MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Import of MSC Diagrams from the ITU-T Z.120 Text Representation

BACHELOR'S THESIS

Matúš Madzin

Brno, 2009

Declaration

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

Advisor: Ing. Petr Gotthard

Abstract

The result of this thesis is a parser which has been developed as part of the SCStudio application. The parser is used to import message sequence chart from a textual file according to the ITU-T standard. Introductions to the message sequence chart and the SCStudio explain why an application such as the SCStudio is important and why message sequence chart helps to save resources and time of a development. This paper describes a design process and actual behaviour of the parser.

Keywords

Message Sequence Chart, MSC, ANother Tool for Language Recognition, ANTLR, basic Message Sequence Chart, bMSC, High-level Message Sequence Chart, HMSC, Sequence Chart Studio, SCStudio, ANTLR, parser, ITU-T, Z.120

Acknowledgement

I would like to thank my supervisor Petr Gotthard for valuable advice, comments and support during my whole work on this thesis. I would like to express my deep gratitude to his interest in the whole SCStudio project and discussions.

I would also like to thank Vojtěch Řehák for the opportunity to joint the SCStudio project and I really regard co-operation with him in last years.

Contents

Chapter 1

Introduction

This thesis was created as a part of The Sequence Chart Studio (SCStudio) which is a user-friendly drawing application and verification tool for Message Sequence Chart (MSC) and UML Sequence Diagrams. The main goal of this work is to import MSC diagrams from the ITU-T Z.120 textual representation to the SCStudio. This feature has two main advantages for SCStudio because it increases interoperability with similar tools (the SCStudio will be able to work with MSC which has been created in other drawing editor) and it creates environment for developers to check new implementations by automatic tests.

ITU-T Z.120 defines an MSC in textual or graphical representation. In this paper, there is discussed only textual form which can be described by formal grammar. Unfortunately, the ITU-T Z.120 standard allows ambiguous behaviour in special situations, so it was necessary to define and rewrite some parts of the grammar.

The parser is mainly written in C++ code and in the paper. In the paper, there are some examples of code. So, the basic knowledge of C++ language is required. The grammar for the parser generator is written in its syntax and it is discussed in part about ANother Tool for Language Recognition (section 4.2). So, there are no requirements about it.

This thesis is divided into four main chapters, not including introduction and conclusion. The chapter called *Message Sequence Chart* provides a brief introduction into the MSC and its main characteristics. There are described what the MSC is, where is used and what differences between basic MSC and high-level MSC are.

In the next chapter, there is introduction of SCStudio application (what the SCStudio enables users to do, which verification tools it supports). In the end of chapter, there is small description of how the SCStudio stores MSC.

The following chapter introduces requirements of the parser and tools which were used during development. Especially, there are introductions of ANother Tool for Language Recognition (ANTLR) and grammar syntax which is used. Then, compatibility with the other applications which can draw MSC diagrams is mentioned.

Finally, in the chapter titled *Parser Design* the general ideas of the parser are explained. There are all parts of the parser described (syntax of the grammar, functionality of the auxiliary structure which is called Context structure, etc.). There is also an example of an input file and the parser's source code. After the grammar is discussed, all steps the parser makes are explained. In the end of the chapter, error reporting is discussed (which kinds of errors can occur in a textual file and what the parser does when it recognize some).

Chapter 2

Message Sequence Chart

Message Sequence Chart (MSC) is a language to describe the communication between a number of independent components. MSC diagram is also known as message flow diagram, time sequence diagram or object interaction diagram and it is used in a number of software engineering notation frameworks such as ITU-T formal description techniques and UML. MSC is an overview specification of the communication behaviour for real time systems, in particular telecommunication switching systems. The main characteristics are the following:

- MSC describes the order of messages and allows restrictions on transmitted data values and on the timing of events.
- MSC allows description of complete or incomplete specifications. It is useful in the first attempts to design behaviour of components.
- MSC is a formal language and has formal expressive power as well as intuitive appearance.
- MSC is widely applicable formalism which can be used in different ways.
- MSC supports structured design. Basic Message sequence Charts is used to visual representation of communication (scenario) and can be combined to more hierarchical form (Hierarchical Message Sequence Charts (HMSC) also known as a High-level Message Sequence Charts).

For more information, see [11].

It is useful to have mechanisms for error detection at the beginning of a development. Software designers use MSC for modeling of communications between components in system. Especially in early analysis because this is the way to represent a simple overview of communication and there are algorithms over MSC to detect errors. It can help to find hidden behaviour during the execution. Many kinds of errors are known. One of them is race condition.

The race condition exists when two messages appear in one (visual) order, but they can be shown in the opposite order during the execution. It is difficult and important to find them because these conflicts can be crucial in the system behaviour and can resolve problems with incorrect or incomplete assumptions about chains of dependencies in the design. See Figure 2.1 and Figure 2.2.

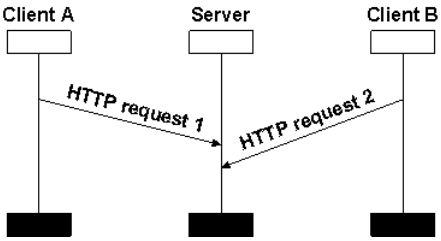

Figure 2.1: Design

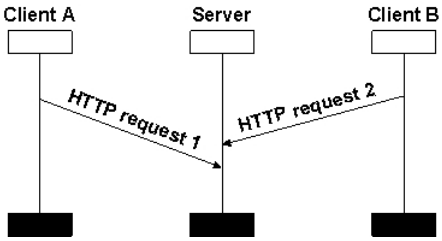

Figure 2.2: Possible execution

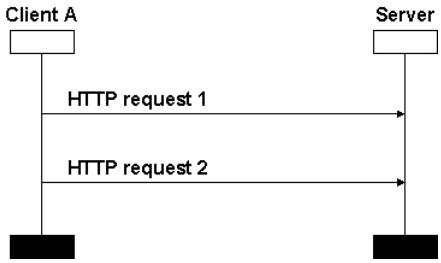

Figure 2.3: FIFO example

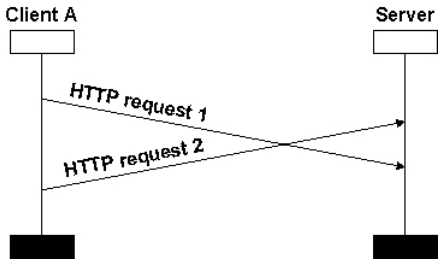

Figure 2.4: Non-FIFO example

Figure 2.1 describes communication which was designed. The message *HTTP request 1* is received first and *HTTP request 2* second but *Client A* and *Client B* are not synchronized. So, a situation can occur where the message *HTTP request 2* is received first. It shows that the messages can be received in opposite ordering as they were designed. This situation is described in Figure 2.2.

