MASARYK UNIVERSITY
FACULTY OF INFORMATICS

New Checkers for Sequence Chart Studio

MASTER'S THESIS

Václav Vacek

Brno, 2011

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.

Brno, January 10, 2011
Václav Vacek

Advisor: RNDr. Vojtěch Řehák, Ph.D.

Acknowledgement

This thesis was created within a joint project of ANF DATA spol. s r.o. and the research centre Institute for Theoretical Computer Science (ITI). I would like to thank Vojtěch Řehák for his help and advice.
Most of all, I want to thank my girlfriend Dominika for her continuous support.

Abstract

For the specification of communication between entities, the formalism called *Message Sequence Chart* is often used. Even though it originates from the telecommunication area, its usage is not limited to it.

A team of students at the Faculty of Informatics has been developing an application for drawing and verifying Message Sequence Charts called *Sequence Chart Studio (SCStudio)*. This thesis describes a part of the implementation process:

The existing verification algorithm have been revised, corrected and improved and five new verification algorithms have been implemented. In addition, transformation of a MSC-specification to the language of an LTL-model-checking tool (DiVinE) has been enabled. Finally, several new generic functions have been made reusable to simplify further development of SCStudio.

Keywords

Message Sequence Chart, MSC, Sequence Chart Studio, SCStudio, verification, model of communication, model checking, LTL, race condition, local choice, boundedness

Contents

Chapter 1

Introduction

When developing a complex distributed system, the planning phase is one of the most important. The behavior of the system needs to be precisely specified during this phase as the specification serves as a basis for all the remaining parts of the development. There are several levels of abstraction and also several possible views of a system.

In practice, there are three basic tasks to be done during the design phase:

1. Identifying logical entities of the system
2. Describing the interaction between the entities
3. Describing local actions necessary for accomplishing the interaction

Phase two is much alike for the majority of communicating systems. Therefore, formalisms for describing communication between entities have been developed and even standardized. In this thesis we deal with one of the formalisms – *Message Sequence Chart* (MSC).

1.1 Message Sequence Chart

Message sequence chart (MSC) is a language recommended by the International Telecommunication Union (ITU) in [9]:

The purpose of recommending MSC (Message Sequence Chart) is to provide a trace language for the specification and description of the communication behaviour of system components and their environment by means of message interchange.

MSC has both a textual and an equivalent graphical form. In this thesis we present and use only the graphical version[1]. In the following sections we present the subset of the MSC language needed for the rest of the thesis.

1. SCStudio, nevertheless, supports both the forms.

1.1.1 Basic MSC

Basic MSC (BMsc) describes a set of communicating *instances* and a communication between them. A single *scenario* is specified by a BMsc. BMscs may contain the following elements:

Instance represents a system component. Events occuring on an instance are ordered in time from the top to the bottom of the instance (with the exception of events in a coregion as described below).

Complete messages: Communication between instances is performed via messages. Each message consists of two asynchronous events – sending and receiving.

Incomplete messages may be *lost* or *found*. The recipient or the sender, respectively, is not defined and such a message comprises only one event.

Coregion is a special area in an instance where the order of events is not linear. Events in a coregion may be (partially) ordered using the *general ordering*. Events that are not ordered may occur in an arbitrary order.

Figure 1.1 shows all the elements. Instance 1 only sends a message to Instance 2. Instance 2 sends a message to Instance 3 *after* receiving the message from Instance 1. Instance 3 contains a coregion with the general ordering. Therefore, the lost message is sent *after* receiving the found message and the message from Instance 2 may be received independently at any time (e.g. even before receiving the found message).

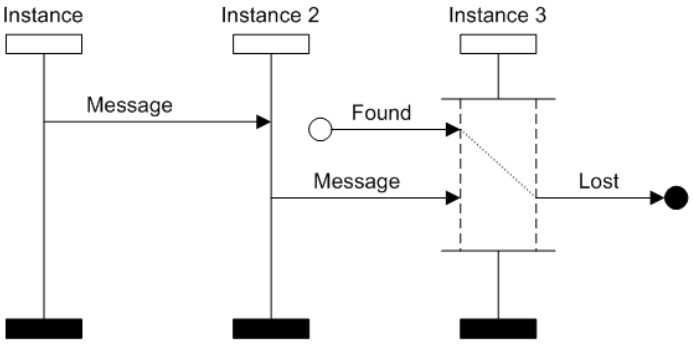

Figure 1.1: Example of BMsc

1.1.2 High-level MSC

As stated in [9], *simple scenarios (BMscs) can be combined to form more complete specifications by means of High-level MSC (HMsc). HMsc describes relationships between BMscs in a system.*

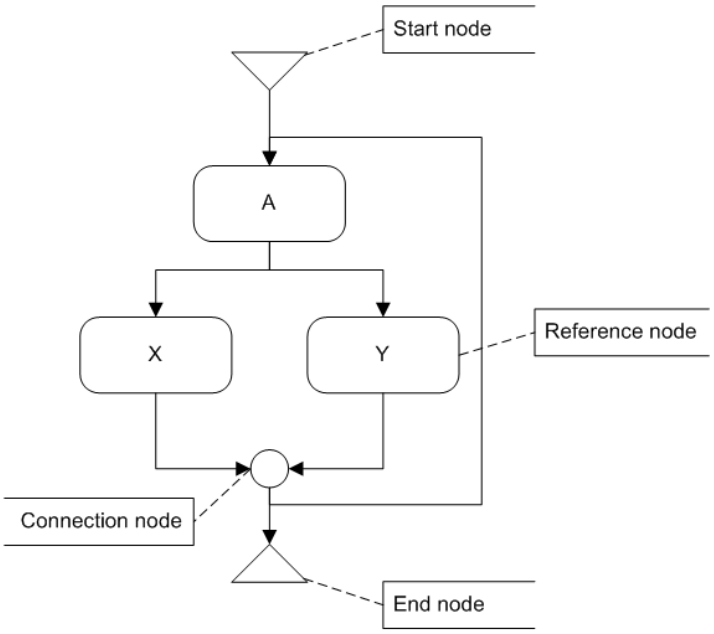

Figure 1.2: Example of HMsc

HMscs may contain the following elements:

Start node is the point the execution of an HMsc start at. From the start node the execution continues at some successor of the start node. There may be only one start node in an HMsc.

End node: In an end node, the execution of an HMsc ends.

Reference node references some other Msc (either a BMsc or an HMsc). When a reference node is reached during the execution, the referenced Msc is executed and after that the execution continues at some successor of the reference node.

Connection node corresponds to an empty operation. It can be used to clarify drawings because it sometimes reduces the number of necessary edges.

Connection line/arrow represents a directed edge between nodes. Using the arrow is recommended because the direction of an edge could be unclear with the line. Usually, an edge goes from the bottom part of a node to the upper parat of a node.

When each reference node in an HMsc references a BMsc (and not another HMsc), the HMsc may be called *BMsc graph*. A sample HMsc is shown in Figure 1.2. When the execution of the HMsc starts, Msc A is performed. After A ends, *either* X or Y is executed. Then the execution may end by entering the end node. Alternatively, the whole procedure may be repeated by going to A again.

1.1.3 Interpretation of HMscs

An HMsc is actually a directed graph: its nodes are vertices of the graph and connection lines and arrows are edges. As HMscs can be *hierarchical*, i.e. a reference node may reference another HMsc, a question how to interpret such HMscs in the terms of graphs naturally arises. We distinguish two interpretations:

Graph-based: Each reference node referencing an HMsc contains an extra edge going to the start node of the referenced HMsc. There is no edge from the end node of the referenced HMsc. An example of such an approach is shown in Figure 1.3.

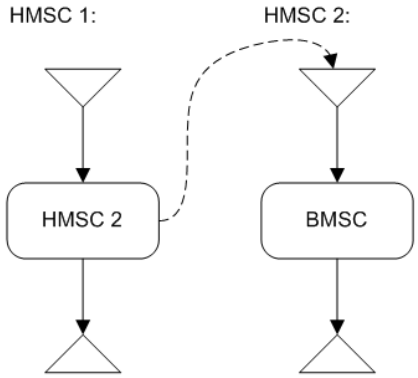

Figure 1.3: Graph-based interpretation

Notice this interpretation does not reflect the way the HMsc is executed: when HMSC 2 is finished, it is not possible to continue with HMSC 1

and the execution gets stuck in the end node of HMSC 2. The other interpretation deals with this issue.

Execution-based: A reference node referencing an HMsc is split into two nodes (in-node and out-node). All edges originally going to the node are connected to the in-node and all edges originally going from the node are connected to the out-node. Additionally, there is a node from the in-node to the start node of the referenced HMsc and another edge from the end node of the referenced HMsc to the out-node. Figure 1.4 shows an example.

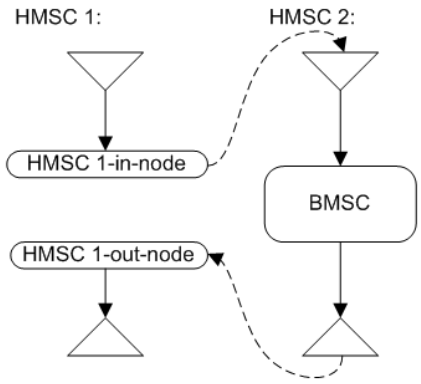

Figure 1.4: Execution-based interpretation

Using this interpretation, it is possible to simulate the execution of an HMsc by simply following edges. In this example, the execution of HMSC 1 may continue when HMSC 2 is finished.

From the execution-based interpretation, the transformation of an HMsc to an equivalent BMsc graph naturally follows. All in-nodes, out-nodes, start nodes and end nodes (with the exception of the start and end node in the root HMsc) are converted to connection nodes.Figure 1.5 shows a BMsc graph equivalent to the HMsc in Figure 1.4. Notice the graph- and execution-based interpretations are identical for a BMsc graph because they only differ in the interpretation of references to HMscs.

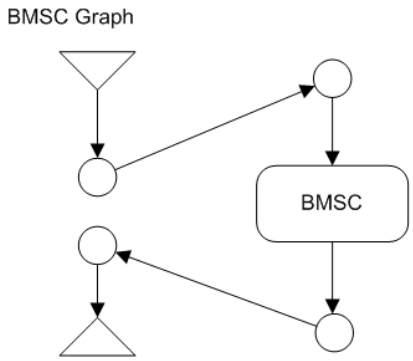

Figure 1.5: BMsc graph

1.2 SCStudio Overview

SCStudio is a tool for working with Message Sequence Charts. It provides support for drawing Mscs in Microsoft Visio [2], loading and saving Mscs in the textual form, algorithms for manipulating Mscs and verification algorithms. SCStudio is platform-independent with the exception of the integration with Microsoft Visio. A screenshot showing the Visio front-end is shown in Figure 1.6.

An overview of the architecture of SCStudio is depicted in Figure 1.7. The central part – *SCStudio core* – represents the basic functionality: a data structure for storing Msc together with a set of functions for creating, modifying and traversing Mscs is provided.

The left group represents *input and output modules*. The core part provides an interface allowing adding new modules.

The right group stands for *verification algorithms* (also called *checkers*). Each algorithm takes an Msc as an input and checks if some property is satisfied in the Msc. Similarly to input/output modules, it is possible to simply add a new verification algorithm using an interface provided by the core.

The bottom group represents *transformers*. Using transformers, it is possible to modify a given Msc. Beautifier [13], for example, changes the graphical layout so that the processed Msc is more legible and well-arranged. The core provides an interface for transformers as well.

