may lead to branching even in BMSC. Inline expressions are not supported by SCStudio, though.

effectively cuts off unsatisfied HMSC branches, as required by the conditions specification. It is a modification transparent to the algorithms—from their point of view, the unsatisfied branches simply do not exist in the HMSC, as they are not reported by the traverser.

For some HMSCs, that modification would be sufficient. There is a problem, though: due to cycles, a node may be executed repeatedly. If there are some setting or unsetting conditions on the cycle, however, the node may be revisited with different conditions. That plays a role if there is also a guarding condition in the cycle, or anywhere later, which refers to a condition variable set or unset within the cycle. The DFS search, when reaches an already visited (grey-colored) node, merely notifies grey node listeners and backtracks. Some guards may have got enabled (or disabled), though, which were not enabled (disabled) for the first time, thus, some new branches could have got available (or cut off). The problem is illustrated in Figure 3.1.

Hence, the second modification is proposed. The idea behind the algorithm is that the search, when reaching the same HMSC node with different set of conditions, will consider it as a new, yet unreached vertex. To do that, the system state (i.e., a set of condition variables) will be preserved with each vertex. The HMSC will get “unfolded” this way to an equivalent HMSC in which semantics of setting and unsetting conditions need not be evaluated. In this unfolded HMSC, branches guarded by unsatisfied conditions will be cut off and the resulting HMSC will be actually presented to algorithms. Due to the equivalence, the algorithms (which do not evaluate conditions) will work correctly. Formal definitions follow.

**Definition 5.** Let  $H$  be an HMSC,  $G = (N, E)$  a graph where  $N$  is the set of nodes from  $H$ ,  $E$  is the set of connection lines in  $H$ . The unfolding of  $G$  for HMSC  $H$  is a graph  $U = (N \times 2^C, F)$ , where:

- $C$  is the set of condition variables used in  $H$  and all its reference nodes, recursively, and
- $F = \{((a, \sigma), (b, \sigma')) \mid (a, b) \in E, \sigma \xrightarrow{a} \sigma'\}$ .

An element  $k \in N \times 2^C$  is called an extended node of  $H$ .

**Definition 6.** Let  $U = (K, F)$  be the unfolding of HMSC  $H$ . The cut-off unfolding of HMSC  $H$  is defined as:

$$(K, F \setminus \{((a, \sigma), (b, \sigma')) \in F \mid b \text{ is a guarding condition not evaluated to true in } \sigma'\})$$

An example of unfolding of an HMSC follows. In Figure 3.2, the original HMSC is presented. The nodes are numbered for further reference. Figure 3.3 shows the unfolding of the HMSC. Each node is commented with the original node numbers and the set of system states valid when the node is entered. The cut-off unfolding is almost the same as the unfolding in Figure 3.3, but without the edge from  $(s_6, \emptyset)$  to  $(s_7, \emptyset)$ .

### Implementation Notes

Regarding the actual implementation of conditions evaluation, it shall not compute the complete unfolding, as it is unnecessary. The extended nodes shall be created on-the-fly

as required, thus, only reachable extended nodes are processed.

Yet before an HMSC traverser is run, a pre-checker shall be executed, checking whether all the static requirements, introduced in Section 2.3, and restrictions mentioned in Section 3.1 are met. That guarantees the required preconditions for the traversers to run correctly.

#### 3.3.2 Backward Traverser Algorithm

The problem with backward traversers is apparent: they first enter guarded branches, and only then may find out whether the guard was satisfied.

One solution to the problem would be to compute an explicit representation of the cut-off unfolding of the given HMSC and run the backward traverser on it. However, the complexity of such approach would be exponential to the number of system states used within the HMSC.

The actually implemented solution performs a forward traversal on the HMSC, storing the whole graph—which is the reachable subgraph of the cut-off unfolding—in memory, and traverses this graph backwards. While the worst-case complexity is the same as in the first approach, it computes only those states which are actually used, hence, is the preferred one.





### 3. CONDITIONS EVALUATION

Name of Traverser Class	Note
DFSEventsTraverser	DFS-traverses a given BMSC from all instances, each from minimal events, follows matching message events
DFSBackwardTraverser	backwards variant of DFSEventsTraverser
DFSInstanceEventsTraverser	similar to DFSEventsTraverser, but does not follow matching message events
DFSAreaTraverser	DFS-traverses all event areas in all instances, independently of each other

Table 3.1: Overview of BMSC traversers.

Name of Traverser Class	Note
DFSBMscGraphTraverser	DFS-traverses an HMSC, each HMSC is traversed as many times as it is referenced
DFSInnerHMSCTraverser	similar to DFSBMscGraphTraverser, but does not care whether there are some end nodes in referenced HMSC—just goes on
DFSHMscTraverser	similar to DFSInnerHMSCTraverser, but processes each HMSC at most once
DFSBMscTraverser	backwards variant of DFSHMscTraverser
DFSRefNodeHMSCTraverser	similar to DFSBMscGraphTraverser, but finds successor nodes using NodeFinder
DFSHMscFlatTraverser	similar to DFSBMscGraphTraverser, but does not recurse, i.e., processes just top layer of HMSC
DFSRefNodeFlatHMSCTraverser	similar to DFSHMscFlatTraverser, but finds successor nodes using RefNodeFinder
DFSHMscsTraverse	traverses HMSC using DFSHMscFlatTraverser and HMSC found event
ElementaryCyclesTraverser	traverses all elementary cycles in an HMSC
FootprintTraverser	traverser used by race checker
NodeFinder	finds successors or predecessors of a node, skipping connection nodes
RefNodeFinder	similar to NodeFinder, but skips all but reference nodes
NonemptyNodeFinder	similar to NodeFinder, but skips all but nodes referring to a non-empty BMSC
AllPaths	traverses the HMSC for all distinct paths, which are reported to listeners

Table 3.2: Overview of HMSC traversers.