The next mistake of message ordering breaks FIFO (First In First Out) ordering. This mistake can be occurred when one component sends two messages to another component. See Figure 2.3 and Figure 2.4.

Figure 2.3 describes communication according to the FIFO ordering. It means that the messages are received in the order as they were sent. The scenario in Figure 2.4 is inconsistent with the FIFO ordering.

2.1 Basic Message Sequence Chart

A basic Message Sequence Chart (bMSC) is a diagram which describes behaviour between instances (components). One Message Sequence Chart describes a partial behaviour of a system. The diagram allows to draw message ordering through communication. It means there can be specified the area where and which messages are ordered or not. One of the important features is expression of time constraints.

There are a few ways to set the time order of events. In the relative timing case, timing is given with respect to a previous event. In the absolute timing, the absolute time is assigned. Then there is time interval which describes time dependencies between pairs of events or events and regions.

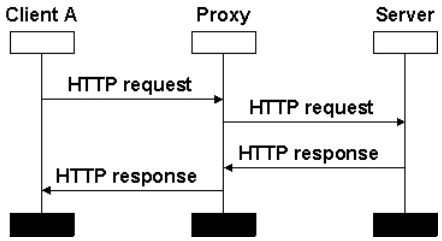

Figure 2.5: A basic MSC example

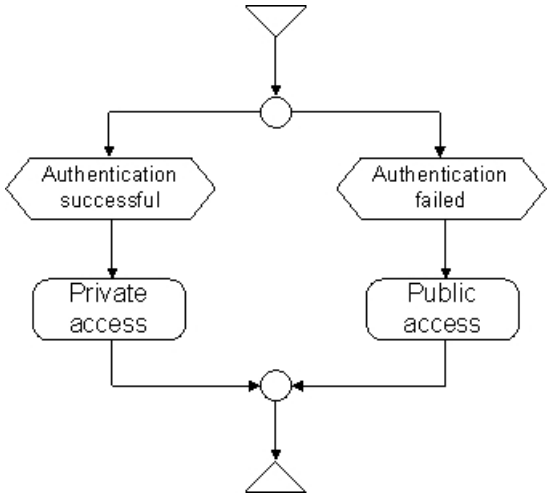

Figure 2.6: A high-level MSC example

A simple example of bMSC is in Figure 2.5. This MSC describes three instances (*Client A*, *Proxy*, *Server*) and communication over the HTTP protocol. *Client A* sends message (*HTTP request*) to the *Proxy*. The *Proxy* resends the message to the *Server*. When the request has been processed, the *Proxy* receives a message (*HTTP response*) and resends the message to the *Client A*.

2.2 High-level Message Sequence Chart

A High-level Message Sequence Chart (HMSC) is a graph that represents hierarchical structure of scenarios. One HMSC can have many levels and each of them can contains nodes which can have reference to another HMSC or bMSC (scenario). Nesting of MSCs is allowed in constricted way. MSC can be nested in finite form (recursive nesting is forbidden). It is the way to combine several MSCs together. Every HMSC contains one start node (initial node) and one end node (final node). A simple example of HMSC is in Figure 2.6.

The HMSC describes several possibilities of communication between user and web page. A execution starts in the start node (triangle on the top). The following node is the connection node which connects the start node with two successors. On each path, after connection node, there is a condition node which defines when the path is used. In case, the user was not authenticated, he has public access to the page (the right path). In case, the authentication was successful, the user gets private access (the left path). Then there is the connection node to connect the end node and its predecessors.

Chapter 3

SCStudio

The Sequence Chart Studio (SCStudio) is a user-friendly drawing and verification tool for MSC and UML Sequence Diagrams. The application was created as a joint project of ANF DATA spol. s r.o. and Masaryk University (the research center Institute for Theoretical Computer Science, Faculty of Informatics). The software is freely available under LGPL (GNU Lesser General Public License) via SourceForge [4].

The SCStudio consists of four main parts (drawing editor, data structure, verification tools, export and import). The SCStudio offers an open interface for additional modules.

One of the SCStudio implementations is module for Microsoft Visio. It is currently the only drawing editor of the SCStudio but there are some ideas to make a SCStudio drawing editor for other platforms. The SCStudio enables users of Microsoft Visio [12] to:

- Draw bMSC: instances, messages, coregions, general ordering.
- Draw HMSC: connections, references, conditions.
- Export drawing to ITU-T Z120 compliant textual format.
- Perform graphical syntax verification.
- Execute verification algorithms: detect deadlock/livelock, detect cycles and race conditions.
- Develop proprietary stencils and verification/export modules.

The next part of the SCStudio the data structure. This structure allows to store bMSC and HMSC. Since the structure is so large topic, an introduction of it is in the next section 3.1. The structure was created by Jindřich Babica (SCStudio developer) and for more information have a look at [8].

The following part is a set of verification tools. There are several verification algorithms which can be run over the message sequence chart (deadlock, livelock, cycles, race condition, FIFO ordering, ...).

Deadlock: During the execution, running comes into a situation when it can not reach any other node from the current node (HMSC).

Livelock: During the execution, running comes into a situation when it can not reach the end node from the current node (HMSC).

Cycle: It is a situation when message ordering creates a cycle among components (bMSC).

Race condition: It is a situation when two messages appear in one (visual) order, can be exchanged in the opposite order during the execution (bMSC).

FIFO ordering: Messages have to be ordered in the same communication channel, that is a set of messages with the same label between two components (bMSC).

Finally, there is a part about export and import. SCStudio supports the ITU-T Z120 standard. Export creates a textual file according to the standard and the parser which is described in this paper is responsible for importing an MSC to the SCStudio. The parser compatibility is discussed in section 4.3.

3.1 Representation of MSC in the SCStudio

The SCStudio uses its own structure to represent a bMSC or an HMSC. This structure was created to store all elements of the ITU-T standard. The next paragraph is a quick tour for imagination. For more information about structure, you can see documentation written by Jindřich Babica [8] or the actual version of the structure in the SourceForge [3].

An MSC is represented in the SCStudio as a hierarchy of classes. The structure has to be able to represent both kinds of MSCs. Each of them contains its own objects. That is the reason why there is the abstract class *Msc*. bMSC is represented as *BMsc* class and HMSC as *HMsc* class. Both are inherited from the *Msc* class.

Figure 3.1 describes classes which are necessary to store a bMSC. In the following description, the main characteristics of classes and they meaning in the hierarchy are explained.

BMsc contains a set of instances. An instance contains a pointer to the first *Event Area*. Every instance is separated into areas. Type of event area (*Strict Order Area* or *Coregion Area*) schedule relations between messages. *Event Area* also contains a set of messages and each message is described by two events (sender, receiver). There are two types of messages (complete, incomplete). Complete message has both events defined. Incomplete message has only one event defined and the second is filled by the type of incomplete message (Lost, Found). Every message is identified by label. It is important for communication channel identification. Every event remembers instance and area where it is. The message ordering is ensured because areas on instance are ordered and events in areas are ordered. *BMsc* itself remembers a set of instances.