Microsoft Visio integration actually spans over the whole SCStudio architecture because it provides the complete functionality of SCStudio from

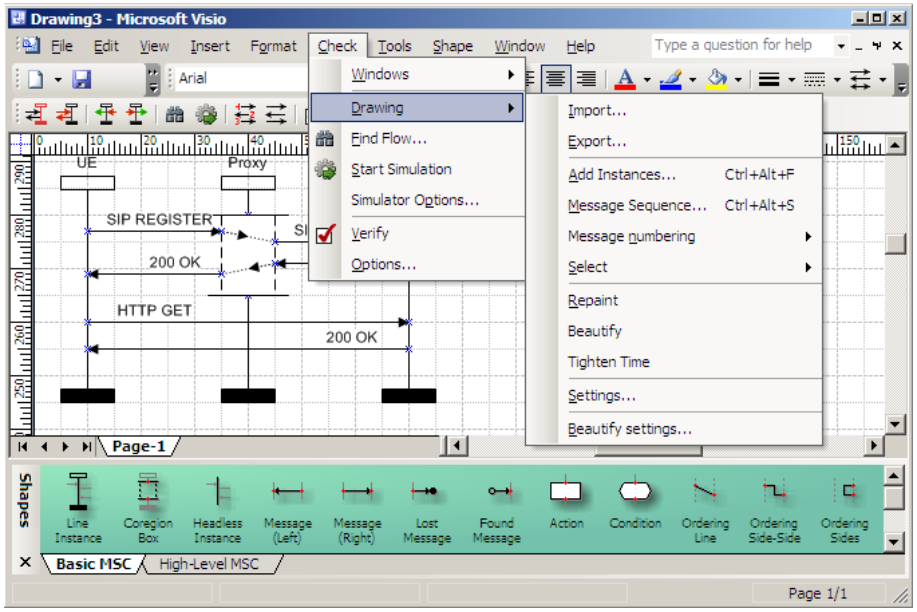

Figure 1.6: SCStudio front-end

within Visio. Additionally, functions for rapid Msc drawing are implemented (e.g. automatic drawing of a message sequence).

Detailed information about SCStudio may be found on its web page [3]. Information about the core functionality is described in [4].

1.3 Thesis Goals

This thesis aims at improving SCStudio by carrying out the following tasks:

- Revision of verification algorithms already implemented in SCStudio and correction of identified defects both in their definitions and implementation. In addition, support for multiple results of the verification algorithms will be introduced.
- Design and implementation of new verification algorithms; the new algorithms will check new semantic properties of Mscs as well as syntax correctness required by other algorithms.
- Design and implementation of an algorithm for exporting Mscs to to the DiVinE modelling language [1]; this will allow model checking of Msc-specifications.

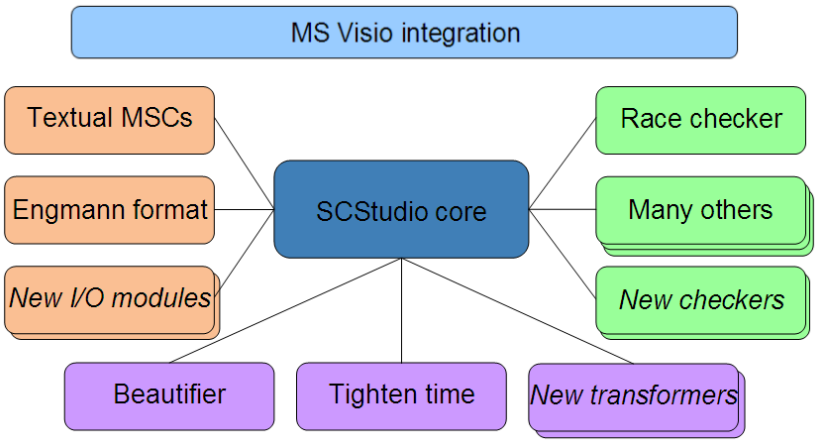

Figure 1.7: SCStudio architecture

1.4 Structure of the Thesis

The next chapter describes new or modified functions that are not directly used by users and are rather integrated into other components. This comprises new Msc-traversal functions, handling of counterexamples etc.

Chapter 3 deals with the checkers already implemented in SCStudio. Modifications to their definitions as well as the implementation are discussed.

Chapter 4 introduces checkers newly implemented: for each checker the definition is given and the implementation is outlined.

Chapter 5 describes the DiVinE-export functionality: first, the DiVinE language is briefly introduced and then the exporting algorithm is described.

Some sections contain a part called *Implementation*. These parts describe important points of the implementation in SCStudio and require the knowledge of the SCStudio architecture and traversing concepts (see [4] for more information).

Chapter 2

Generic Concepts

The functions described in this chapter do not directly influence the end-user experience. They have been developed to simplify the implementation of other algorithms. The functions are used to build up some of the checkers described in the following chapters, for example. Because the functions could be needed in the future, they have been implemented separately and may be easily reused.

The first section introduces a *new traverser* capable of finding *elementary cycles* in an HMsc. The second section describes the implementation of the *communication graph* as defined in [10]. The third section depicts functions for finding neighboring nodes and the final section deals with promoting BMsc checkers to the HMsc level.

2.1 Enumeration of Elementary Cycles

As using cycles is the only correct way of introducing an infinite behavior in an HMsc, a tool for finding cycles is often useful. In what follows, we first give the definition of and elementary cycle, then describe the usage of the implemented class and finally describe its implementation.

Definition 2.1.1. *Elementary cycle* in an HMsc is a series of nodes $a_1, a_2, \ldots, a_n, a_1$ such that a_i is a successor of a_{i-1} for $i = 2, \ldots, n$, a_1 is a successor of a_n and $a_i \neq a_j$, $i, j = 1, \ldots, n$. The nodes belong to the HMsc or some of the referenced HMscs.

An HMsc is interpreted as a graph using the *graph-based interpretation* (see Section 1.1.3). Notice a cycle in an HMsc may be absent when interpreted using the execution-based interpretation even though it is present using the graph-based interpretation as shown in Figure 2.1.

When finding cycles using the *execution-based* intepretation is needed, it is necessary to to transform the HMsc to an equivalent BMsc graph first. SCStudio contains BMscGraphDuplicator for that purpose.

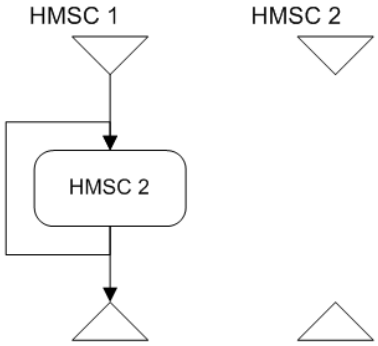

Figure 2.1: Cycle only in the graph-based interpretation

Using the Finder

The interface of the cycle finder is aligned with the interface of DFS* traversers. The name of the class is ElementaryCyclesTraverser. The name of the header file to be included is data/elementary_cycles_traverser.h. The following procedures are available:

add_cycle_listener: adds a new listener. When the traversr finds an elementary cycle, the method on_elementary_cycle_found of all listeners is called.

remove_cycle_listeners: the procedure removes all the previously stored listeners.

traverse: starts the process of finding elementary cycles.

cleanup_traversing_attributes: performs a cleanup. It is only necessary if the traverser is interrupted prematurely.

enable_restriction: when the restriction is enabled, the traverser finds only cycles composed solely of nodes with the dynamic attribute of the given name set. The attribute cannot be the empty string. The restriction is used, for example, in the deadlock checker (see Section 3.3).

disable_restriction: disables a previously enabled restriction.

is_restriction_enabled: returns the status of the restriction.

It is also necessary to describe the base class of elementary cycles listeners. The name of the class is ElementaryCyclesListener and contains a single abstract method called on_elementary_cycle_found. It takes one parameter – a constant reference to a cycle. The cycle is of type MscElementPList and contains the nodes and also the relations between them making up a cycle. The first node of each cycle is present also as the last.

Implementation

The traverser is based on the standard Tarjan's algorithm for enumerating elementary circuit as described in [16]. Because it was necessary to adapt the algorithm to the HMsc structure, it is beneficial to briefly describe both the algorithm and the modifications.

The algorithm works as follows:

1. The vertices of the examined graph are numbered from 1 to n. In the following text nodes are represented by those numbers.

2. The algorithms performs n steps. In step i, only cycles containing i as the smallest node are found. This assures that each cycle is found only once. Step i is realized by running a backtracking procedure from i.

3. The backtracking procedure is similar to the depth-first search. The rules for entering nodes are slightly different, though. In step i, the procedure never enters a node less than i. Traversing is further driven by a so called *marked stack*. The procedure does not enter a node if it is on the marked stack. A node is put on the marked stack as soon as it is entered and is removed when a new cycle is found. This assures that the algorithm is efficient; the information that there is no cycle containing a node v may be reused throughout the run of the backtracking procedure because v is kept on the marked stack. In addition the path from i to the current node (it is called DFS stack in [16]) is stored for the purpose of outputting a cycle, if found.

   The backtracking procedure proceeds in the following way for vertex v when initially run for vertex s:

   (a) Put v on the marked stack.

   (b) If s is a successor of v, output the elementary cycle from s to s.

   (c) Run the backtracking procedure for all successors of v greater than or equal to v.

(d) If an elementary cycle was found in step 2 or deeper in the recursive call sequence, remove all the vertices from the top of the marked stack up to v including v.

Please refer to [16] for further details.

The following points had to be carried out in order to use the algorithm for HMscs:

1. DFSHMscTraverser is used to assign a number to each node.

2. Node relations are stored along with nodes on the DFS stack.

3. For running the backtracking procedure from each node, DFSHMscTraverser is used again; the backtracking procedure is called whenever a white node is found.

4. When DFSHMscTraverser finds a restricted node, it is skipped. When the backtracking procedure finds a restricted successor, it is ignored (i.e. an edge from a node to its restricted successor is ignored).

5. The backtracking procedure does not follow references in reference nodes; DFSHMscTraverser does. This assures that cycles in all referenced HMscs are found.

2.2 Communication Graph

The communication graph class is a new class for working with communication graphs introduced in [10]. For a given BMsc, the communication graph contains the information *who communicates with whom*.

Definition 2.2.1. Let B be a BMsc. The *communication graph* of B is a directed graph $G = (V, E)$ where

- V is the set of instances in B and
- there is and edge from i to j if instance i sends a message to j.

For certain purposes it is useful to define the graph as weighted. Then the weight of an edge (i, j) represents the number of messages sent by i to j.

Functionality

The class is located in files communication_graph.h and .cpp and is a part of the scpseudocode library. The class stores and manipulates the communication graph defined above.

Vertices are identified by *nonnegative integers*. The mapping between names of instances and numbers differs in different methods. Users may choose the mapping that suites their purpose the best. The following procedures are available:

Constructor: when called without parameters, a new empty communication graph is created. When called with a BMsc as a parameter, the communication graph is created from the BMsc in the same way as in the create_from_bmsc procedure.

create_from_bmsc: the communication graph is created from the given BMsc. The current data stored in the communication graph is discarded. Instances are numbered using the lexicographical order of their names starting with zero.

create_from_bmsc_with_param: the communication graph is created from the given BMsc. The current data is discarded. Instances are numbered using numbers stored by the user in a dynamic attribute. The name of the attribute is passed to the procedure as a parameter. The procedure does not modify the parameter. If a size of the graph greater than the number of instances in the BMsc is required (e.g. for the purpose of merging), it can be specified using the third parameter.

It is the user's responsibility to assure that the indexing falls within the range of the graph. If not, a standard out-of-range exception is thrown.

create_from_bmsc_with_map: this procedure is very similar to the previous one. Only the indexing is defined using a map mapping instance names to numbers. Again, the validity of the map is not checked.

merge: the procedure merges the communication graph with the one passed as a parameter. If the sizes of the two graphs differ, a runtime error is thrown. The result represents the communication graph of the concatenation of the BMscs represented by the communication graphs on the input.

is_strongly_connected: the procedure returns *true* iff the communication graph is *locally* strongly connected. The graph is locally strongly connected if each component of the graph is strongly connected.

get_graph: returns a constant reference to the communication graph.