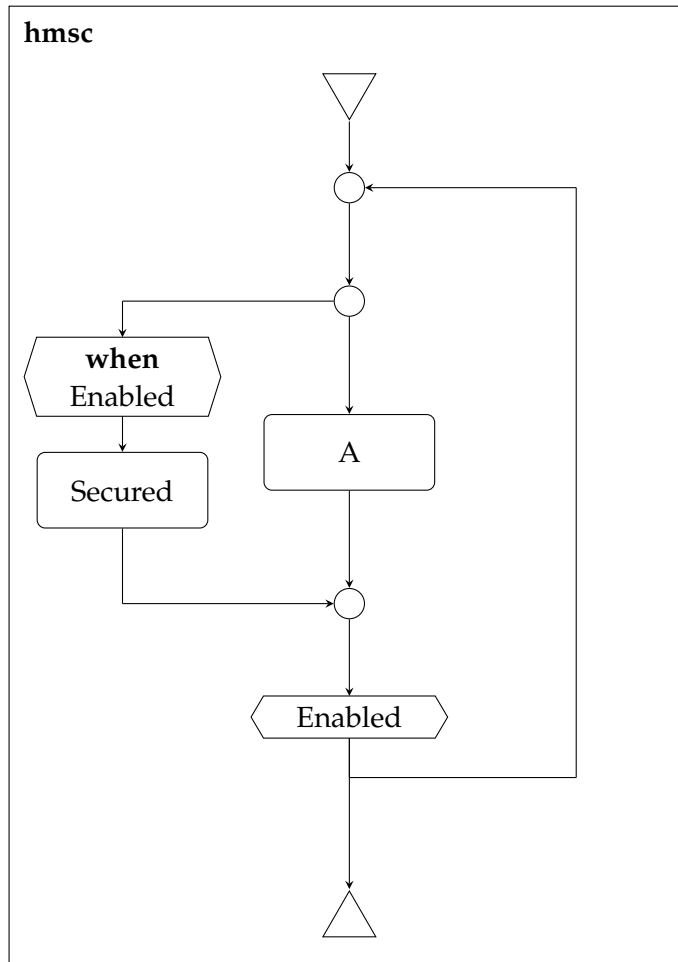


Figure 3.1: The *Secured* reference node gets reachable only when repeating the cycle, after setting the *Enabled* condition variable.

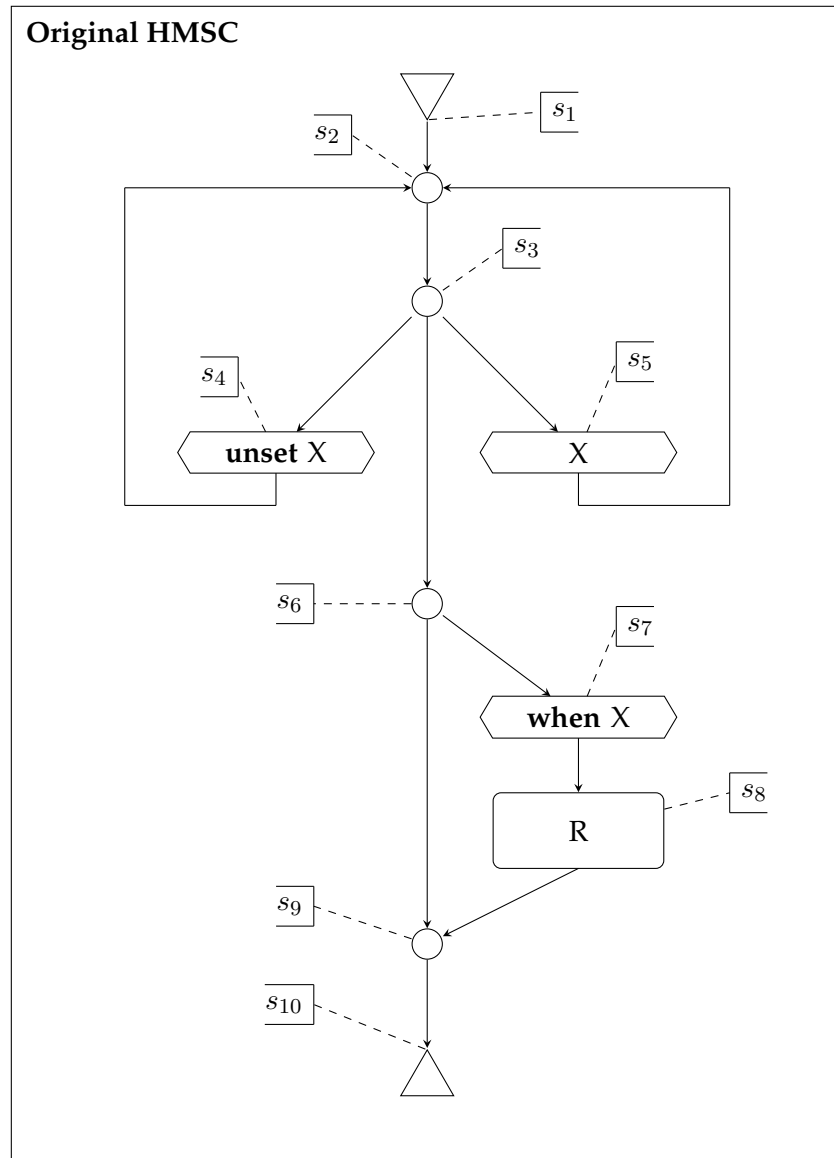


Figure 3.2: An HMSC to be unfolded. Note the comments labelling the nodes for reference in Figure 3.3.