Figure 3.2 describes class hierarchy of an HMSC. There are classes which are necessary to store an HMSC. The following paragraph describes the main characteristics of classes and their meaning in the class hierarchy.

An HMSC remembers the start node and the set of all nodes. *Start Node* contains a set of its successors (*HMsc Node*). *HMsc Node* can be represented by *Connection Node*, *Reference Node* or *Condition Node* and each of them remembers a set of its successors and predecessors. *Reference Node* references to either *BMsc* or *Hmsc*. *Condition Node* allows condition on path in

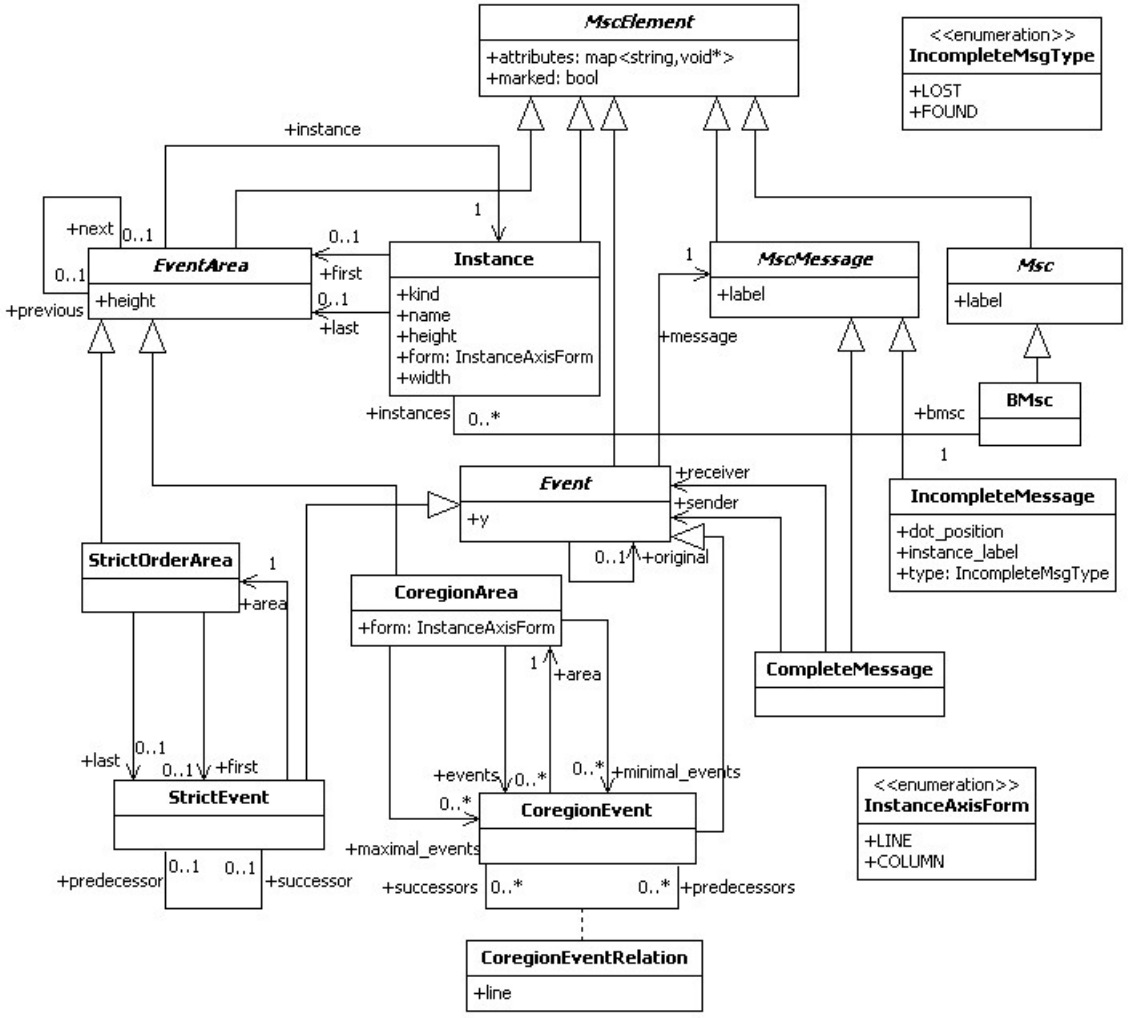

Figure 3.1: *BMsc* class diagram [8]

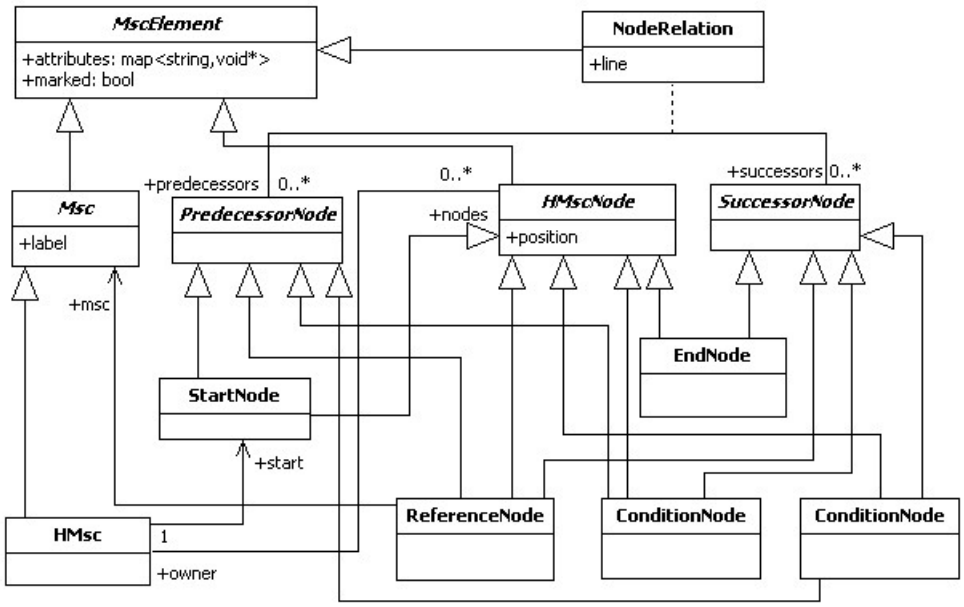

Figure 3.2: *HMsc* class diagram [8]

the graph, this condition is performed during execution and the running decides whether the condition is executed or not. *Connection Node* is used for multiple connections between the predecessor of a node and its successors.

Chapter 4

Function Description

The parser is an important part of the SCStudio because it allows to use this application without any drawing editor. It is one way to create the SCStudio platform independent. As was said, the SCStudio allows exporting an MSC structure to a textual file. So, the main requirement of the parser is to create the same MSC structure from the textual file which was created by the SCStudio.