Implementation

The communication graph is stored in the form of an adjacency matrix. The matrix is represented by a vector of vectors of unsigned integers. The value stored at position (i, j) represents the number of messages sent by i to j. The mapping between instances and indexes may be specified by the user.

The creation of the communication graph with indexes of instances stored in dynamic attributes of instances uses DFSEventsTraverser – whenever a *matched send event* is found, the value at position (i, j), where i is the index of the sending instances and j is the index of the receiving instance, is incremented by one.

The other two methods of creation (with a map and from the lexical order) make use of the first method; first, indexes are stored in dynamic attributes. Then the first method is run and finally the attributes are removed.

Merging of communication graphs is done by adding values of one graphs to the values stored in the second.

Checking whether the communication graph is strongly connected uses the standard Tarjan's algorithm described in [15].

2.3 Node Finder

Node finder is capable of finding successors or predecessors of a given HMsc node. The particular behavior may be specified by the user. By default, the finder ignores connection nodes and finds other nodes (reference, condition, start or end nodes) reachable only via connection nodes. The user may specify another condition: empty reference nodes may be additionally ignored, for example.

Functionality

NodeFinder provides these methods:

Constructor: the name of the coloring attribute used by the underlying traverser may be optionally specified. This is useful if more NodeFinders are used simultaneously.

find_successors: for a given HMsc finds nearest (possibly indirect) successors of a node which satisfy the is_terminal() condition. ConnectionNodes are *always skipped* and references in reference nodes are not fol-

lowed. Keep in mind that it is necessary to call cleanup_traversing_attributes() if the finder is reused.

find_predecessors: dual function to find_successors.

is_terminal: this function is a predicate which, for a given node, determines if the node is to be passed through or if the node is terminal and no paths from it should be followed. By default, only connection nodes are passed through and all other nodes are terminal. This behavior can be changed by deriving a new class from NodeFinder and overriding this virtual method.

get_result, get_skipped: returns the set of terminal nodes and nodes that were passed-through, respectively, computed during the last run of get_successors or get_predecessors. Connection nodes are always ignored and are not returned.

successors, predecessors: static methods providing the same functionality as get_$*$ methods above without instantiating the class. For derived classes, it is not possible to use the static method.

Implementation

SCStudio contained a class called ReferenceNodeFinder. For a given HMsc node, it was capable of finding its successors reachable along paths containing only *connection nodes*. The class uses a modification of DFS; when a white reference node is found, it is added to the list of results and marked black. Its successors are not processed. This assures that no node appears more that once in the result. Three modifications have been made to the class (in addition to renaming to NodeFinder, which reflects its functionality better):

- A bug preventing NodeFinder from finding the initial node in the case it is reachable from itself has been fixed; the initial node is marked gray in order that is not further processed when found. A dummy dynamic attribute is set at the same time. Then, when a gray node with that attribute is found, the initial node is added to the list of results and the dynamic attribute is removed. This assures that neither the initial node is found more than once.

- NodeFinder has been improved such that it is capable of finding successors as well as predecessors.

- A configuration has been enabled. It is possible to specify at which nodes searching for successors should terminate and which nodes should be only passed through. When a node does not satisfy the is_terminal condition, it is considered a connection node and searching continues to its successors.

2.4 HMsc Reference Checker

SCStudio already contained a class for promoting BMsc checkers to the HMsc level. The promotion works as follows – an HMsc violates a BMsc property iff a BMsc referenced by the HMsc violates the property. The counterexample consists of the HMsc path from the start node to the violating reference node and the BMsc counterexample returned by the BMsc checker.

Having a BMsc checker B, one can simply create a HMsc version H by writing

```
class H: public HMscReferenceChecker<B>;
```

The HMsc and BMsc checker may be even contained in a single class:

```
class BH: public HMscReferenceChecker<BH>, ...
{
  definition of the BMsc checker
};
```

Similarly to other checkers, HMscReferenceChecker was capable of finding only a single counter example. Therefore, a modification has been made to enable returning multiple counter examples. The algorithm now works as follows:

- The HMsc is traverserd using DFSHMscTraverser (therefore, each referenced Msc is traversed only once, no matter how many times it is referenced).
- For each referenced BMsc, the BMsc checker is run and if it returns a non-empty list of counterexamples, the list is stored in a dynamic attribute of the reference node referencing the BMsc.
- The HMsc is traversed using DFSHMscTraverser again. When a reference node with a list of counter examples is found, the path from the start node to the reference node is duplicated for each entry in the list of counter examples. For each such a path, one BMsc counter example is set to be referenced by the last reference node of the path. Finally, the HMsc list of counter examples is made of all the paths computed in the previous step.

Chapter 3

Existing Checkers Revisited

This chapter describes changes to the five checkers already implemented in SCStudio and described in [4]. Their implementation has been improved and in some cases, definitions have been refined as well. The first two checkers check properties of BMscs and the remaining three are destined for HMscs.

3.1 Acyclic Checker

The first of the existing checkers is *acyclic checker*. It is a BMsc checker and checks whether a given BMsc contains a cycle of events. That is not acceptable because it would require events occuring after some event to occur before the event, which is a non-sense.

Definition

Before defining the acyclic property, we have to define the *visual order* of a BMsc.

Definition 3.1.1. Let a, b be events in a BMsc. Then the *visual order* of the BMsc (denoted as $<$) is the smallest relation on events in the BMsc satisfying the following conditions:

- $a < b$ if a is a send event, b is a receive event and the events share a common message.
- $a < b$ if both the events belong to the same instance, the events do not belong to the same coregion and a is placed higher on the instance.
- $a < b$ if the events belong to the same coregion, say C, and $a <_C b$ where $<_C$ is the general ordering of the coregion C.
- $<$ is transitive.

Remark. When a BMsc is cyclic, its visual order is not an order because no cycles are allowed in an order.

Now we are ready to define the *acyclic property*.

Definition 3.1.2. A BMsc is *acyclic* iff the visual order of the BMsc has no nontrivial cycles.

Please notice the definition differs from the definition in [4]: a BMsc is said to be acyclic if the visual order is *antisymmetric* in [4]. In the following, we show the two definitions are equivalent.

Lemma 3.1.3. *The visual order $<$ of a BMsc has no nontrivial cycles iff it is antisymmetric.*

Proof.

- "$\Rightarrow$": By contradiction. Let $<$ has no nontrivial cycles and $<$ be not antisymmetric. Then there are events $a, b, a \neq b$ such that $a < b$ and $b < a$. But then a, b, a is a nontrivial cycle, which is a contradiction.
- "$\Leftarrow$": By contradiction. Let $<$ be antisymmetric and let $<$ contain a nontrivial cycle. Then the cycle has the form $a, b, \ldots, z, a$. Because $<$ is transitive, $a < z$ and also $z < a$. As the cycle is nontrivial, $a \neq z$ and thus $<$ is not antisymmetric, which is a contradiction.

□

Before describing the implementation, we have to prove one more claim. The visual order is defined to be transitive but in practice the transitive and reflexive closure is not computed. As shown below, it is not needed and it only increases the complexity of the algorithm.

Lemma 3.1.4. *Let $<$ be an arbitrary relation and $<_T$ be a transitive closure of $<$. Then $<$ contains a nontrivial cycle iff $<_T$ contains a nontrivial cycle.*

Proof.

- "$\Rightarrow$": Let $<$ contain a nontrivial cycle. Since $<\subseteq<_T$, $<_T$ also contains a nontrivial cycle.
- "$\Leftarrow$": Let $<_T$ contain a nontrivial cycle of the form $a_1 <_T a_2 <_T \cdots <_T a_n <_T a_1$, where $a_i = a_j \Rightarrow i = j$. Due to transitivity of $<_T$ it also holds that $a_1 <_T a_2 <_T a_1$, where $a_1 \neq a_2$. By the definition of the transitive closure, there are $a_{11}, \ldots, a_{1k}, a_{21}, \ldots a_{2l}$ such that $a_1 < a_{11} < \cdots < a_{1k} < a_2 < a_{21} < \cdots < a_{2l} < a_1$ and therefore $<$ contains a nontrivial cycle.

□

Each counterexample has the form of a BMsc with some events and some messages marked. The marked elements together form one cycle in the checked BMsc. A user can trace the cycle in the following manner:

- Start at any marked event.
- If it is a send event and the corresponding message is marked, continue to the corresponding receive event.
- If the event is placed in a coregion and it has a marked successor (with respect to the coregion's general ordering), continue to that successor[1].
- Otherwise, continue to the next event on the instance.
- Repeat until the start event is reached.

Notice that the acyclic property is strictly defined as a *graph property* and no logic behind the BMsc execution is applied. As a consequence, multiple results may be generated from a single artifact in a BMsc. Please see Figure 3.1. Even though the BMsc contains only one erroneous message (4), there are two elementary cycles in the BMsc: $1, 2, 3, 4, 1$ and $1, 4, 1$.

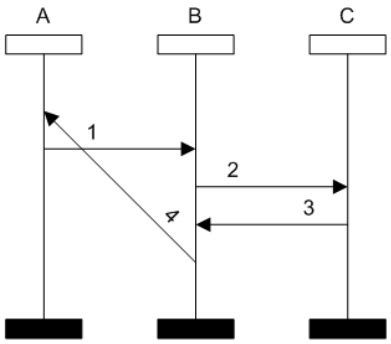

Figure 3.1: Acyclic checker – superfluous counterexamples

1. Please notice that this step may be ambiguous. It results from the fact general ordering lines cannot be marked. Anyway, examples of the ambiguity are mostly synthetic.

Implementation

The checker had to be completely rewritten in order that it finds all the results (i.e. all nontrivial elementary cycles) in a given BMsc. Finding an arbitrary cycle in a graph is significantly simpler than finding all cycles; a simple depth-first search can be used for the first problem but a more powerful (and more complex as there may be exponentially many cycles in a graph) mechanism is needed for the second.

Since there is a fully functional and tested HMsc circuit enumerator (see Section 2.1, we decided to use it for finding cycles in a BMsc.

The checker consists of three major parts:

1. Transforming a BMsc into an "equivalent" HMsc: This part makes use of DFSEventsTraverser. In the first pass of the BMsc, every event is assigned a new ConnectionNode. The reference to the HMsc nodes are stored in a dynamic attribute. At the same time, references to the original BMsc events are stored in connection nodes' dynamic attributes. The new nodes are also added to a new HMsc.

   Functions on_send_receive_pair and on_event_successor events supplied by the traverser are used to create relations between the new connection nodes in the second pass. When a relation is created as a result of a message in the BMsc, it is assigned a reference to the original message in the form of a dynamic attribute. This is necessary for proper result highlighting later.

   At this point, there is a new HMsc created reflecting the structure of the BMsc. It only contains connection nodes. In a HMsc a start node is required to be present. StartNode is created and all minimal events at all instances of the BMsc are set to be its successors. A BMsc and the HMsc resulting from the above described transformation are shown in Figure 3.2.

2. Finding cycles in the resulting HMsc: Finding all the cycles in the HMsc is simple because ElementaryCyclesTraverser (see Section 2.1) is used. The listener just stores all the elementary cycles in a list.

3. Mapping the cycles back to the original BMsc: Whenever the list of cycles is not empty, the BMsc is *not acyclic* and counterexamples are generated in the following way:

   For each elementary cycle:

   - Create a new duplicate of the BMsc.