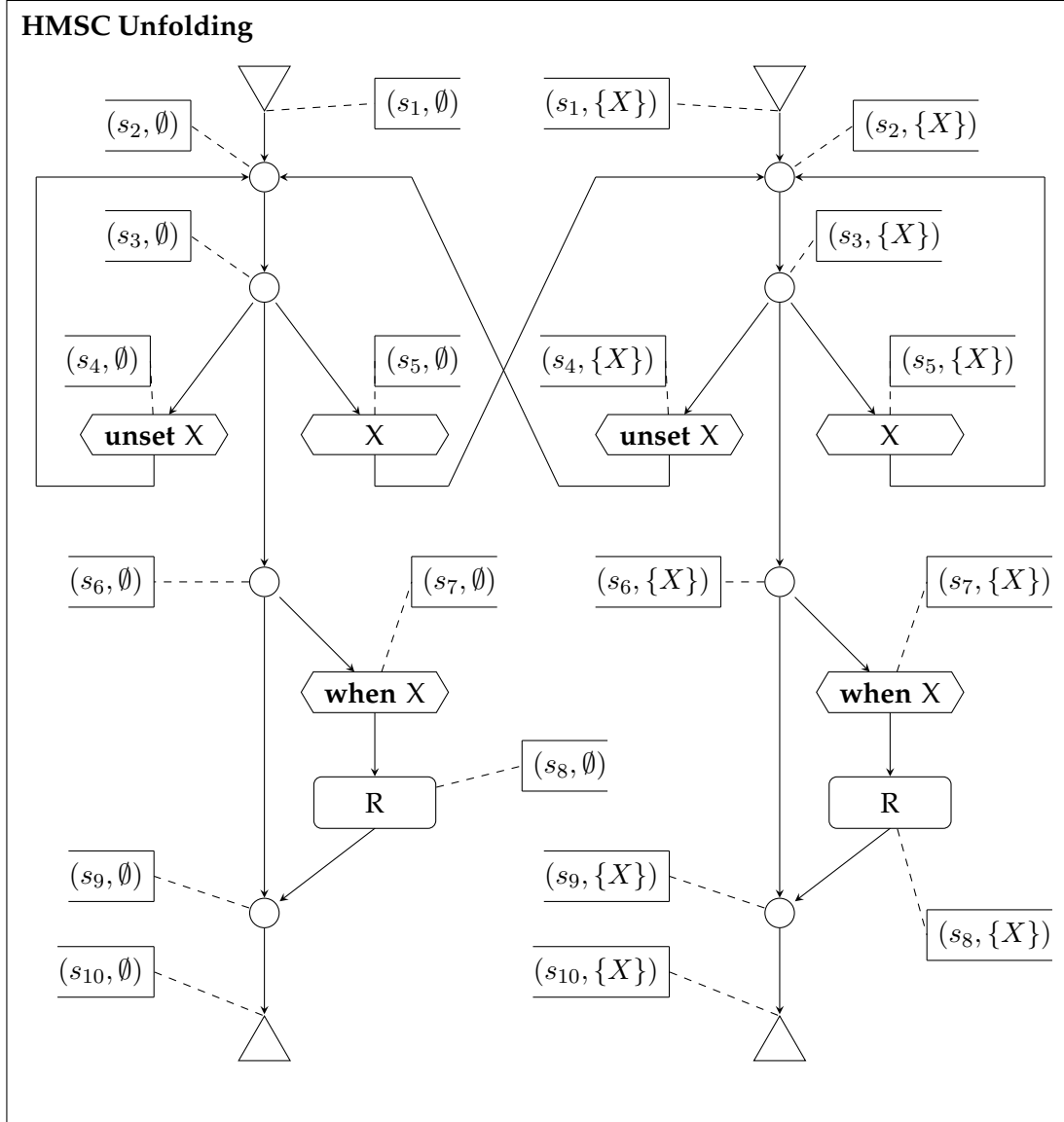


Figure 3.3: The unfolding of HMSC from Figure 3.2. Original nodes have been supplemented with sets of condition variables valid in the extended nodes.



## Chapter 4

### Internal Structure Refactoring

As required by Chapter 2, the internal data structure, which represents MSC diagrams in memory, has to be extended. In particular, two new types of BMSC events have to be added for representing local actions and conditions. However, with the current object model, it is not possible. The initial design [2] took only message events into account and, as analysed in Section 4.1, does not allow for generalizing this type of events and adding new subtypes. Hence, before actually implementing the support for local actions and conditions, the object model has to be reworked—refactored, as defined by [6]:

“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.”

Indeed, the goal is to make the object model cleaner and extendable, while the changes must not alter the external behaviour. Moreover, with regards to the volume of code depending on particular features of the internal structure, a series of gradual changes has to be made rather than a complete redesign of according parts. The list of proposed changes is stated in Section 4.2. Finally, in Section 4.3, the refactored object model is described.

#### 4.1 Initial State

The initial state of the internal data structure part relevant to BMSC diagrams is depicted in Figure 4.1. Each type of an MSC element has its own class, inheriting from `MscElement` class as a common ancestor. Class `BMsc`, representing the whole BMSC diagram, is a subclass of `MscElement`, too. Also note `StrictOrderArea` and `CoregionArea` classes, which represent the according segments of an instance.