The parser is generated by ANTLR. ANTLR was selected because it is a powerful tool. The main advantages of it are:

Platform independent: It is important for the parser because the SCStudio was designed as an application for different kinds of operating systems.

Simple grammar: ANTLR syntax allows to create a grammar in human readable form. It is also important because the grammar of the MSC language is quite wide and it is also necessary to add actions.

LL(*) parsing strategy: It allows the grammar in more natural forms, so it makes the building process easier because ambiguous statements can be designed. It means that the grammar can have more statements which look similar and for recognition the parser is able to have a look at the next possible statements and then decides whether the input is acceptable or not.

4.1 Standard ITU-T Z.120 grammar

The standard was published in April 2004 by the International Telecommunication Union. This standard describes the main characteristics of MSC and declares symbols in representation form. There exist meta-languages for a graphical grammar and a textual grammar. The parser works only with the textual form of the grammar. For more information about graphical grammar see [11].

The textual grammar is in the Backus-Naur Form (BNF) and a non-terminal symbol is enclosed in angle brackets. Terminal symbols are not enclosed. For example:

```
<instance parameter decl>::=
                inst <instance parm decl list> <end>
```

In this case the *inst* is a terminal symbol and the others are non-terminal. This expression describes that non-terminal symbol on the left-hand side consists terminal symbol *inst*, non-terminal <*instance parm decl list*> and non-terminal <*end*>.

There is the next example for introducing new symbols.

```
<message sequence chart> :: =
                [<virtuality>] msc <msc head>
                {<msc> | <hmsc>} endmsc <end>
```

Enclosing of a terminal or a non-terminal in square brackets means that the terminal or non-terminal can be present 0 or 1 times. A vertical line separates alternatives one of which has to be present. In this case it chooses between <*msc*> and <*hmsc*>.

To express a number of occurencies, there are more symbols than the square brackets. It is possible to say that one or a group of terminals or non-terminals (elements) can be present 1 or more times. This is expressed by plus (+). The symbol asterisk (*) expresses that the element can be present 0, 1 or more times. The fragment of the code below is example of using asterisk:

```
<hmsc body> ::=
                <hmsc statement>*
```

4.2 ANother Tool for Language Recognition

ANother Tool for Language Recognition (ANTLR) [14] is a popular tool for constructing some parsers, translators, recognizers, interpreters and compilers from grammatical descriptions. ANTLR was written by Terence Parr and it was published under the BSD license [1]. It can be used in many various because its grammar can be described in ANTLR syntax or a special AST (Abstract Syntax Tree) and it supports multiple target languages such as Java, C#, Python, Ruby, Objective-C, C and C++. Nowadays, there is a sophisticated grammar development environment for ANTLR v3 called ANTLRWorks. This environment was written by Jean Bovet.

ANTLRWorks combines grammar editor with interpreter and debugger for isolating grammar errors. It requires Java 1.5 or later to run and it is available for Windows, Linux and Mac OS X. Source code of the ANTLRWorks is distributed under the BSD license. The ANTLRWorks debugger is a very useful feature because it creates parse tree which displays whole parsing process and it allows using step by step grammar debugging. It helps to understand what the grammar does. Free download and more information are available at [9].

Let's discuss the general idea of language recognition. At the beginning it is important to say that the language which we meet daily is very difficult to recognize because there are many ways to express the same meaning but for this example we can take only one sentence. For example there is sentence: *The girl read a book*. Let's analyze the sentence. The first thing that human brain does is that it recognizes words and then it connects words and finds the meaning of sentence. As you see, for complete recognition it was necessary to do two analyses and separated the language recognition into two parts. At the beginning, it was necessary to recognize words, connect them and recognize the sentence constructions.

The part when words are recognized is usually called the lexical analysis performed by a lexical analyzer, also called a lexer. The lexer reads letter by letter and creates tokens[1] (symbols) at the beginning and at the end of a word, so then it knows where the word starts, ends and it also knows what is the structure of the word (integer, set of letters or a more complicated type). The sentence construction is recognized by a syntactical analyzer usually called parser. The main difference between a lexer and a parser is that the lexer recognizes words in a stream of characters while the parser recognizes the grammatical dependencies in a stream of tokens. When the ANTLR wants to recognize some language, it uses the lexer and then the parser. Since there exist the lexer and the parser, the grammar defines two kinds of rules.

First rules are lexical and define which characters can be found in the word (usually the name of a non-terminal which is described by a lexical rule is in uppercase) and second rules are syntactical and define a order of words.

It was said how to recognize the sequence of characters and how to recognize the order of words, but usually there is needed something to do when the parser recognizes a special sequence of words. For this situation, ANTLR defines actions in the grammar. It is usually a code in a programming language in which the parser is generated. Actions are allowed to make the sense of the recognized language.

4.2.1 ANTLR Example

There is described basic example of a program, which reads and then prints a name of an MSC to the system output. At first there is shown the grammar which is used in syntax according to the ITU-T standard:

```
<first_rule> :=
               msc <NAME> ;
```

There is one rule which defines that the parser reads keyword *msc* and then non-terminal *NAME* (sequence of letters). There is implementation of the same rule in a grammar according to ANTLR syntax:

Testname.g

```
grammar Testname;

options { language = C; }
                                          //rule
first_rule returns [char* name]:
    'msc' NAME ';'
    {                                     //action
    printf("The name was detected\n");
    $name = (char*) $NAME.text->chars;
    }
;
```

1. Token - includes start/stop indexes into a stream of characters and token's character position.

```
                                        //lexical rule
NAME:
    ('a'..'z' | 'A'..'Z')+
;
```

Let's explain what each part of the grammar means. On the first line, the name of grammar is defined, it has to be the same as a name of a file in which the grammar is saved. In the *options* block, there can be set some options of the parser. In this case there is specified the language in which the parser will be generated. In this example, the parser is generated in C language. The first rule with name *first rule* returns one value (*char* name*) and declares that the read text has to have the keyword *msc* at the beginning. Then it expects the sequence of letters which represents the name of an MSC (non-terminal *NAME*). After that, it expects character *;*.