- Highlight all the events referenced by the connection nodes in the cycle.
- Highlight all the messages referenced by NodeRelations in the cycle.
- Add the highlighted duplicate to the list of results.

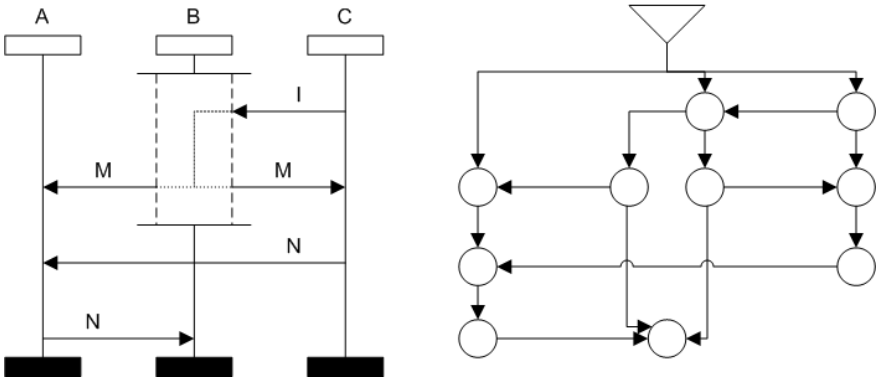

Figure 3.2: BMsc transformation

3.2 FIFO Checker

FIFO is a property of the *visual order* $<$ (see Definition 3.1.1) of a BMsc. A BMsc is FIFO if its messages do not overtake each other – the order in which they are sent must be the same as the order or their reception. In coregions the order of events is not defined in general. Then the FIFO property ensures that the behavior described by an Msc is realizable in an FIFO environment[2]. An example of a non-FIFO BMsc is shown in Figure 3.3.

In some cases, overtaking of messages is acceptable, for example when the messages are sent using different protocols and do not share a common receive-buffer. Therefore a notion of *channels* has been introduced. Then a BMsc is non-FIFO only if messages from the same channel overtake each other. Usually, messages belong to the same channel if their sender is the same and the receiver is the same. In addition, the label of the messages may be considered.

2. An environment in which messages cannot overtake each other during transmission

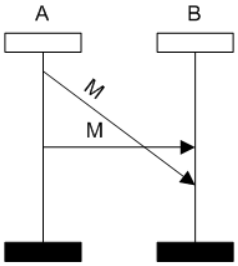

Figure 3.3: Non-FIFO BMsc

Definition

First, the definition will be given and then it will be compared with the old one.

Definition 3.2.1. Let B be a BMsc and $<$ its corresponding visual order. B is *FIFO* iff for all receive events c, d and their matching send events a, b, respectively, where the two arising messages belong to the same channel, it holds that

$$c < d \Rightarrow a < b$$

The definition in [4] is too loose because it does not distinguish send and receive events. The former implementation does, though. However, the role of send and receive events is reversed with respect to the new definition. Therefore BMsc 1 in Figure 3.4 was marked as non-FIFO, which is incorrect; the behavior described by the BMsc can be implemented in a FIFO environment; whenever send events are ordered and receive events are not, the correct (FIFO) order is naturally enforced by the environment. The opposite case (BMsc 2 in Figure 3.4) was considered FIFO by the former implementation. This is also incorrect; since send events occur before receive events, the environment can in no way enforce the order of the send events with respect to the receive events. Thus, the execution may lead to a deadlock. This BMsc is non-FIFO by the new definition.

Implementation

No major changes to the implementation had to be made. The algorithm for computing the visual order has been left intact. The condition on the relation has been modified to reflect the new definition of FIFO. Finally, the original algorithm throws an exception after finding the first pair of events

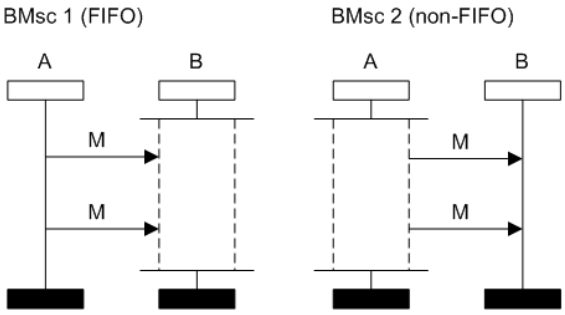

Figure 3.4: FIFO and non-FIFO BMscs

not satisfying the condition. This behavior has been changed; for each such pair a counterexample is generated and stored. After checking the whole relation, the list of all counterexamples (possibly empty) is returned.

3.3 Deadlock Checker

Let's now move on to the *HMsc checkers*. The first, and the most basic one, is the *deadlock checker*. Deadlock is a very common term and may refer to various situations. In SCStudio, deadlock occurs when a reference node is reached and it is impossible to continue from then. An example of a deadlock is shown in Figure 3.5. Even though an execution of an Msc might hang up for other reasons, those are *not called* deadlock in SCStudio.

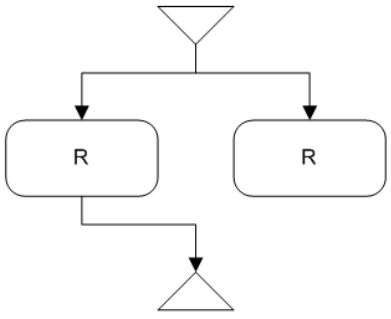

Figure 3.5: Deadlock example

Definition

The definition from [4] has been transformed because it has shown to be incorrect. The differences are discussed later, let us first give the new definition proper.

Definition 3.3.1. Let H be a HMsc and a, b nodes of H or some HMsc referenced by H. Then b is *flat-reachable* from a iff there exists a path $a = a_1, a_2, \ldots, a_i = b$ such that a_j is a successor of a_{j-1} for $j = 2, \ldots, i$.

b is *reachable* from a iff there exist nodes $a = a_1, a_2, \ldots, a_i = b$ such that a_j is flat-reachable from a_{j-1} or a_j is the start node of the HMsc referenced by a_{j-1} for $j = 2, \ldots, i$.

Definition 3.3.2. An HMsc with a start node s contains a *deadlock* iff there is a reference node r *reachable* from s such that there are no reference nodes or end nodes *flat-reachable* from r. Then r is called a *deadlocked*.

Notice he fact a reference node is deadlocked does not depend on the Msc referenced by the reference node. Intuitively, a reference node r is deadlocked if it is not possible to continue with the execution of the HMsc after executing the Msc referenced by r. It is, however, *not* guaranteed that the referenced Msc reaches its end – it may contain another deadlock[3].

Since our algorithm independently finds all erroneous nodes, the above described approach is viable – a user is given precisely the points to be corrected in order that his/her HMsc is deadlock-free. Marking a reference node as deadlocked in the case the problem only lies somewhere in the referenced Msc might be confusing.

The definition of deadlock in [4] uses the notion of *paths*. The existence of a path between a pair of nodes corresponds to the above defined reachability, i.e. references may be followed. A reference node is deadlocked by the old definition if there is no path from the node to a node different from the connection node. This has shown to be incorrect. An example of a deadlock-free HMsc by the old definition is shown in Figure 3.5. Let us assume R is a simple HMsc containing only the start node and the end node. The right node referencing R is not deadlocked because there is a path from the node to the start node of R. No node in R is deadlocked either since there exists a path to the end node of the root HMsc – this follows from the fact that R is also referenced by the left reference node.

The new definition correctly finds the deadlock in Figure 3.5 and highlights the right reference node as deadlocked.

3. or a livelock defined in Section 3.4

Implementation

The deadlock checker has been rewritten completely. The main reason is the change of the definition. Also, the old implementation was not suitable for finding more than one result.

The new implementation is very simple as it makes use of NodeFinder (see Section 2.3). The checked HMsc is traversed using DFSHMscTraverser. Successors of each white reference node are found using NodeFinder. Whenever a reference node does not have any reference-node or end-node successors, it is marked as deadlocked. Finally, a list of paths to each deadlocked node is returned as the result.

3.4 Livelock Checker

The second HMsc checker we describe is the *livelock checker*. Livelock closely relates to deadlock because both the properties represent a situation when the end node of a HMsc is not reachable. In case of deadlock the execution is stopped and no actions are performed while an infinite loop of actions with no possible end is performed in case of livelock.

Definition 3.4.1. An HMsc with a start node s contains a *livelock* iff there exists an elementary cycle reachable from s containing at least one reference node and no end node is reachable from the cycle.

In an HMsc containing a livelock, it is possible that an infinite loop is entered; even though the processing of the HMsc continues, no end node can ever be reached. A simple example of a livelock is shown in Figure 3.6.

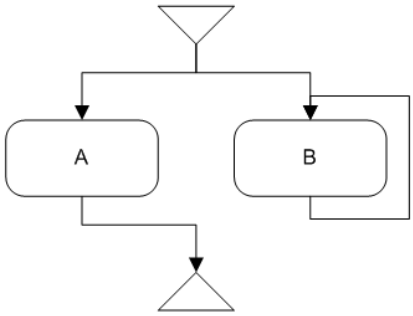

Figure 3.6: Example of livelock

Similarly to the the deadlock property, the content of Mscs referenced by reference nodes forming the livelocking cycle is not considered – if some of the referenced Mscs contains, for example, another livelock or a deadlock, it is possible that the livelocking cycle is not processed infinitely many times. The same argument as for the deadlock checker applies also here; a livelock found by the checker indicates a point in the given HMsc to be modified in order that the whole HMsc is correct.

The above given definition is only a reformulation of the definition in [4].

Implementation

The livelock checker has been rewritten because finding multiple results requires a completely different approach. The former gray-node approach finds only a single cycle.

The goal is to find all livelocking cycles in a given HMsc. Because all cycles are made up of elementary cycles, it is sufficient and efficient to find only all elementary cycles.

The implementation is very straightforward as it makes use of existing components. The following steps are performed by the checker:

1. The HMsc is traversed using DFSHMscTraverser. For each visited node a dynamic attribute is set to indicate that the node is reachable from the start node.

2. The HMsc and each HMsc referenced by a node reachable from the start node are traversed using DFSBHmscTraverser. It traverses HMscs backwards (from the end node) and does not follow references. For each visited node the dynamic attribute set in step 1 is removed. After the traversal is finished, only nodes reachable from the start node from which the corresponding end node is not reachable have the attribute set.

3. The HMsc and each HMsc referenced by a node reachable from the start node are traversed using ElementaryCyclesTraverser. The traverser is configured to restrict the cycle enumeration only to nodes having the dynamic attribute used in the previous steps set (see Section 2.1). When a cycle is found, the corresponding listener checks if the cycle contains a reference node. If yes, a livelocking cycle is found and stored as a dynamic attribute of the first of its nodes.

4. The stored livelocking cycles are found using DFSHMscTraverser and for each cycle the path to the cycle and the cycle itself is used to produce

one countexample. The list of all counterexamples is the result of the algorithm.

3.5 Race Checker

Race condition is a property of both HMscs and BMscs. Informally, race condition occurs if a pair of events is drawn in some order in an Msc, but the events may appear in the opposite order when the Msc is executed. The pair of messages is then said to be *in race*. The most typical example of a race condition is shown in Figure 3.7.

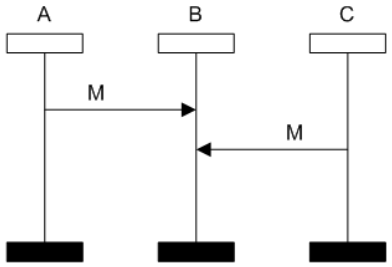

Figure 3.7: Typical race condition

Definition

The following definitions are taken from [6] for a quick reference. First, we have to define a *causal order* of a BMsc.