As it can be seen, the `Event` class should be better named `MessageEvent` since it actually represents a message sending or receiving event, and it contains the `m_message` attribute referring to an `MscMessage` object. Apart from other similar imperfections, the most problematic is the type differentiation of events to `StrictEvent` and `CoregionEvent` according to whether they are contained in a `StrictOrderArea` or `CoregionArea`. The

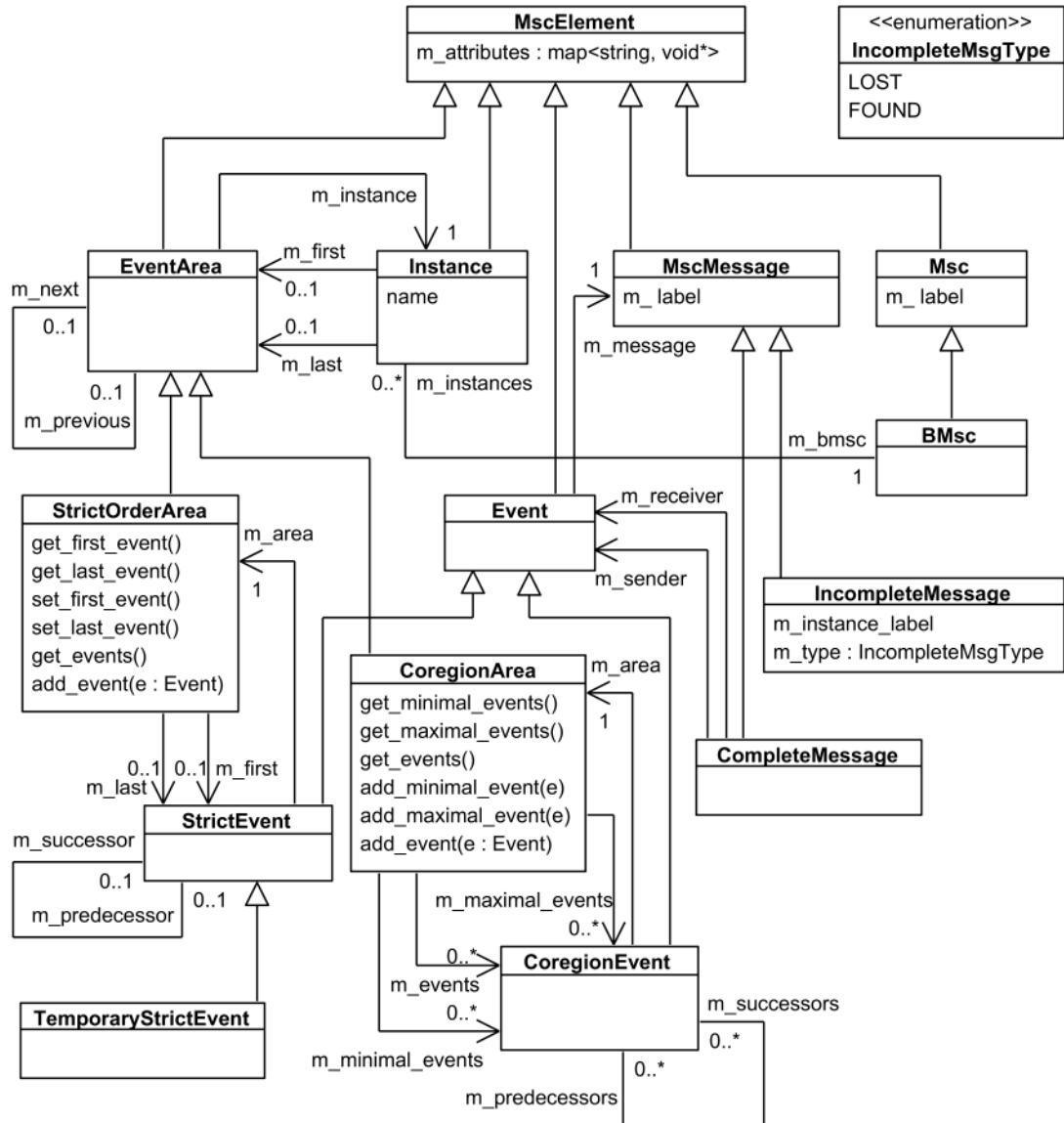


Figure 4.1: Class diagram of the relevant part of the original data structure.



events themselves manage their relations to other, predecessor or successor events, which, however, depends on the type of event area. The application code, which needs to traverse an instance, then has to typecast an event at runtime to correctly get its successors. That prevents from generalizing the `Event` class and defining other types of events. If, for instance, a new class `LocalActionEvent` was introduced, all the other code would have to differentiate not only between two types of message event, but among two additional types of local action event, each according to its placement (either for strict-order area or coregion area). That would have to be further extended with each new event type. Overall, adding a new type of event would touch all code which traverses the diagram. The only workaround would be to take the `Event` class as a generic class for events, which would have an attribute for each possible type of element: `m_message` pointing to a message, `m_local_action` for a local action, etc. Nevertheless, the base design is wrong in that an event does too much—it should only manage its own properties, especially the reference to its MSC element (e.g., a `MscMessage` object). If the responsibility for the order of events within an event area was given solely to the event area, the problem would be solved and the model could be extended with new types of events. Likewise, the order of event areas on an instance should be governed by the instance, not by the event area objects.

Another issue with the object model is that the `StrictOrderArea` and `CoregionArea` classes have each a different interface for getting or manipulating with events. Thus, all application code has to obtain the event area type at runtime and call the according methods—effectively substituting for the dynamic dispatch mechanism, which is one of the fundamental concepts in object-oriented programming [15]. Even though it does not prevent from extending the application with local actions or conditions, it is principally the same problem as with event types. In the following section, refactorings for fixing both these shortcomings are proposed.

## 4.2 Enhancement Proposals

The identified problems will get corrected by a sequence of refactorings, each of which performs a relatively small step towards the final result. The reason for dividing the whole operation into several steps is that after every refactoring, the application is in a stable state which may be tested and potential faults may be found and fixed easily.