In the grammar, *NAME* is non-terminal and lexical rule declares that it is a sequence of letters. The length of a name has to be longer than 0 letters. At the end of *first rule*, there is declaration of action. The action definition is between pair of curly brackets. In this action it prints *The name was detected* and assigns the name of the MSC to the return variable. The return variable is returned to the program which called the parser. The following code is an example of the program which calls the parser. This program creates a lexer, a parser then it runs the parser and in the end it prints the returned name of the MSC (*The name of MSC is: <name>*.).

main.cpp

```
#include <iostream>
#include <string>
#include "TestnameLexer.h"
#include "TestnameParser.h"

void main(int argc, char** argv)
{
  std::string filename = argv[1];

  pANTLR3_INPUT_STREAM input =
    antlr3AsciiFileStreamNew((pANTLR3_UINT8) filename.c_str());

  pTestnameLexer lxr = TestnameLexerNew(input);

  pANTLR3_COMMON_TOKEN_STREAM tstream =
    antlr3CommonTokenStreamSourceNew
    (ANTLR3_SIZE_HINT, TOKENSOURCE(lxr));

  pTestnameParser psr = TestnameParserNew(tstream);

  std::string name = (psr->first_rule(psr));
  std::cout << "The name of MSC is: " << name << std::endl;

 }
```

For the parser and the lexer generation, it is necessary to run command:

```
java org.antlr.Tool Testname.g
```

Then it has to be compiled. There is the command for compilation:

```
g++ TestnameParser.c TestnameLexer.c main.cpp -l antlr3c
```

For running, there is command:

```
./a.out name.txt
```

The *name.txt* is the file where a name of an MSC is stored. There is an example of the *name.txt*. In the file, there is written a text to parse.

```
msc myMsc;
```

The output of the program is:

```
The name was detected
The name of MSC is: myMsc
```

The previous example needs some small extension. This change deallocates a memory which was allocated by the parser creation. It is necessary to call ANTLR's free functions. The following code deallocates all allocated memory (add before the last right curly bracket):

```
psr->free(psr);
tstream->free(tstream);
lxr->free(lxr);
input->free(input);
```

For more information about ANTLR grammar or installation, see [13].

4.3 Compatibility

As was said, the ITU-T standard is not strict in each part of the grammar. There are several places where two files can be different. The main difference can be in an implementation of *document head* because an MSC file can contain all important information without *document head*.

Document head: It occurs before MSC description in an MSC file. The ITU-T standard defines that the document head must contain all messages and timers that possess parameters. The parameters are defined in a parameter data language. The standard defines different ways to describe things which can be mentioned in document head. For more information about the document head see [11].

Since there can exist an application which doesn't use document head, the parser is extended for this non-standard description. So in general, the parser is available to read description according to significant part of the ITU-T standard and some non-standard representations too.

The parser exploits only information which is important to create the MSC structure in the SCStudio. For example: in document head, references can be defined by keyword

reference. The parser is able to read it but this information is not necessary because the parser can notice this information when it reads reference node.

In the part of HMSC and bMSC recognition, the parser is fully compatible with the standard and allows two kinds of bMSC descriptions. Now, there is small example why the parser has to be more general than it is necessary for working with files which were created by the SCStudio. When the SCStudio wants to represent information about message ordering in coregions, it uses only the *before* expression. It is quite easy and expression power is not decreased because each *after* expression can be rewritten as *before* expression. But the parser has to be compatible with an application which uses *after* expression. It is able to work with both kinds of message ordering representations.

It is not easy to make the parser compatible with some drawing application because there are applications which use extra information which is not in the standard notification. One way to store extra information and not to break the standard is to store the information in comments.

There are several programs which allow to draw MSC diagrams but some of them do not support textual file according to the ITU-T standard at all. The parser is not compatible with these applications. The Telelogic SDL Suite, The MSCan and SOFAT are applications which are widely used to create MSC diagrams and support the ITU-T standard.

4.3.1 The Telelogic SDL Suite

The Telelogic is being developed by IBM. It is a software development solution which supports the ITU-T Specification and Description Language (SDL) standard. It uses bMSC notification which is called event-driven (more information about event-driven notification is mentioned in section 5.1). For more information about The Telelogic SDL Suite see [10].

This application allows to export two types of textual files. The first export type is without extra information. The parser is compatible with this representation. The second export type contains extra graphical information. The problem with this export type is that there is used non-standard comment syntax (comments syntax as C/C++ languages) for storing these information.

Since the SCStudio and the parser are in development, there is activity to add syntax of C/C++ comments. After that, the SCStucio will be able to work with MSCs which were created in The Telelogic SDL Suite.

4.3.2 Message Sequence Charts analyzer

Message Sequence Chart analyzer (MSCan) is application for mistake detection in message sequence graph (simpler version of MSC). The application offers three interfaces: graphical (application with graphical interface), textual (textual console) and web interface. For more information see [7].

When this application was tested, two problems were datected with compatibility. The first problem is that MSCan creates non-standarded bMSC textual file. The application does

not use a mandatory element. The ITU-T standart declares that before behaviour description of each instance, there must be present a line: *<Name of instance>: instance;*. The second problem is that MSCan uses keyword *expr* and the ITU-T standard does not declare what this keyword means.

For better compatibility with this application it is necessary to find out what the keyword *expr* means and then the grammar can be extended so that the parser recognizes this notification. The problem that the mandatory element does not occurred is not too crucial because there is a way to find out this information. The solution can be function extension (in the case there was read new name of instance, the function creates new instance with the name which was read) which is working with the name of instance.

4.3.3 A Scenario Oracle and Formal Analysis Toolbox

A Scenario Oracle and Formal Analysis Toolbox (SOFAT) is a formal toolbox for working with MSC diagrams. This application is distributed as free software with its own licence (for more information about the licence see [6]). This application offers graphical interface and for running, there is required a version of the JDK (Java Development Kit) greater than 1.2. For more information about SOFAT see [5].

A textual file which was exported from SOFAT is similar to a text file from MSCan. Since the application uses non-standard keyword *expr* which is not mentioned in the ITU-T standard, the parser is not compatible with this application.

Because the keyword *expr* is mentioned in several applications, the further development of the parser will be focused on extending the grammar and making the Context structure compatible with applications which use this non-standard keyword.

Chapter 5

Parser Design

This part describes the whole development process. At the beginning there is description of an input file (what the structure of file is). Then the parsing process is described (what the parser does and what it expects when it reads the input file). The following section of the chapter describes what was necessary to create because there is a need to store context information during the parsing process. Also there is description of the main parser functionality. The following section describes what the parser does when it recognizes some inconsistency during the parsing process. In the end of this chapter, there are mentioned problems which occurred during the development.

5.1 Input and Output

The structure of a textual MSC file is described in this part. Two kinds of bMSC representations (differences between them) and a HMSC representation are shown. The whole structure of MSC document is discussed in section 4.3 (Compatibility) because some applications break the standard.