Definition 3.5.1. Given a BMsc B with an acyclic and FIFO visual order $<$ we define a *causal order* $\ll$ as the least partial order on the event of B such that $e \ll f$, if

- e and f are the send and the receive event of a message, or
- e and f are on the same instance, $e < f$ and f is a send event, or
- e and f are both receive events on the same instance and their corresponding send events e' and f' also share a common instance and $e' \ll f'$.

Now we can define the race condition for BMscs:

Definition 3.5.2. If and BMsc contains some events e, f such that $e < f$ and $e \not\ll f$, we say that the BMsc contains a *race (between events e, f)*. The BMsc is said to be race-free otherwise.

Finally, the trace race condition for BMsc graphs is defined in the following way:

Definition 3.5.3. A BMsc graph contains a *trace race* if there is a run through the BMsc graph such that the concatenation of all BMscs reached in the run contains a race. We say that the BMsc graph is *trace-race-free* otherwise.

Remark. In this thesis, the terms *race* and *trace race* are used interchangeably and both refer to the *trace race*.

Please refer to [6] or to Appendix A for the formal definitions.

Footprint Algorithm

The HMsc-algorithm newly implemented in SCStudio a slight modification of the algorithm presented in [6]. The original algorithm has been introduced in SCStudio as described in [4]. While the original algorithm halts after finding the first pair of messages in race, the new algorithm continues and finds additional events in race, if any.

The proof of correctness had to be also updated a result. The updated proof can be found in Appendix A.

Limits of the Footprint Algorithm

The main benefit of the footprint algorithm is the fact it only uses *local information*. When a new BMsc is concatenated, no data is cumulated; majority of data is rather replaced with new one.

This is, however, in contradiction to finding all events in race in an HMsc. In the HMsc in Figure 3.8, for example, all receive events in A are in race with all receive events in B. Notice that events of instance A are not mutually in race; therefore, to find all the events in race, it would be necessary to store all the messages instead of a small subset of them.

As a result, the idea of finding all events in race has been abandoned. Anyway, the algorithm has been improved so that it finds at least the race condition between maximal and minimal events in neighboring BMscs.

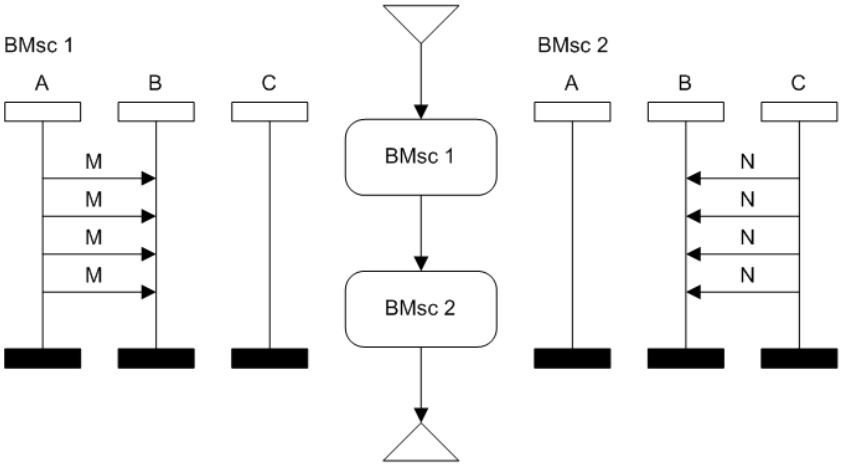

Figure 3.8: All receive events in race

Implementation

The race checking algorithm consists of two parts. In the next part, each of them is described together with the modifications made to it.

- BMsc race checker – for a given BMsc, the algorithm computes relations $<$ and $\ll$ as defined above. By comparing the relations, pairs of messages in race are found. The original algorithm stopped after finding the first pair of messages in race. Thus, a slight modification has been made in order that all such pairs are found. For each pair, a duplicate BMsc is created and the pair of events is highlighted. The algorithm then returns the list of counterexamples containing *all pairs of messages in race* in the given BMsc.

- HMsc race checker – first, the algorithm transforms a given HMsc into an equivalent BMsc graph. Then, the BMsc race checker is run for each referenced BMsc. The original algorithm stopped completely after finding the first BMsc containing a race condition. In the new implementation, a path to each such BMsc is stored and highlighted and the algorithm continues. Then, maximal and minimal events are computed for each BMsc. After that, footprints for each BMsc are iteratively updated as described in [6]. This part has not been modified. When a new footprint is computed, the race condition is checked. The original algorithm stopped after finding the first pair of messages in

race. This behavior has been changed; now, the algorithm continues until no footprint can be further updated. This modification is correct thanks to the updated proof above.

Chapter 4

New Checkers

In this section, five new checkers verifying new properties are described. Four of the checkers are HMsc checkers; only the name checker is applicable for both the Msc forms.

4.1 Recursivity Checker

The recursivity checker checks if a given HMsc is recursive, i.e. a reference node references itself directly or indirectly (see Figure 4.1).

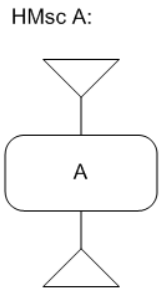

Figure 4.1: Recursive HMsc

Definition 4.1.1. An HMsc is *recursive* iff there exists a reference node a in the HMsc referencing an HMsc H such that a is reachable from the start node of H.

Recursive HMscs are considered erroneous; infinite behavior should be introduced using cycles, not recursion. From the practical point of view, recursive Mscs cannot be correctly processed by checkers that use BMscGraphDuplicator because it unfolds the given HMsc, which is not possible for recursive Mscs.

Implementation

The implementation is very straightforward. DFSHMscTraverser is used for traversing the checked HMsc. When a gray start node is found, the HMsc is recursive. The checker currently returns only one occurrence of recursivity.

What remains is to show that the described implementation is correct.

Theorem 4.1.2. *An HMsc is* recursive *iff DFSHMscTraverser finds a gray start node.*

Proof. Before the proof proper, let us discuss the behavior of DFSHMscTraverser. It traverses a given HMsc in the depth-first manner. The coloring of nodes is *global*. This means that colors of nodes of an HMsc are shared by all occurrences of that HMsc. Now we can proceed with the proof.

- "$\Rightarrow$": Let HMsc H be recursive. Then there is a reachable reference node a in an HMsc I which references an HMsc J such that a is reachable from the start node of J. Let us assume a is the first node found by DFSHMscTraverser satisfying the condition.

  When DFSHMscTraverser reaches a, it first follows the reference (HMsc J in this case). We show that a gray start node is found in J or some Msc referenced by J by induction on the number n of references on the path from the start node of J to a. Let us assume, without loss of generality, that the first reference node (with respect to the order the traverser finds nodes) through which a path to a exists is used in each HMsc on the path.

  - Base case $n = 0$: there are no references on the path from the start node of J to a. Then necessarily $I = J$. Because a is just reached by DFSHMscTraverser, the start node of I is gray. Following the reference in a, DFSHMscTraverser immediately finds a gray start node.
  - Inductive step $n = i + 1$: in this case, the start node of J is either white or gray (it cannot be black because a is the first node found by the traverser satisfying the condition of recursivity). If it is gray, we are done. If it is white, the traverser enters J. We know there is a node x in J from which the path to a continues via a reference. Then a gray start node is found in some Msc referenced by x because the number of references on the path from the start node of the HMsc referenced by x to a is smaller than $i + 1$ and the induction hypothesis may be used.

- "⇐": DFSHMscTraverser finds a gray start node only if there is a cycle containing the start node. A start node is only reachable via reference; there can be no edge directing to a start node. Let us assume that DFSHMscTraverser has found a gray start node s of an HMsc M. Then there is a cycle containing s. As described above, there must be a reference involved in the cycle. Therefore, a node a in M referencing some other HMsc H must be also a part of the cycle. This node a is exactly the node a from the definition of recursivity.

□

4.2 Universal Boundedness Checker

Universal boundedness is an interesting property of HMscs. Informally, when an HMsc is universally bounded, it is assured that no buffer of incoming messages in any instance can grow infinitely, regardless of the behavior of instances – in the computer-network wold, the situation when the IP dispatcher works but the application processing received messages hangs up, is a suitable example.

A simple real-world example of bounded communication is shown in Figure 4.2. It models a mail conversation between friends. If either of the friends dies, his or her mailbox will not get congested because the other friend will wait for a response before sending another letter (infinitely long).

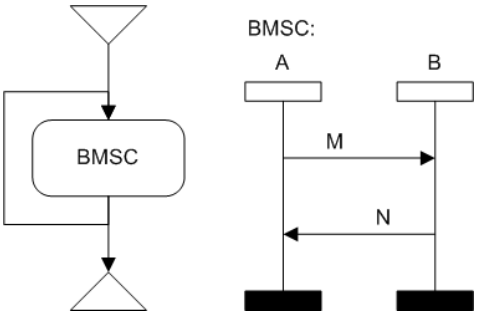

Figure 4.2: Bounded communication

The simplest example of unbounded communication is shown in Figure 4.3. Since advertisement producers do not wait for any response, not checking a mailbox eventually leads to its congestion. This phenomenon is widely recognized both in electronic and paper communication.

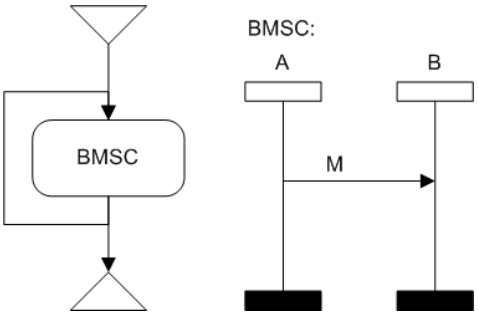

Figure 4.3: Unbounded communication

Definition 4.2.1. A BMsc graph G is *universally bounded* iff there exists a positive integer b such that for every execution of G the number of pending messages in all receive buffers is lesser than b all the time.

A nonrecursive HMsc H is universally bounded if the BMsc graph G equivalent to H is universally bounded.

A more formal definition may be found in [10]; it is, however, not necessary to present it in this thesis.

Implementation

The definition is not suitable for direct implementation. In [11] it was shown that a BMsc graph G is universally bounded if and only if for each elementary cycle c in G holds that the BMsc representing the concatenation of all the BMscs referenced by reference nodes in c has *locally strongly connected* communication graph. This result is used in the implementation in SCStudio.

For an HMsc H the following steps are performed:

1. H is transformed into the equivalent BMsc graph G using BMscGraphDuplicator.

2. ElementaryCyclesTraverser (see Section 2.1) is used to find all cycles in G. For each cycle

   (a) the communication graph of the concatenation of its BMscs is computed using CommunicationGraph (see Section 2.2) and

   (b) it is checked if the communication graph is locally strongly connected. This is completely carried out by CommunicationGraph.

(c) If the cycle violates the condition of universal boundedness, the list of its elements is stored in a dynamic attribute of its first node.

3. All the stored violating cycles are found using DFSHMscTraverser and for each such cycle a counterexample consisting of the cycle and the path from the start node to the cycle is created.

4. Finally, the list of all counterexamples is returned.

Related Definitions

While we use the definition of universal boundedness from [10], there are similar properties differing in the condition on the communication graph of elementary cycles. While we require the graph to be *locally strongly connected*, it may also be required to be *strongly connected*. Our condition only assures the property of message buffers. The other definition then, in addition, assures that an HMsc is *regular* [12].

The HMsc in Figure 4.4, for example, is universally bounded, but not regular. Groups A, B and C, D are not synchronized and thus they may go through the cycle at different speeds. When the execution ends, the number of messages sent within the two groups is required to be the same. However, that cannot be guaranteed in a realization.