1. `EventArea` attributes `m_previous` and `m_next`, and `Instance` attributes `m_first` and `m_last` will be removed. Instead, `EventArea` objects will be held in `m_event_areas` attribute of class `Instance`. The application code using the successor/predecessor access methods will be rewritten to use iterators on the according event area list.
2. For manipulation with contained events, generic methods will be defined in the `EventArea` class, which the `StrictOrderArea` and `CoregionArea` classes will implement:

## 4. INTERNAL STRUCTURE REFACTORING

---

- methods `get_minimal_events` and `get_maximal_events` to access the set of minimal or maximal events in the area,
  - methods `get_predecessor_events` and `get_successor_events` to retrieve the set of predecessor or successor events of a given event within the area,
  - method `add_event` to add an event as a new maximal event in the area.
3. `StrictEvent` attributes `m_successor` and `m_predecessor`, and `StrictOrderArea` attributes `m_first` and `m_last` will be removed. Instead, events in a strict-order area will be maintained in a list attribute `m_events` of class `StrictOrderArea`. Likewise, `CoregionEvent` attributes `m_successors` and `m_predecessors`, and `CoregionArea` attributes `m_minimal_events` and `m_maximal_events` will get replaced by oriented graph attribute `m_events` in class `CoregionArea`. The application code using the successor/predecessor access methods will be rewritten to use iterators on the according event list or graph, respectively, where feasible—in the remaining cases, successors or predecessors of an event will be retrieved by the according `EventArea` methods.
  4. `TemporaryStrictEvent` will be renamed to `TemporaryEvent` and inherit from `Event`, as it does not actually use anything from `StrictEvent`.
  5. Classes `StrictEvent` and `CoregionEvent` will be removed as their only function will have been superseded by event area classes.
  6. Class `Event` will be renamed to `MessageEvent`.
  7. A new class `Event` will be created as an ancestor of `MessageEvent`. The generic event functionality will be moved from `MessageEvent` to `Event`, so that the `MessageEvent` class will contain only attributes and methods specific to message events.
  8. The code using `MessageEvent` objects will be generalized to accept generic `Event` objects, where possible. Class `TemporaryEvent` will inherit from `Event` instead of `MessageEvent`.

### 4.3 Refactored State

As a result of refactorings performed according to the previous section, the object model was changed, as illustrated in Figure 4.2. The most important change is the generic `Event` class. It now serves as a base class for any type of events. The already existing event classes inherit from `Event`. See, e.g., the `TemporaryEvent` class, which represents auxiliary events related to time extensions [19]. In the old object model, it had to inherit from `StrictEvent`, even though it has no message-related functionality. In the new object model, `TemporaryEvent` is placed correctly, inheriting directly from `Event`. The new `MessageEvent` class now only represents message receive or send events, inheriting

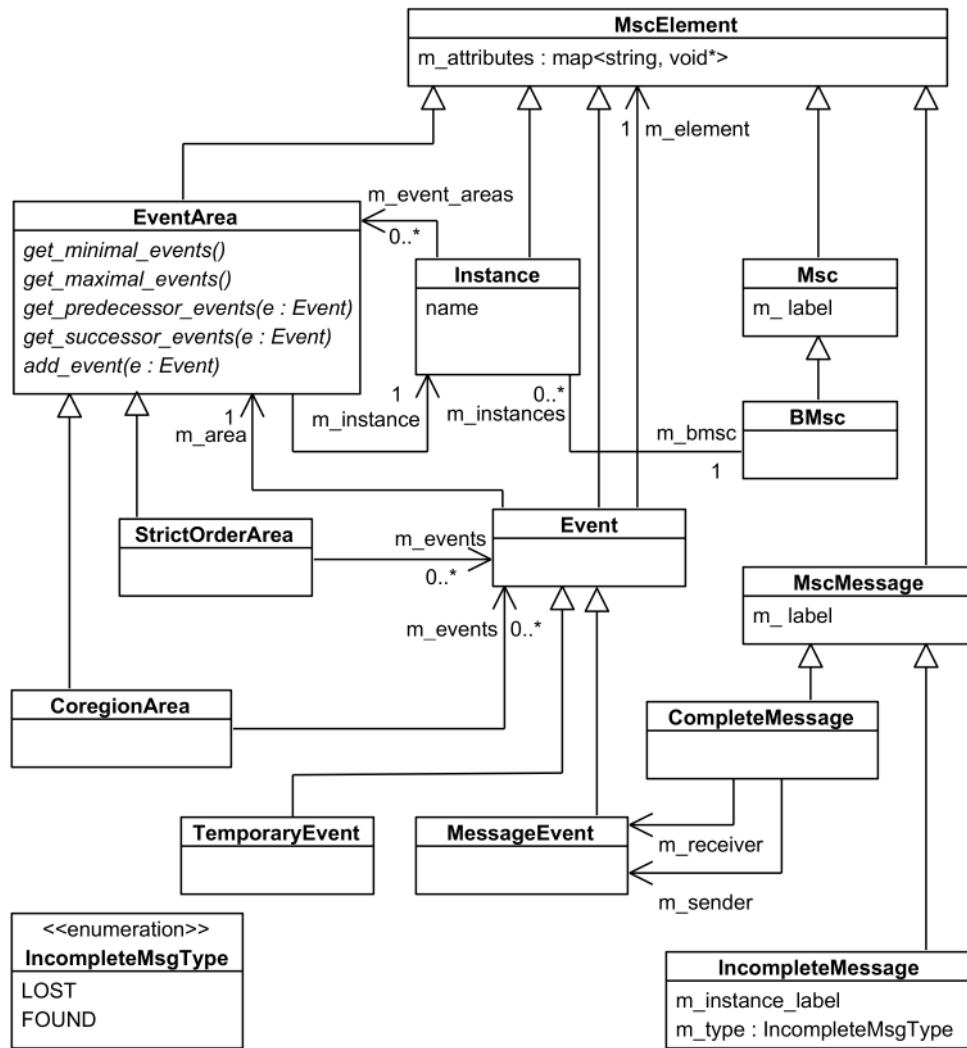


Figure 4.2: Class diagram of the refactored data structure.

the common functionality of events (assigned comments, temporal data, etc.) from the `Event` class.