The first possibility to describe bMSC (known as event driven) is that the name of instance is mentioned at the beginning of each line. The fragment of instances descriptions can be unordered (one line can describe behaviour of the first instance and the next line can describe behaviour of another instance). Before the description of the instance behaviour starts, initialization of instance must be present (*A: instance;*). Then follows the description of behaviour on instances. Each instance should be ended (*A: endinstance;*) and after that there must not be present any description of behaviour on the instance. There follows an example of this notification:

```
msc FirstNotification;
A: instance;
B: instance;
A: in A,1 from found;
A: out request,2 to B;
B: in request,2 from A;
A: endinstance;
B: endinstance;
endmsc;
```

The second notification of instance description defines instance behaviour in one place (known as instance driven). At the beginning there is defined which instance is described and then there follows behaviour definition of the whole instance. When the instance has

finished, there follows a description of another instance. The difference between this and the first notification is that there is not mentioned a name of an instance at the beginning of each line. There is example of the second type of notification:

```
msc SecondNotification;
inst A;
inst B;
A: instance;
   in A,1 from found;
   out request,2 to B;
   endinstance;
B: instance;
   in request,2 from A;
   endinstance;
endmsc;
```

There is only one way to describe an HMSC. A name of node is mentioned at the beginning of the line (without initial node). At the end of the line, there is list of nodes which should be connected such as successors (without end node because it is the last node in an HMSC). There follows an example of HMSC notification:

```
msc HmscNotification;
initial connect L0;
L0: connect L1, L2;
L1: reference NAME connect L2;
L2: reference NAME1 connect L3;
L3: final;
endmsc;
```

5.2 Parsing

When the parser starts to read a file, it reads the document head (name of document, options, etc.) and then descriptions of MSCs. Each MSC description starts with the MSC name. After that, the parser creates a bMSC or an HMSC. If a bMSC is described in the textual form, the parser works as follows:

1. The parser expects start of instances. When it has read the beginning of instance (for example: *A: instance;*), it creates an instance with a name which is mentioned at the beginning of the line (in this case: *A*) and adds the instance to a map of instances in the Context structure.

2. The parser expects definitions of messages and coregion areas. When it has read a message, it recognizes if the line describes a complete message, incomplete message, coregion start, or coregion end. In the case the line describes:

   - incomplete message, the parser creates the message and continues reading;

- complete message (it is described in two lines), the parser calls function which finds out whether the second event of the message has already been created or not;
  - If the event has not been created, the parser creates a message and adds it to the multi map of messages. One of the message nodes is from the textual file description and the other has been created previously. The event which has been created previously is added to the map of events (future events).
  - If the event has been created previously. The parser creates the new event and replaces an event in a suitable message. Then the parser removes the event from the future events.
- coregion start, the parser creates coregion on the instance and adds the name of instance to a list of instances which have got non-finished coregion;
- coregion end, the parser removes instance form the list of instances which have got non-finished coregion;

3. When the parser reads the end of an instance, it does not expect any description of behaviour about this instance.

If the textual file describes an HMSC, the parser works following:

1. The parser reads the name of HMSC.

2. The parser expects a start node and its successors (nodes which should be connected to this node). When it has read the start node and its successors, it creates start node and stores successors to a map.

3. The parser expects a nodes descriptions. When the parser has read another node. It creates the node and then it looks into the stored map and creates connections between the node and nodes from the map where are stored nodes which have the current node set as a successor.

5.3 Grammar

The parser should accept a textual MSC file according to the ITU-T standard. The base of the grammar is built by the ITU-T grammar but some parts of the grammar were modified to improve compatibility and solve ambiguous behaviour. The semantic actions are created in C++ code.

Due to the fact that the grammar is broad and many difficult actions leave it unreadable, it was necessary to create very simple actions. That is the reason why there is created helpful structure. This structure is called Context structure, because there are stored information which contain context of the read MSC. Since there is the Context structure, actions contain only calls of functions, so actions have got only a few lines. For more information about the Context structure see section 5.4 where the structure is discussed.

In this part, there is explained representation of an MSC. As you know, there are two kinds of MSCs (bMSC, HMSC). The description of an MSC is mentioned in *message sequence chart* rule. Each MSC description has to contain the keyword '*msc*' and then there must follow an MSC name. But there can be mentioned some other information (for example information about time), so the name of MSC and extra information are covered to non-terminal called as *msc head*. After *msc head* there is the description of an MSC. Since there are two possibilities, bMSC is represented as non-terminal *msc* and HMSC as non-terminal *hmsc*. Every description of an MSC has to be ended with keyword '*endmsc*' and non-terminal *end*. Non-terminal *end* consists of optional comment and semi-colon. The following code describes non-terminal *message sequence chart* in ANTLR syntax:

```
message_sequence_chart:
   'msc' msc_head (msc | hmsc) 'endmsc' end
;
```

Since it is necessary to initialize the Context structure before the MSC starts and to check some read information in the Context structure after an MSC definition, there are created actions in the grammar. There is the same rule with actions:

```
message_sequence_chart:
   {
     init(context);
   }
   'msc' msc_head (msc | hmsc) 'endmsc' end
   {
     msc_was_read_fun(context);
     check_collections_fun(context);
   }
;
```

Function *init* initializes memory in the Context structure. Function *msc was read fun* does everything what is needed when the MSC was read. Function *check collections fun* checks if the collections in the Context structure are empty.

5.3.1 Grammar for bMSC

The bMSC is described by non-terminal *msc*. When the running comes into it, there is a need to create bMSC in the Context structure before the MSC will be read. Function *new bmsc fun* is called, before *msc body* is executed.

```
msc:
   {
      new_bmsc_fun(context);
   }
   msc_body
;
```

Since bMSC can be described in many lines, it is necessary something that can be called recursively. For this functionality, there is a non-terminal *msc body*. Because a textual file of MSC can contains comments, non-terminal *msc statement* separates comment lines and MSC's lines.

```
msc_body:
  (msc_statement)*
;

msc_statement:
     text_definition | event_definition
;
```

In one bMSC representation, there is a name of an instance at the beginning of a MSC's line. When reading of the name is finished *set instance name fun* function is called. After the name, there follows a colon and then the part which describes some event. These events can be separated into two groups. The first is *orderable*, the second is *non orderable*. The *orderable* group contains message events and incomplete message events. The *non orderable* group contains start coregion, end coregion, new instance and end instance.

```
event_definition:
  NAME
  {
  set_instance_name_fun(context,(char*)($NAME.text->chars));
  }
  ':' instance_event_list
;

instance_event_list:
   (instance_event)+
;

instance_event:
   orderable_event | non_orderable_event
;
```

orderable event can contain an optional part which defines the name to unambiguous identification in case there is message relations (before, after). Then there follows the type of message which can be *message event* (for representation complete message in MSC), *incomplete message event* (for representation incomplete message in MSC) or time description. After that there can be mentioned optional part which describes ordering of events. In this rule, there are actions to set the event name (*set event name fun*) and to set relations in case they are defined (*add before relation fun*, *add after relation fun*).

```
orderable_event:
   ('label' NAME end
   {
     set_event_name_fun(context,(char*)($NAME.text->chars));
   })?
   (message_event | incomplete_message_event | timer_statement)
   ('before' order_dest_list
   {
     add_before_relation_fun(context);
   })?
   ('after' order_dest_list
   {
```

```
      add_after_relation_fun(context);
    })? end
;
```