Another example is *global cooperativeness* – the communication graph is required to be *weakly connected* in this case. These properties may be easily implemented in SCStudio, if needed. Majority of the code for the universal boundedness checker may be used; only the condition on the communication graph is to be modified.

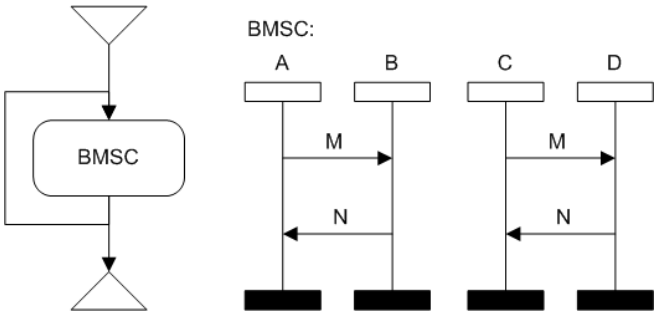

Figure 4.4: Not cooperative bounded HMsc

4.3 Local Choice Checker

Another property newly introduced to SCStudio is the *local choice*. This section is based on [5]. While a brief excerpt of the thesis is presented in the following text, details may be found in the original work.

Local choice is a property of HMscs. (In [5], the property is defined for BMsc graphs. It can be easily extended by transforming an HMsc to the equivalent BMsc graph.) *Non-local choice* occurs if an HMsc node has more than one successor and it is possible that the communication is initiated by different instances in different branches. Such a situation is shown in Figure 4.5. Non-local choice is a flaw in an HMsc: because instances don't share any memory and communicate exclusively via messages in the HMsc, it may happen that the initiating instances decide to use different branches. That behavior is, however, not intended as it is not specified by the HMsc: the HMsc specifies that either of the possible branches is used. In reality, this cannot be fulfilled.

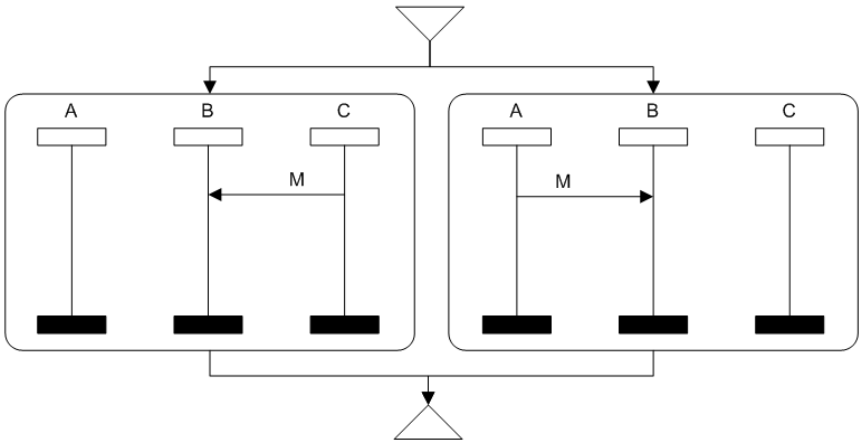

Figure 4.5: Non-local HMsc

Definition

The formal definition is not presented here and may be found in [5]. Informally, an HMsc node is called *local* if it has at most one successor or there is at most one instance initiating the communication in all branches leading from the node. Otherwise, the node is called *non-local*. Notice that the initial

message does not necessarily have to be in the direct successor of the node. An HMsc violates the non-local choice if there is at least one non-local node.

Implementation

The algorithm presented in [5] is used. Due to the fact initial messages need not be in direct successors of a node, computing initiating instances requires an iterative approach. First, for each BMsc the set of initiating instances and the set of idle instances is computed. Then, for each node n the set of initiating instances is extended: all initiating instances of direct successors of n are added if those instances are idle in n. This process is repeated until no more extensions can be done. Finally, for each branching node it is checked whether its successors fulfill the definition above.

Within SCStudio, the following approach is used for implementing the algorithm.

1. The given HMsc H is transformed to an equivalent BMsc graph G.
2. Using DFSRefNodeHMscTraverser the sets of initiating and idle instances are computed for each reference node in G. The result is stored in dynamic attributes of the respective node. Start node is considered an empty reference node.
3. DFSRefnodeHMscTraverser is used again and for each reference node (and the start node) n
   (a) The list of reference successors of n is computed by NodeFinder. That means reference nodes accessible along connection nodes are considered successors as well.
   (b) The set of initiating events of n is extended as described above using the information from the successors.
4. The previous step is repeated until no set can be further extended.
5. DFSRefNodeHMscTraverser is used one more time and for each node n it checks the condition of local choice:
   (a) NodeFinder finds reference successors of n.
   (b) If there are at least two successors, their sets of initiating instances are compared. If they are different or if there is more than one instance in the sets, the node is non-local.

(c) If n is non-local, *the node in the original HMsc* h from which n is created *is marked*.

6. *The original HMsc* H is traversed using DFSHMscTraverser and for each marked node m the path from the start node to m is stored to the list of results.

7. If the list of results is empty, H satisfies local choice. Otherwise, the list contains paths to all the non-local nodes in H.

Decidable Local Choice

In [5], another algorithm for *decidable local choice* is presented. The main idea is that some non-localities could be resolved without modifications to the sequence of messages: in some cases, it is possible to add information to an existing message before the non-local branching occurs. Using the information, instances may agree on the branch to use.

The algorithm is yet to be implemented in SCStudio; it has been decided to *postpone* the implementation because of these facts:

- Adding underhand information to existing messages is actually a workaround solution, which is rather to be avoided at an early design phase.
- The format of messages is often rigidly defined in advance (or even standardized) and no additional information can be added to it.
- The deciding event may be far from the choice it decides. This reduces the practical application of this method.
- The effort estimation for the implementation is high.

Despite this, implementing the algorithm could be beneficial in the future.

4.4 Realizability

In the current implementation, an HMsc is *realizable* iff it is universally bounded and local choice. In future, the checker is expected to be configurable. Each user shall be able to define which properties are realizable in the environment the Msc is to be implemented.

4.5 Name Checker

Name checker is a very simple checking algorithm for BMscs and HMscs. The checker consists of two independent parts. The first part checks if there are any duplicate names of instances in a BMsc (possibly referenced by an HMsc). As some checkers reference instances by their names, such Mscs may be verified incorrectly. In a BMsc, it is possible to avoid using names for referencing instances. In an HMsc, however, it is often necessary to match instances from different BMscs; matching by name is the only option here and no algorithm can generally work correctly if there are any duplicate instance names.

The other part *has not been released* in SCStudio. It checks whether sets of instances of BMscs referenced by an HMsc are *equal*. The property is violated whenever there are two BMsc having different instance names in an HMsc. It's been decided not to consider this setup a flaw. It is actually common in an HMsc. When an instance present in a BMsc referenced by an HMsc is missing in other BMscs from the HMsc, it is considered *idle* in such a BMsc.

Implementation

The implementation of the first part is straightforward as it is basically just the matter of finding duplicates is a list of strings or comparing such lists.

Chapter 5

Realization

An Msc gives a *global* view of a distributed communicating system. An actual implementation of such a system, however, encompasses a set of independent entities with an *individual behavior* of each of them. This chapter deals with producing the specification of the individual behavior from an Msc-specification (*realization*).

We use the DiVinE modelling language [1] as the target formalism for the realization. This allows for the *LTL model checking* of the realization and brings together two research branches at the Faculty of Informatics.

First, DiVinE and the modelling language is introduced and then the Msc-transformation algorithms are presented.

5.1 DiVinE Basics

DiVinE is an open source tool for platform-dependent LTL model checking and reachability analysis of discrete distributed systems [1]. It has been developed at the Faculty of Informatics.

First and foremost, the model needs to be specified in DVE modelling language and the property needs to be specified either as an LTL formula or as a Büchi automaton. In this thesis, we deal with transforming an MSC-specification of a distributed system to the DVE modelling language. Further usage of the model is not discussed.

DVE Modelling Language

The basic modelling unit in DVE is a system, which is composed of processes. Processes can go from one process state to another through transitions, which can be guarded by a condition – this condition, also called a "guard" determines whether the transition can be activated.

Transitions can be synchronised through channels. Only exactly two processes can be synchronised in a single step. Through the synchronisation

on a channel, a value can be optionally transmitted from one process to the other.

A system may be synchronous or asynchronous – in an asynchronous system, synchronization through channels is non-blocking, i.e. the sending process may continue immediately after putting a message to the channel.

The following example shows the syntax of contructs needed for the MSC-transformation.

First, *channels* are defined. In this case, a typed buffered channel is defined. It is capable of storing up to three integers. After the definition of channels, *processes* are defined. The definition begins with *local variables*. Then *states* of the process are declared. One of the states is marked as *initial*.

```
channel {int} to_b[3];
process a
{
  int i;
  state a_0, a_1;
  init a_0;
```

Next, transitions between states are defined. For each transition, additional actions or conditions may be specified. In this example, a puts the value 0 to the channel while executing the transition.

```
trans
  a_0 -> a_1 {sync to_b!{0}};
}
process b
{
int i;
state b_0, b_1, b_2, b_3;
init b_0;
trans
```

This transition may be executed only if there is a value in the channel. The value is then stored in the local variable i.

```
  b_0 -> b_1 {sync to_b?{i};},
```

The following transitions may be executed only if i equals 0 or 1, respectively.

```
  b_1 -> b_2 {guard i == 0;},
  b_1 -> b_3 {guard i == 1;};
}
system async;
```

By specifying the system to be asynchronous the definition of the model ends.

5.2 Transformation Algorithm

The main idea of the transformation of an MSC-specification to the DiVinE modelling language is straightforward. *Instances* in an Msc naturally correspond to *processes* in DiVinE. In the following, a transformation algorithm for *BMsc graphs* is presented. It can be used for an individual BMsc or a nonrecursive HMsc by constructing a BMsc graph equivalent to the BMsc or the HMsc.

5.2.1 Messages

For each process, one *channel* is defined[1]. The size of each channel is set to the total number of receive events of the process. A message sent to a process is realized as putting a value to its channel. The value consists of three parts in our realization:

- Identification of the HMsc node the message belongs to
- Identification of the sender
- Label of the message

The *sending process* simply puts these elements to the receiver's channel:

```
state_before_send -> state_after_send
 {sync receivers_channel!{hmsc_node,sender,message;}
```

Receiving a message is done in two steps. First, the data is copied to local variables of the receiving process and the process enters a special testing state. Then, a *guard* is used to check if the values of the local variables correspond to the expected message in the testing state.

```
state_before_send -> test_state
  {sync receivers_channel?{node,sender,message;},
test_state -> state_after_send
  {guard node==expected_node
    and sender==expected_sender
    and message==expected_message;}
```

5.2.2 Intra-BMsc Transitions

Let us first consider an individual *BMsc*. The BMsc is processed instance by instance. For each instance, states and transitions are generated as follows:

1. This allows detecting the race condition, for example.

- Whe the BMsc has no event, it is ignored as a whole and no states or transitions are created.
- There is an initial state representing the situation when no actions have been performed yet.
- For a sequence of send or receive events *outside a coregion*, states are logically placed between events and transitions between the states represent the events. Transitions are specified as shown above.
- For a *coregion*, there is a state for every possible combination of executed and not executed events in the coregion. From each such a state, only events allowed by the general ordering of the coregion may be executed. The execution continues after the coregion when all its events are executed. Transitions are again specified as shown above. The system may be non-deterministic in this case. When multiple *receive* events are allowed, there is only one common testing state and the message actually received is distinguished in it.
- When the instance is idle, only two states are generated – an initial and a final state.