With the new hierarchy of internal structure classes, new types of events may be added besides the `MessageEvent` class. For SCStudio to enable support for local actions and conditions in BMSC diagrams, the classes `LocalActionEvent` and `ConditionEvent` may now be created, inheriting from `Event`. Moreover, the overall hierarchy is apparently simpler, with less interdependencies, which is due to the escalation of responsibility for order of events purely to event areas, and order of event areas to instances. Considering the implementation, standard data structures (`std::list` [10], `boost::graph` [17]) were used for that, instead of custom pointer-linked structures, which makes the code more compact and easier to understand.

The interface of event area classes was unified. Both `StrictOrderArea` and `CoregionArea` still offer functionality specific to the according type of area, but also feature more methods implemented from the common `EventArea` parent. Much of the application code is now neutral to the particular event area type, yielding in more compact code and less runtime typecasting—which should speed up the execution.

Upon performing the refactoring steps, all dependent application code was adjusted to work correctly with the new class scheme. Thus, the overall goal has been accomplished, resulting in an improved structure while maintaining the external behaviour. As a side-effect of the refactoring, the set of automated tests now runs notably faster, and the number of memory defects reported by Valgrind [16] was reduced from 471 to 207.

## Chapter 5

### Implementation

The refactoring, performed according to the previous chapter, had a goal to make the internal structure extendable. The fact that this goal has been successfully achieved is best proved by actually extending the refactored structure with the new types of events. Both local action and condition event classes have been added to the structure, providing the backend support for local actions and conditions. In addition to adding these elements, evaluation of conditions has been implemented for a selected HMSC traverser. In this chapter, the implementation is briefly described.

#### 5.1 Local Actions

As mentioned earlier, local actions are rather easy to implement. Given the fact SCStudio does not support data language statements, local actions only serve documentation purposes, and are to be ignored by the algorithms.

Two new classes are introduced: `LocalAction` and `LocalActionEvent`. The former represents the local action element, defining its dimensions and textual content, while the latter denotes the event that occurs on an instance. The classes are added to the internal structure as follows in Figure 5.1. Note the `MscElementTmpl` and `EventTmpl` template classes, which were omitted from previous diagrams for simplicity reasons. Both these classes provide a generic methods for their subclasses (MSC elements, and BMSC events, respectively) which are declared using the template type, so that type-safe operations can be made. For instance, subclasses of `EventTmpl` are supposed to provide the type of element the event may bind to. Thus, the event may only be bound to the correct type of element, and returns exactly that type with its `get_element()` method.

After adding the aforementioned classes, the glue code between graphical frontend (which already supported local actions) and backend was adjusted. Other related parts of the application, e.g., exporting the objects to the MSC textual format, were worked out by other SCStudio developers.

#### 5.2 Conditions

Conditions are implemented similarly to local actions. Classes `Condition` and `ConditionEvent` are introduced, with analogous meaning. Besides defining these classes for use in BMSC diagrams and supplying the glue code between frontend and backend, condi-