Non orderable event separates four types of events (*start coregion*, *end coregion*, *instance head statement* and *instance end statement*). There is an action for each grammar statement in this rule.

```
non_orderable_event:
    | start_coregion {start_coregion_fun(context);}
    | end_coregion {end_coregion_fun(context);}
    | instance_head_statement
    {
      new_instance_fun(context);
    }
    | instance_end_statement
    {
      end_instance_fun(context);
    }
;
```

The fragment of the code below describes a rule for coregion beginning recognition (*start coregion*). This rule contains keyword *concurrent* and non-terminal *end*.

```
start_coregion:
   'concurrent' end
;
```

5.3.2 Grammar for HMSC

HMSC can be described in many lines too. There exists similar non-terminal *hmsc body* and *hmsc statement*. *hmsc statement* can be present many times. The non-terminal is also rewritten to non-terminals *text definition* or *node definition*.

```
hmsc:
    {
      new_hmsc_fun(context);
    }
    hmsc_body
;

hmsc_body:
    (hmsc_statement)*
;

hmsc_statement:
    text_definition | node_definition
;
```

node definition separates the HMSC node into three groups (*initial node*, *final node* and *intermediate node*). There is code of rules:

```
node_definition:
   initial_node | final_node | intermediate_node
;
```

The initial node has not got a name because every HMSC has only one. After keyword *initial* there is a non-terminal *connection list*. *connection list* represents list of HMSC nodes which are connected to this node as its successors. The *final node* has got a name for identification and it has not got *connection list* because it is the last node of the HMSC.

```
initial_node:
   'initial' connection_list end
   {
     new_start_node_fun(context);
   }
;

final_node:
   NAME
   {
     set_node_name_fun(context, (char*) $NAME.text->chars);
   }
   ':' 'final' end
   {
     new_end_node_fun(context);
   }
;
```

The *intermediate node* is a little more complicated (a fragment of code below) because a non-terminal *intermediate node type* can create three types of nodes (*reference node*, *connection node* and *condition node*). In the action, there has to be recognized which node has been read.

```
intermediate_node:
   NAME
   {
     set_node_name_fun(context,(char*)($NAME.text->chars));
   }
   ':' intermediate_node_type connection_list end
   {
     switch (get_node_type_fun(context))
     {
       case 0:
         new_reference_node_fun(context);
         break;

       case 1:
         new_connection_node_fun(context);
         break;

       case 2:
         new_condition_node_fun(context);
         break;
```

```
          default:
            bug_report(context, "Internal Error 16: ...");
          }
      }
  ;
```

This solution creates the node after the successors are known. The parser reads the node and it remembers node's type which was read. Then it reads nodes which should be connected to this node (successors) and after that, it creates the node.

Another possible solution is that the creation of the node would be performed after congruous non-terminal and successors of this node would be added after they are read (in intermediate node).

The intermediate node type separates two kinds of nodes (*timeable node* | *untimeable node*). The *timeable node* contains *reference node* and the *untimeable node* contains *connection node* and *condition node*.

5.3.3 bMSC Line Recognition

There is described how the parser regocnizes a bMSC line. For example: *A: concurrent;*. The analysis starts in *msc statement* rule.

1. In the *msc statement*, the parser recognizes that the line is not a *text definition* (the non-terminal *text definition* consists of the keyword *text* and a stream of characters which is ended by a semi-colon)

2. The parser checks if the line is a *event definition*. In *event definition*, the parser recognizes the name then it performs the action (*set instance name fun*). After that, it reads colon and checks if the rest of the line corresponds to *instance event list*.

3. The parser looks at the *instance event*. It recognizes that the rest of the line does not correspond to the *orderable event*, so the parser checks the non-terminal *non orderable event*.

4. It looks if the rest of the line corresponds to the *start coregion*. In the *start coregion* rule, the parser reads keyword *concurrent* and then a semi-colon. The rest of the line corresponds to *start coregion* and so the parser performs an action (*start coregion fun*).

Now, the parser knows that it read the whole *msc statement* and then it continues with the next *msc statement* (with the next line).

5.4 Context Structure

There are several reasons to create the Context structure. One of them is that the grammar is unreadable when large actions are into it. The second reason is that it is necessary to have

global knowledge. The last reason is that the SCStudio is written in C++ and uses objects but parser is generated in C language and it can not work with objects.

The global knowledge means that it is necessary to know information which were read. For example: when there is event of complete message, it is necessary to know whether the second event of complete message has been read or not. Another situation is when the parser performs a line with HMSC node. It has to store nodes which should be connected to this node and they have not been created yet.

In the Context structure, there are variables to store useful information about MSC which is currently being read. The variables can be devided into two main groups. The first group is used independent on the type of an MSC (bMSC, HMSC) which is being read. For example, there are *nonpointed*, *msc name*, *element name*, etc. The second group depends on the type of an MSC. For example: *coregion area finished* (bMSC), *messages* (bMSC), *connect name* (HMSC)...

The first group can be divided to variables which are used to store general information (dependencies between MSCs) and variables which are used to store information during one description of MSCs. For example: variable *nonpointed* stores names of MSCs on which does not exist a reference (stores general information). Variable *element name* stores name of an instance in case bMSC or name of a node in case HMSC (local information about one MSC).

The second group can be devided to variables which are used to store information about bMSC and variables which are used to store information about HMSC. Variable *coregion area finished* stores information about areas in bMSC. For example: variable *connect name* stores information about successors of an HMSC node.

In following sections, there are discussed what happens when the parser calls some methods. There are descriptions of main functionality of the parser. For more information see [2]

One described functionality can contains more similar methods. For example: complete message contains two methods (*message input fun* and *message output fun*).

5.4.1 Complete Message

Every complete message consists of two parts in the textual form. One of them is send and the other receive. Methods for creation send (*message output fun*) and receive (*message input fun*) are very similar, so there is description what has to be done when it is read. Functions' work flow:

1. separate a name of a message and a unique identification;

2. find an instance by the name which is stored in the element name in the Context structure;

3. look if it is necessary to create new area (*strict order area*) on the instance;

4. look whether the node was created previously or not

- true: find message and set an event of message;
- false: create a new message;

5. add the event into the map (*named events*) in case the event is labeled.

5.4.2 Incomplete Message

Incomplete message has only one event. This event can be send if it is lost message (*incomplete message output fun*) or the event is receiver if it is found message (*incomplete message input fun*). Methods for creation lost and found messages are very similar, so there is description what has to be done when it is read. Functions' work flow:

1. separate a name of a message and a unique identification;
2. find an instance by the name which is stored in the element name in the Context structure;
3. look if it is necessary to create a new area (*strict order area*) on the instance;
4. create a new message;
5. add the event into the map (*named events*) in case the event is labeled.

5.4.3 Message Relation

Messages can have relations defined between themselves. This relations are described by two keywords (before, after). Methods for before (*add before relation fun*) and after (*add after relation fun*) relations are very similar. Functions' work flow:

1. try to typecast current event to the *CoregionEvent*;
2. look if the event exists.