5.2.3 Inter-BMsc Transitions

BMsc graphs are required to be *local-choice* because the algorithm from [7] is used. The transformation proceeds as follows:

- The BMsc graph is processed instance by instance. In each step, transitions throughout the whole BMsc graph for one instance are generated.
- Each process starts at an initial state corresponding to the start node of the BMsc graph.
- In a *non-branching node*, the execution simply continues in the succeeding node after the referenced BMsc is executed.
- In the case of a *branching node*, we make use of the fact the BMsc graph is *local-choice* and therefore there is only one process initiating the communication in all branches.

  The initiating process is allowed to enter any of the branches after executing the referenced BMsc. Other processes enter a special *awaiting state*. When they receive a message in the awaiting state, the information about the new node from the message is used to enter the

correct node chosen by the initiating process. Therefore, transitions with appropriate guards from the awaiting state to the state *after a minimal receive event* in all branches are added.

- BMscs with no events are considered condition nodes and no states or transitions are generated for them.

5.2.4 Coregion Algorithm

While the actual implementation of the majority of the points is relatively straightforward, the implementation of the transition-generation in a coregion is a bit more complicated and thus the algorithm is outlined below.

- Number the events of the coregion arbitrarily from 1 to n. In the following part, events are referenced using their numbers.
- Let S be a string "$0 \cdots 0$"of the length n. Such a string represents a state in the coregion. When S contains 0 at the position i, event i has not been yet executed in state i. When S contains 1 at that position, i has been already executed.
- Let Q be a queue of strings. Initially, put S to Q.
- While Q is not empty:
  - Remove the front of Q and store it in S.
  - For each not executed event i in S such that all predecessors of i (with respect to the general ordering of the coregion) are executed in S:
    - Set $S' = S$ and set i to "1" in S'.
    - If S' is not in Q, add S' to Q.
    - If i is a send event, add a transition from S to S' together with the corresponding send action.
    - If i is the first processed receive event in S, create a new testing state T and transitions from S to T and T to S' together with the corresponding receive and guard actions.
    - If i is another receive event, use the previously created T and add only a transition from T to S' with the corresponding guard.

Chapter 6

Conclusion

Before the work on this thesis started, SCStudio had actually been a prototype with a limited set of functions. As some parts of the code had been written at an early phase of SCStudio development when no advanced features had been available, the code had been long and ilegible. Verification algorithms had been capable of showing only a single counterexample even if there had been multiple flaws in an Msc.

This thesis has improved SCStudio both from the user perspective and the programming perspective. Namely, the existing verification algorithms have been reviewed and also rewritten in some cases. Errors in the implementation as well as the definition of some of them have been discovered and corrected. All the algorithms are newly capable of finding multiple counterexamples. This is especially valuable for the race checker; the original footprint algorithm has been improved and also the proof of its correctness has been updated to reflect the multiple-result capability.

Five new verification algorithms have been implemented. Some of them check simple syntactic properties of Mscs while others are capable of detecting interesting semantic issues. We would like to point out the local choice checker because its implementation is a continuation of a previous theoretical work written at the Faculty of Informatics.

In addition, an algorithm for exporting Mscs to the DiVinE format has been implemented. This opens a whole new world of LTL model-checking for MSC-specifications and gives a fundamentally different view of Msc-specifications. Last but not least, this functionality joins different research branches at the Faculty of Informatics in a way.

Finally, several functions simplifying further programming in SCStudio have been implemented. Those functions may be reused to speed up further development of new features in SCStudio.

Generally, SCStudio has been significantly set on and it has become a mature tool. We hope this thesis will serve as a documentation to the existing functionality as well as the basis for further development.

Bibliography

[1] DiVinE Tool. `http://divine.fi.muni.cz/`.

[2] MS Visio. `http://office.microsoft.com/en-us/visio/`.

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

[4] J. Babica. Message Sequence Chart Properties and Checking Algorithms. Master's thesis, Faculty of Informatics, Masaryk University, 2009.

[5] M. Chmelík. Deciding Non-local Choice in High-level Message Sequence Charts. Bachelor's Thesis, Faculty of Informatics, Masaryk University, 2009.

[6] J. Fousek, V. Řehák, P. Slovák, and J. Strejček. Decidable race condition in high-level message sequence charts, 2008. Unpublished preliminary version of [14].

[7] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-State High-Level MSCs: Model-Checking and Realizability. *Journal of Computer and System Sciences*, 72(4):617–647, 2006.

[8] ITU Telecommunication Standardization Sector Study group 17. ITU recommandation Z.100, Specification and Description Language (SDL), 2002.

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

[10] M. Lohrey and A. Muscholl. Bounded MSC Communication. volume LNCS 2303. Springer, 2002.

[11] R. Morin. On Regular Message Sequence Chart Languages and Relationships to Mazurkiewicz Trace Theory. *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, pages 332–346, 2001.

[12] A. Muscholl and D. Peled. Deciding properties of message sequence charts. In Stefan Leue and Tarja Johanna Systä, editors, *Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, pages 43–65. Springer Berlin / Heidelberg, 2005.

[13] Z. Pekarčíková. Computer Aided Layout of Message Sequence Charts. Bachelor's Thesis, Faculty of Informatics, Masaryk University, 2011.

[14] V. Řehák, P. Slovák, J. Strejček, and L. Hélouët. Decidable Race Condition and Open Coregions in HMSC. *Electronic Communications of the EASST*, 29:80–91, 2010. Postconference proceedings of the 9th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010).

[15] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[16] R. E. Tarjan. Enumeration of the elementary circuits of a directed graph. Technical report, Ithaca, NY, USA, 1972.

Appendix A

Updated Proof for the Race Checker

This chapter contains definitions and lemmata relevant for the proof of correctness of the footprint algorithm as presented in [6]. For each section we discuss differences arising from the fact the algorithm does not halt after finding the first occurence of race condition. Please refer to Section 3.5 for information about the modification. The following text is not my authorial work. It is taken from [6], including numbering of sections.

Definition 1. *An* MSC *(with coregions and connections) is defined as a tuple* $(E, <, \mathcal{P}, P, \mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{G})$ *where:*

- E *is a finite set of* events *partitioned as* $E = \mathcal{S} \cup \mathcal{R}$*, where* $\mathcal{S}$ *and* $\mathcal{R}$ *are disjoint sets of* send *and* receive events*, respectively.*
- $<$ *is a strict partial order on* E *called* visual order *and specified in Definition 3.1.1.*
- $\mathcal{P}$ *is a finite set of* processes*.*
- $P : E \to \mathcal{P}$ *is a mapping that associates each event with a process.*
- $\mathcal{M} \subseteq (\mathcal{S} \times \mathcal{R})$ *is a bijective mapping, relating every send with a unique receive. For any* $(e, f) \in \mathcal{M}$ *we also write* $\mathcal{M}(e) = f$*. We assume that* $P(e) \neq P(\mathcal{M}(e))$*, i.e. a process cannot send a message to itself.*
- $\mathcal{C}$ *is a set of pairwise disjoint* coregions *where a coregion, say* $C \in \mathcal{C}$*, is defined as a subset of events on some process, i.e.* $C \subseteq P^{-1}(p)$ *where* $p \in \mathcal{P}$*.*
- $\mathcal{G} \subseteq \bigcup_{C \in \mathcal{C}} C \times C$ *is a union of (strict) partial orders on events within coregions. These partial orders are called* connections*.*

Definition 7. *An* MSC-graph *is a tuple* $(S, \to, s_0, s_f, L, \mathcal{L})$ *where*

- S *is a finite set of states,*
- $\to \subseteq S \times S$ *is a transition relation,*

- $s_0 \in S$ *is a distinguished* initial state,
- $s_f \in S$ *is a distinguished* final state,
- $\mathcal{L}$ *is a finite set of MSCs over a common set of processes, and*
- $L(s) : S \to \mathcal{L}$ *is a mapping assigning to each state an MSC.*

A sequence of states $\sigma = s_1 s_2 \cdots s_k$ *is a* path*, if* $(s_i, s_{i+1}) \in \to$ *for every* $1 \leq i < k$*. A path is a* run *if* $s_1 = s_0$ *and* $s_k = s_f$*.*

Definition 8. *Given two MSCs* $M_1 = (E_1, <_1, \mathcal{P}, P_1, \mathcal{S}_1, \mathcal{R}_1, \mathcal{M}_1, \mathcal{C}_1, \mathcal{G}_1)$ *and* $M_2 = (E_2, <_2, \mathcal{P}, P_2, \mathcal{S}_2, \mathcal{R}_2, \mathcal{M}_2, \mathcal{C}_2, \mathcal{G}_2)$ *over a common process set* $\mathcal{P}$ *and disjoint sets of events* $E_1 \cap E_2 = \emptyset$ *(we can always rename events so that the sets become disjoint), let the concatenation of* M_1 *and* M_2 *be the MSC* $M_1 \cdot M_2 = (E_1 \cup E_2, <, \mathcal{P}, P_1 \cup P_2, \mathcal{S}_1 \cup \mathcal{S}_2, \mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{M}_1 \cup \mathcal{M}_2, \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{G}_1 \cup \mathcal{G}_2)$ *where* $<$ *is the transitive closure of* $<_1 \cup <_2 \cup \bigcup_{p \in \mathcal{P}} (P_1^{-1}(p) \times P_2^{-1}(p))$*.*

Definition 10. *Let* $M = (E, <, \mathcal{P}, P, \mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{G})$ *be an MSC. First, we define an auxiliary function* $label : E \to \{p!q, p?q \mid p, q \in \mathcal{P}\}$ *such that*

$$label(e) = \begin{cases} p!q & \text{– if } e \in \mathcal{S},\ p = P(e), \text{ and } q = P(\mathcal{M}(e)) \\ p?q & \text{– if } e \in \mathcal{R},\ p = P(e), \text{ and } q = P(\mathcal{M}^{-1}(e)) \end{cases}$$

A linearization *of the MSC* M *according to a relation* $\sqsubset \in \{<, \ll\}$ *is a word* $label(e_1) label(e_2) \cdots label(e_n)$ *such that* $E = \{e_1, e_2, \cdots, e_n\}$ *and* $e_i \sqsubset e_j$ *implies* $i < j$*. Moreover, we define* $Lin_{\sqsubset}(M)$ *to be the set of all linearizations of* M *according to* $\sqsubset$*.*

Definition 14. *Let* $M = (E, <, \mathcal{P}, P, \mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{G})$ *be an MSC. We set:*

$$\begin{aligned} MaxP(M) &= \{e \in E \mid \forall e' \in E\ .\ P(e') = P(e) \implies e \not\ll e'\} \\ MinP(M) &= \{e \in E \mid \forall e' \in E\ .\ P(e') = P(e) \implies e' \not\ll e\} \end{aligned}$$

This definition remains valid if M is not race-free. Notice there may be events that are minimal/maximal with respect to $\ll$ but not minimal/maximal with respect to $<$. This actually allows finding more occurences between BMscs if some of the BMscs contains border events that are in race with other events of the BMsc.

Lemma 15. *Let* M_1*,* M_2 *be two race-free MSCs. The concatenation* $M_1 \cdot M_2$ *contains a race if and only if* M_1 *and* M_2 *are race-free and there are two events* $e \in MaxP(M_1)$ *and* $f \in MinP(M_2)$ *such that* $P(e) = P(f)$ *and* $e \not\ll f$*.*

The lemma is *not valid* if M_1 or M_2 is not race-free. The "only if" part does not hold any more. The "if" part follows directly from the definition of a race and remains valid. The purpose of the lemma is to show that only

"border" events of two concatenated MSCs need to be checked in order to identify a race condition. Because the new algorithm searches for race condition in individual BMscs prior to checking *border* events between BMscs, we can reformulate the lemma in the following way:

Lemma 15a. *Let M_1, M_2 be two MSCs. The concatenation $M_1 \cdot M_2$ contains a race if and only if there are two events $e \in MaxP(M_1)$ and $f \in MinP(M_2)$ such that $P(e) = P(f)$ and $e \not\ll f$ or M_1 contains a race or M_2 contains a race.*

For race-free MSCs, the original proof in [6] is applicable. When either of the MSCs contains a race, the concatenation obviously contains a race as well.