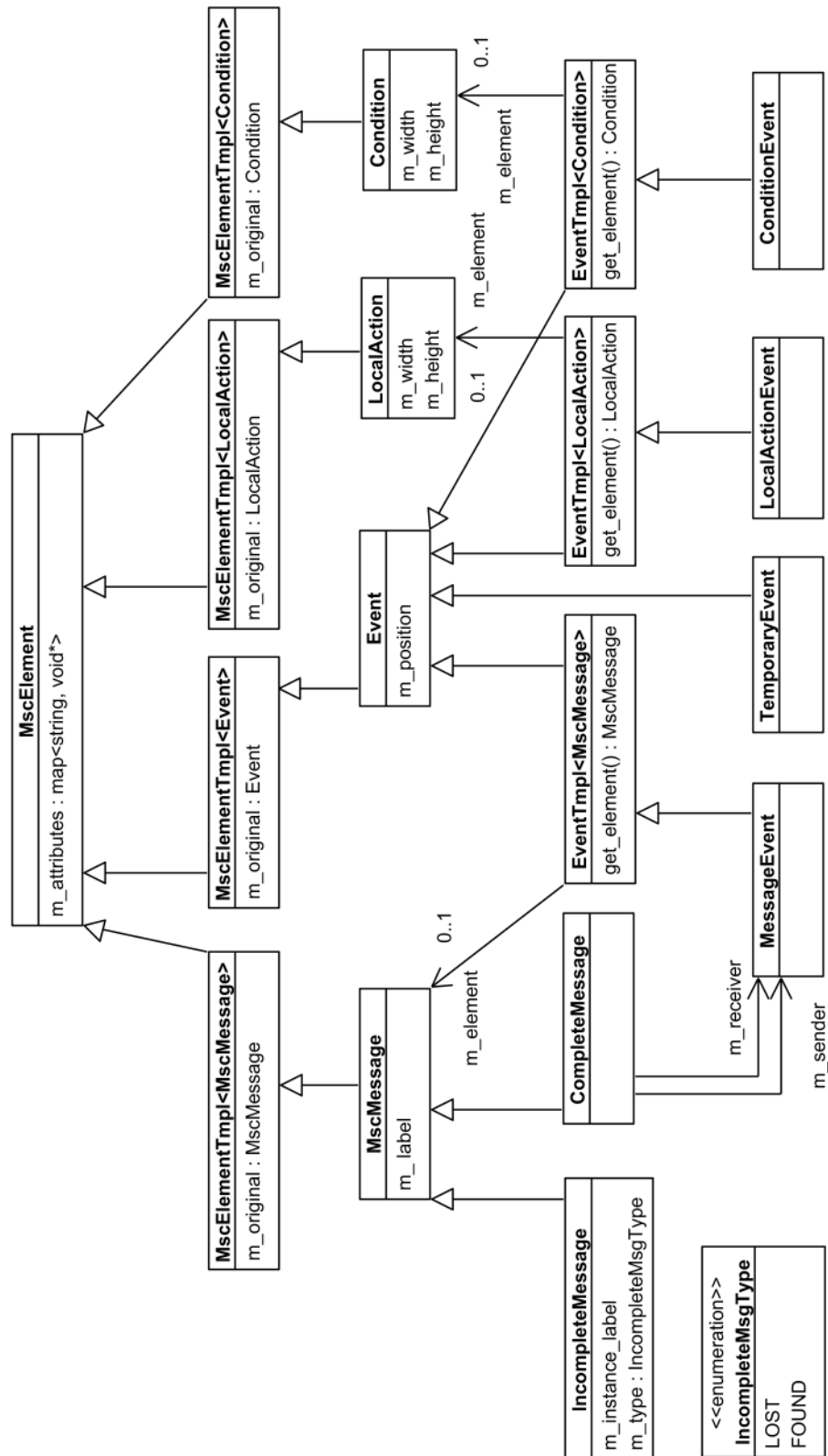


Figure 5.1: The new internal structure with local actions and conditions.

tions semantics have also been partly implemented, demonstrating the proposal from Section 2.3.3. HMSC traverser class `AllPaths` has been modified to evaluate the proposed semantics. This traverser is used by the time race checker [19] and several other algorithms. It searches for all paths within a given HMSC diagram in a depth-first-search manner and reports each path to its listeners. As there might be infinite paths, it features a custom limit for how many times each node may be visited before backtracking.

The `AllPaths` traverser is implemented by a recursive method which searches for successor nodes of a given HMSC node. Any successor which has not been visited too many times (not exceeding the limit) is passed to the recursive call, effectively implementing DFS. First, the notion of system state (which names all the condition variables that have been set) is introduced: the recursive method `all_paths` is extended to pass the system state along the node to be searched. Second, the methods for counting the number of visits of a given node are extended so that the visit count is recorded for a combination of the visited node and the system state it is visited in. Now, the traverser is ready for conditions evaluation. Each node is examined for its condition-effect (as defined in Section 2.3.3):

- condition variables from setting conditions enrich the current system state;
- conversely, condition variables from unsetting conditions are erased from the system state;
- reference nodes are also examined—a DFS traversal is executed to collect condition variables which should be set or unset.

The remaining task is to actually take the system state into account. When iterating over potential successors of a node, guarding conditions are evaluated with respect to the current system state. Those which are not satisfied get skipped, thus, the search only continues to valid branches.

An example follows in Figure 5.2, in which the time race checker reported a race condition between the two messages. Indeed, if the conditions were not evaluated, branches containing both *A* and *C* reference nodes could have been taken. When the condition evaluation is applied, however, at most one of the branches is permitted—and the time race checker reports the diagram as valid.