Lemma 16. *Let $M_1 \cdot M_2$ be a concatenation of two race-free MSCs M_1 and M_2, and $e \in MaxP(M_1)$, $f \in MinP(M_2)$ be two events such that $P(e) = P(f)$. Then $e \not\ll f$ if and only if $f \in \mathcal{R}$, $label(e) \neq label(f)$, and*

$$\{P(e') \mid e' \in MaxP(M_1) \wedge e \ll e'\} \cap \{P(f') \mid f' \in MinP(M_2) \wedge f' \ll f\} = \emptyset.$$

Proof. Let M_1, M_2 be two race-free MSCs, $e \in MaxP(M_1)$, and $f \in MinP(M_2)$ such that $P(e) = P(f)$. We prove that the equivalence holds in all situations. The situations are divided into several cases:

1 Let f be a sent event, i.e. $f \in \mathcal{S}$. In this setting, the statement says that $e \not\ll f$ iff $FALSE$, i.e. $e \ll f$.

From the definition of concatenation of MSCs, $P(e) = P(f)$ implies that $e < f$. Then the definition of $\ll$ and $f \in \mathcal{S}$ conclude that $e \ll f$.

2 Let f be a receive event, i.e. $f \in \mathcal{R}$. We divide this case into the following two subcases (according to the value of $label(e)$ and $label(f)$).

2.1 Let $label(e) = label(f)$. In this setting, the statement says that $e \not\ll f$ iff $FALSE$, i.e. $e \ll f$.

If $label(e) = label(f)$, then e is also a receive event, i.e. $e \in \mathcal{R}$, and there are two events e' and f' such that $(e', e) \in \mathcal{M}_1$, $(f', f) \in \mathcal{M}_2$, and $P(e') = P(f')$. From the definition of concatenation of MSCs, $P(e') = P(f')$ implies that $e' < f'$. Due to the fact that $f' \in \mathcal{S}$, it holds that $e' \ll f'$. Finally, from the fifo condition, it follows that $e \ll f$.

2.2 Let $label(e) \neq label(f)$. We divide this case into the following two subcases depending on whether the event e is a send or a receive event.

2.2.1 Let e be a receive event, i.e. $e \in \mathcal{R}$. Let us note that $\{P(e') \mid e' \in MaxP(M) \wedge e \ll e'\} = \emptyset$ if $e \in \mathcal{R} \cap MaxP(M)$. Therefore, in this setting, the statement says that $e \not\ll f$ iff $TRUE$.
An quick inspection of the definition of $\ll$ shows that $e \ll f$ could hold only due to the transitivity of $\ll$. As there are no actions greater (w.r.t. $\ll$) than e in M_1 and f is minimal on the process $P(e)$, there is no event g such that $e \ll g \ll f$. Therefore, $e \not\ll f$ holds.

2.2.2 Let e be a send event, i.e. $e \in \mathcal{S}$. We show that $e \ll f$ if and only if there are two events $e' \in MaxP(M_1)$ and $f' \in MinP(M_2)$ such that $e \ll e'$, $f' \ll f$, and $P(e') = P(f')(\neq P(e) = P(f))$. We discuss the following three subcases. The first two prove the implication "$\Leftarrow$" and the last proves the implication "$\Rightarrow$".

2.2.2.1 Assume there are the events e' and f', and $f' \in \mathcal{S}$. From the definition of concatenation it follows that $e' < f'$. The definition of the causal ordering implies $e' \ll f'$, and so, due to transitivity of $\ll$, the relation $e \ll f$ holds.

2.2.2.2 Assume there are the events e' and f', and $f' \in \mathcal{R}$. Due to $P(f') \neq P(f)$ and $f' \ll f$, there is a send event f'' on the same process as the event f' such that $f' < f'' \ll f$. From the definition of concatenation, it follows that $e' < f'' \ll f$ and the definition of the causal ordering implies $e' \ll f''$. Therefore, $e \ll f$.
Note that there is a race between events e' and f'. This race would be found by checking the events e' and f', not now.

2.2.2.3 Finally, we assume that there are no such events e' and f'. As M_1 and M_2 are race-free, there is no event g such that $e < g < f$, much less $e \ll g \ll f$. Quick inspection of the definition of $\ll$ shows that $e \not\ll f$.

□

Notice the fact M_1 and M_2 are race-free is used only in part 2.2.2.3. We present an alternative proof of the part:
2.2.2.3 Let us assume $e \ll f$. Because e is a send event and f is a receive event and both the events are on the same instance, $e \ll f$ must hold due to transitivity of $\ll$. Therefore there are some events $e_1, \ldots, e_n$ from M_1 and $f_1, \ldots, f_m$ from M_2 such that $e \ll' e_2 \ll' \cdots \ll' e_n \ll' f_1 \ll' \cdots \ll' f_m \ll' f$ where $\ll'$ is $\ll$ without the application of the transitive closure.

Since $e_n \ll' f_1$ and there may be no messages sent across BMscs, we get e_n and f_1 are on the same instance by the definition of $\ll$ (see Definition 3.5.1).

e_n, however, need not be maximal on the instance. Then there is some maximal e'_n such that $e_n \ll e'_n$. Analogously, for f_1 we can obtain f'_1 that is minimal on the instance.

Finally, let $e' = e'_n$ and $f' = f'_1$. Then $e' \in MaxP(M_1)$ and $f' \in MinP(M_2)$ such that $e \ll e'$, $f' \ll f$, and $P(e') = P(f') (\neq P(e) = P(f))$. □

Definition 17. *Let $M = (E, <, \mathcal{P}, P, \mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{G})$ be an MSC. A pair (l, F) of a label and a set of processes is called a* footprint *of M if there is an event e in $MaxP(M)$ such that $l = label(e)$ and $F = \{P(e') \mid e' \in MaxP(M) \wedge e \ll e'\}$.*

Moreover, we define an auxiliary function $active$ from labels to processes such that $active(p!q) = p$ and $active(p?q) = p$. Furthermore, for an MSC M, we set $active(M)$ to be the set of all active processes of M, i.e. $active(M) = \{P(e) \mid e \in E\}$.

This definition is valid for non-race-free MSCs as well.

Lema 18. *Let M_1 and M_2 be two race-free MSCs. The concatenation $M_1 \cdot M_2$ contains a race if and only if there is a receive event $f \in MinP(M_2)$ and a footprint (l, F) of M_1 such that $active(l) = P(f)$, $l \neq label(f)$, and $F \cap \{P(f') \mid f' \in MinP(M_2) \wedge f' \ll f\} = \emptyset$.*

Due to the change in Lemma 15, this lemma must be reformulated in the following way:

Lemma 18a. *Let M_1 and M_2 be two MSCs. The concatenation $M_1 \cdot M_2$ contains a race if and only if M_1 and M_2 are race-free and there is a receive event $f \in MinP(M_2)$ and a footprint (l, F) of M_1 such that $active(l) = P(f)$, $l \neq label(f)$, and $F \cap \{P(f') \mid f' \in MinP(M_2) \wedge f' \ll f\} = \emptyset$ or M_1 contains a race or M_2 contains a race.*

Lemma 19. *Let $\mathcal{FP}_1$ be the set of all footprints of a race-free MSC M_1 and $\mathcal{FP}_2$ be the set of all footprints of a race-free MSC $M_2 = (E, <, \mathcal{P}, P, \mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{G})$ such that $M_1 \cdot M_2$ is also race-free. Then the set of all footprints of the concatenation $M_1 \cdot M_2$ is equal to*

$$\mathcal{FP}_2 \cup \{(l, F \cup F') \mid \begin{array}{l} (l, F) \in \mathcal{FP}_1 \wedge \\ F' = \{P(e) \mid e, f \in E \wedge P(f) \in F \wedge f \ll e\} \wedge \\ active(l) \notin active(M_2)\ \}. \end{array}$$

Since the lemma requires M_1, M_2 and $M_1 \cdot M_2$ to be race free, a reformulation is needed. When we allow some of the MSCs to contain a race, we cannot guarantee the new set of footprints to contain all the footprints of the concatenation. This is, however, not a problem because we do not require

the algorithm to find all occurences of the race condition. The reformulated lemma guarantees that the computed set contains only valid footprints of the concatenation:

Lemma 19a. *Let $\mathcal{FP}_1$ be the set of footprints of an MSC M_1 and $\mathcal{FP}_2$ be the set of footprints of an MSC $M_2 = (E, <, \mathcal{P}, P, \mathcal{S}, \mathcal{R}, \mathcal{M}, \mathcal{C}, \mathcal{G})$. Then a* subset *of the set of all footprints of the concatenation $M_1 \cdot M_2$ is equal to*

$$\mathcal{FP}_2 \cup \{(l, F \cup F') \mid \begin{array}{l} (l, F) \in \mathcal{FP}_1 \wedge \\ F' = \{P(e) \mid e, f \in E \wedge P(f) \in F \wedge f \ll e\} \wedge \\ active(l) \notin active(M_2)\ \}. \end{array}$$

Now, we are ready to present the updated algorithm for finding trace race condition in a BMsc-graph:

Algorithm 1 – Trace Race in MSC-graph

Input: An MSC-graph $G = (S, \rightarrow, s_0, s_f, L, \mathcal{L})$
Output: a set of occurences of the race condition if the MSC-graph contains a trace race, an empty set otherwise

for every M of $L(S)$ **do**

if M contains a race **then** add M with the events in race marked and a path to M to the set of results

$TODO := \{(s_0, (l, F)) \mid (l, F)$ is a footprint of $L(s_0)\}$

while $TODO \neq \emptyset$ **do**

take $(s, (l, F))$ from $TODO$

add $(s, (l, F))$ to $DONE$

\# *race checking*

for every s' such that $(s, s') \in \rightarrow$ **do**

for every $f \in \mathcal{R} \cap MinP(L(s'))$ **do**

if $active(l) = P(f)$ and $l \neq label(f)$
and $F \cap \{P(f') \mid f' \in MinP(L(s')) \wedge f' \ll f\} = \emptyset$ **then**
add a path to s and s' to the set of results and mark l and f

\# *new footprints computation*

$\mathcal{FP}' :=$ footprints of $L(s')$

if $active(l) \notin active(L(s'))$ **then**

add $(l, F \cup \{P(e) \mid e, f \in E \wedge P(f) \in F \wedge f \ll e\})$ to $\mathcal{FP}'$

for every (l', F') of $\mathcal{FP}'$ **do**

if $(s', (l', F')) \notin TODO \cup DONE$ **then**

add $(s', (l', F'))$ to $TODO$

return the set of occurences of the race condition

The new algorithm is correct; the same argument as in the original work may be used: As each state s can be associated with only a finite number of possible footprints, the Algorithm 1 terminates.

Lemma 18 guarantees the correctness of the race checking part and Lemma 19 guarantees the correctness of the new footprint computation part.

Appendix B

Content of the CD

thesis.pdf	the pdf version of this document
thesis.zip	the archive containing the LaTeX source of this document
scstudio.exe	installer for the latest version of SCStudio
scstudio.zip	source code of the latest version of SCStudio

Description of the content of the release can be found in the README file.