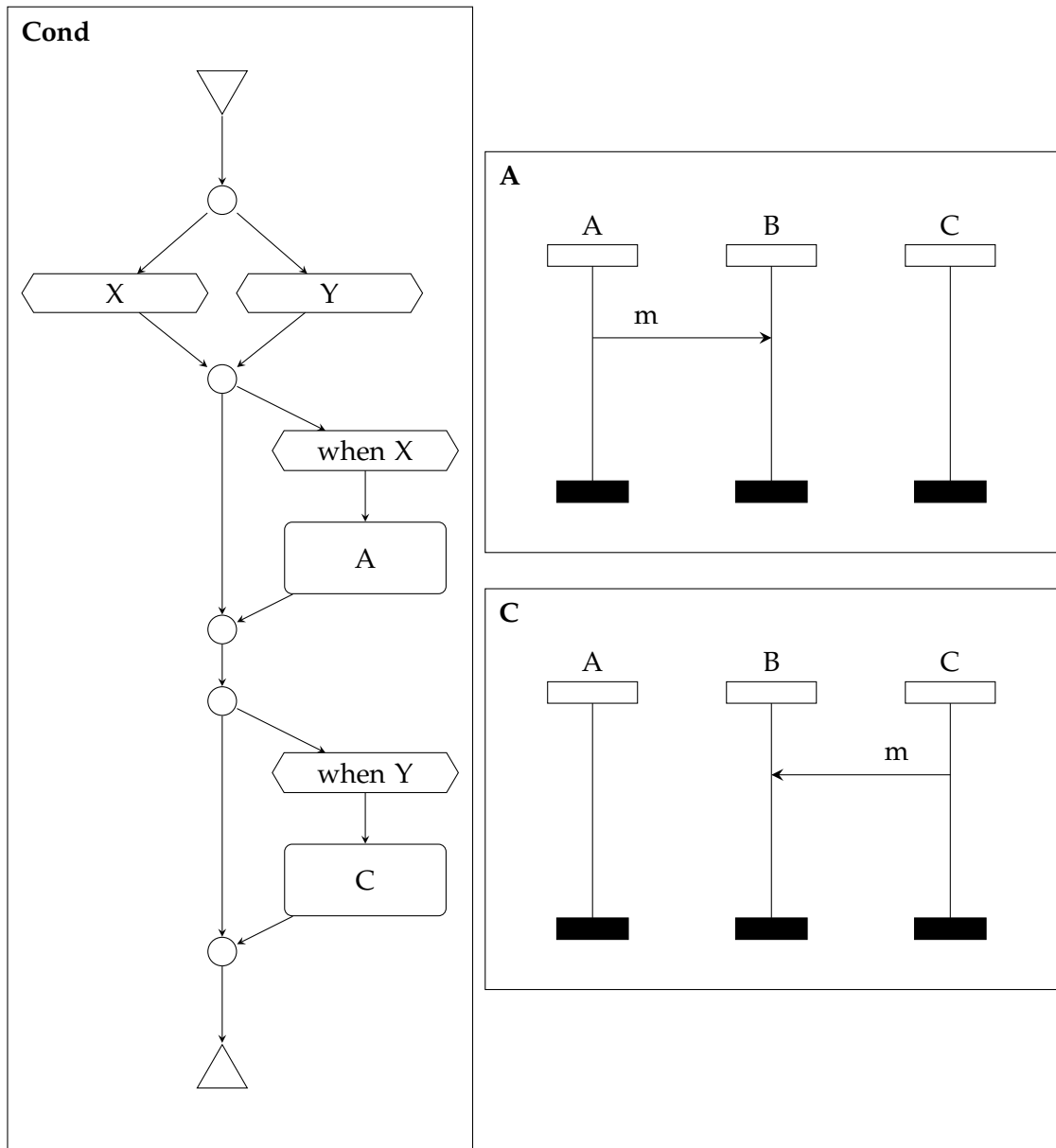


Figure 5.2: Example of an MSC diagram which is valid only when the conditions semantics is considered. If both *A* and *C* branches were executed, a race condition would occur at instance *B*: the messages from *A* and *C* could arrive at any order. As either *X* or *Y* condition variable is set, but not both at the same time, only one of the referred BMSC diagrams may be executed, thus, no race condition occurs.



## Chapter 6

### Conclusions

The goal of this work was to refactor SCStudio internal data structure. Since the initial version, only single type of events was allowed, and by design, further types of events could not be added. The object model had to be altered, without any external impact on the rest of the application. Two new types of events should have been added then—local actions and conditions.

The refactoring was performed in several consecutive steps. The object model was gradually improved to a simpler and, more importantly, extendable state. All dependent application code was adjusted appropriately; SCStudio has been developed for more than five years, thus, a very high volume of code had to be revised. Both local actions and conditions were implemented to the refactored data structure. The Message Sequence Chart recommendation was to be adhered to, which turned out rather imperfect in question of conditions semantics. An alternative syntax and semantics were proposed.

As a result, the application external behaviour was maintained, while the internal structure has become cleaner and richer of two new types of events. Standard libraries were used instead of some custom code, which resulted in a great reduction of memory flaws reported by dynamic analysis tools. The results have already been used by other project collaborators.

## 6. CONCLUSIONS

---

## Bibliography

- [1] Sequence Chart Studio Documentation. <<http://scstudio.sourceforge.net/help/>>. [Online; accessed 2-April-2013].
- [2] Jindřich Babica. Message Sequence Charts properties and checking algorithms. Master's thesis, Masaryk University, 2009.
- [3] Martin Bezděka, Ondřej Bouda, Ľuboš Korenčiak, Matúš Madzin, and Vojtěch Řehák. Sequence Chart Studio. In *2012 12th International Conference on Application of Concurrency to System Design*, pages 148–153, Los Alamitos, California, USA, 2012.
- [4] Martin Chmelík. Realizability of Message Sequence Graphs. Master's thesis, Masaryk University, 2011.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 2001.
- [6] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
- [7] Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Incorporated, 2004.
- [8] ITU Telecommunication Standardization Sector – Study group 17. ITU recommendation Z.100, Specification and Description Language (SDL), 2002.
- [9] ITU Telecommunication Standardization Sector – Study group 17. Recommendation ITU-T Z.120: Message Sequence Charts (MSC), 2011.
- [10] Nicolai M. Josuttis. *C++ Standard Library: A Tutorial and Handbook, The*. Addison-Wesley Professional, 1999.
- [11] Matúš Madzin. Import of MSC Diagrams From the ITU-T Z.120 Text Representation. Bachelor's thesis, Masaryk University, 2009.
- [12] Steve McConnell. *Code Complete*. Microsoft Press, 2004.
- [13] Helmut A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer, 1990.

- [14] Zuzana Pekarčíková. Computer Aided Layout of Message Sequence Charts. Bachelor's thesis, Masaryk University, 2011.
- [15] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- [16] Julian Seward, Nicholas Nethercote, and Josef Weidendorfer. *Valgrind 3.3-Advanced Debugging and Profiling for Gnu/Linux Applications*. Network Theory Ltd., 2008.
- [17] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The*. Addison-Wesley Professional, 2001.
- [18] Václav Vacek. New Checkers for Sequence Chart Studio. Master's thesis, Masaryk University, 2011.
- [19] Ľuboš Korenčiak. Time Extension of Message Sequence Chart. Bachelor's thesis, Masaryk University, 2009.
- [20] Ľuboš Korenčiak. Effective Algorithms for Time Relation Checking in Message Sequence Charts. Master's thesis, Masaryk University, 2011.
- [21] Vojtěch Řehák, Matúš Madzin, Ľuboš Korenčiak, Petr Gotthard, Ondřej Kocian, Martin Bezděka, Ondřej Bouda, Václav Vacek, Milan Malota, and Zuzana Pekarčíková. Sequence Chart Studio: user-friendly drawing and verification tool for MSC, 2012.