5.4.4 Creation of HMSC Node

As was said, there are three types of HMSC nodes, not including start node and end node. There are small description and work flow of each method.

Connection node connects node and its successors. The work flow of *new connection node fun* function:

1. checks if the node is currently exist in an HMSC
   - true: error message reporting (Error 15);
2. creates a new connection node and adds it to the map of HMSC nodes;
3. creates connections between the node and its successors.

Condition node describes when the path is running in an HMSC diagram. The work flow of *new condition node fun* function:

1. checks if the node is currently exist in an HMSC
   - true: error message reporting (Error 15);
2. creates a new condition node and adds it to the map of HMSC nodes;
3. creates connections between the node and its successors.

Reference node refers to another bMSC or HMSC. The work flow of *new reference node fun* function:

1. checks if the node is currently exist in an HMSC
   - true: error message reporting (Error 15);
2. creates a new reference node and adds it to the map of HMSC nodes;
3. checks if the MSC on which is referred is currently existed
   - true: creates reference
   - false: add a record into the map of future references;
4. creates a new reference node and adds it to the map of HMSC nodes;
5. creates connections between node and its successors.

5.4.5 Collection Checking

In the Context structure, there are some collections which are used to store information that was read. The first checking is executed in the end of an MSC (*check collections fun*) and the second in the end of a file (*check references fun*). Functions checks whether the collections are empty or not. In the case that some collection is not empty, the function reports error. Functions' work flow:

1. check suitable collections.

5.4.6 MSC Finished

When a MSC is finished, there is a need to set and check references. Function *msc was read* is responsible for this. Function's work flow:

1. inserts read MSC into the *mscs*;
2. finds references to MSC;
3. looks if there exists reference to this MSC
   - true: sets references
   - false: adds the MSC into the set of nonpointed MSC.

5.4.7 Returning MSC

There is one function which is called when the textual file has finished (*get total msc fun*). It returns structure (*s_Msc*) which is used only in this place. Structure is created because the parser can not work with *MscPtr* (what is smart pointer of *Msc*). Function's work flow:

1. checks how many non-referred MSC was read;
2. finds MSC in mscs;
3. typecasts *MscPtr* to *s_Msc*;
4. returns *s_Msc*.

5.5 Error message reporting

Now, there is created the parser and there is known what an input file should contain. In this section, there is described what happens when the file doesn't contain required parts. There are two types of mistakes when the parser reports error message.

The first is when the parser can not read a character in an input file. Messages of this type are reported to the system error output.

The second mistake occurs when the data in a textual file is not written according to the standard. These mistakes can be devided in two groups. The first group breakes the standard in word ordering. It means that there is word missing or words are in incorrect order. These mistakes are recognized by the grammar definition. The second group breaks the standard in ordering of MSC decription. These mistakes are recognized by the Context structure. For example: the parser has read the whole MSC description and then it calls *check collections fun* function. The error message is reported if some of the collections are not empty. This situation occurs when a BMSC description has not got definition of one event in complete message, an HMSC node has got successor which has not been defined in the HMSC description, etc. This mistakes are reported to output which the user specified in the parser parameter.

5.6 Problems

During the development of the parser, there occurred several types of mistakes. They were usually small problems which were solved step by step. In this section, there is described where the problems occurred and some crucial problems are described in more detail. At the beginning, mistakes were caused by working with ANTLR especially when we started to generate first parsers in C language because materials and tutorials, which we used, generate Java code. The first crucial problem was with the target language of the grammar. There was problem to set the target language to C++. So, it was decided to generate parser in C language and write actions in C++ language. Then the parser would be compiled with C++ compiler. Linux compiler (gcc) has not got problem to compile the generated code but

Windows compiler (Microsoft Visual C++ .NET) has got problem with C++ code because ANTLR enclosed actions (C++ code) into an extern C block because the target language was set to C language. This problem had to be solved because the parser has to be platform independent. The resolution of this problem is that there was created the Context structure where declarations of functions are compatible with C syntax. Since the grammar contains only functions calls, the actions of parser meet with C compiler requirements and the Context structure which is compiled with C++ compiler can work with objects from the SCStudio.

The next troubles were about returning value. The parser needs to return object *MscPtr* (smart pointer to the *Msc*). Since the parser can not work with objects but it can work with a structure, there was created *s_Msc* structure. The *MscPtr* can be typecast to *s_Msc* structure. The generated code in C language can work with structure and in C++ code it can be typecast back to *MscPtr*.

The following problem occurred when it was necessary to report error message. Since the SCStudio has special function for error reporting, the parser has to get object as a parameter. The solution is not difficult because ANTLR is prepared for getting a parameter for a parser. ANTLR allows parameters for each syntactical rule in the grammar and so there was added the parameter to declaration of rule (type of parameter and parameter name enclosed in square brackets follows the name of non-terminal). But the parameter is an object and the parser can not work with it. This problem was resloved like the previous one (return value). *s_Z120* structure was created (*Z120* is object which is necessary for error message reporting).

Chapter 6

Conclusion

The parser is being developed within the scope of the SCStudio project. Before the parser was created, there were two ways to create an MSC data structure, used by verification algorithms. The first was a drawing editor and the second was writing the whole structure in a programming language. Now, the parser allows the third way. It is a big development step of the SCStudio project because the parser brings compatibility with other drawing editors and makes testing easier for the developers. The advantages for developers are a little bit hidden but they are also important because it allows to increase the number of MSCs for testing easily and the time which is necessary to create a test file is shorter. The parser can make testing superior and so it can make the development time shorter.

Nowadays, the parser provides the main functionality which the SCStudio needs. It allows to read both types of bMSC notification and notification for HMSC description. In one textual file, it allows description of more MSCs which are combined in one HMSC. The parser can read textual file according to ITU-T standard. When the textual file is not written according to the standard or contains construction mistakes, the parser reports error messages to the error output.

The development of the parser will also continue in the future. The next progress will be focused on compatibility extension and adding support for new parts of the SCStudio (for example: working with time information). The compatibility will be extended with accepting extra comments recognition and adding compatibility for MSCan and SOFAT (drawing applications). Probably, the comments will be used to store graphical information for drawing applications from the SCStudio project. The following development will be also focused on improving user friendliness.

Bibliography

[1] The ANother Tool for Language Recognition: License. [online], [cited April 14, 2009]. Available at: < http://antlr.org/license.html >.

[2] The Sequence Chart Studio: Context.cpp. [online], [cited May 10, 2009]. Available at: < http://scstudio.svn.sourceforge.net/viewvc/scstudio/trunk/src/data/Z120/Context.cpp?view=log >.

[3] The Sequence Chart Studio: msc.h. [online], [cited May 10, 2009]. Available at: < http://scstudio.svn.sourceforge.net/viewvc/scstudio/trunk/src/data/msc.h?view=log >.

[4] The Sequence Chart Studio: SVN. [online], [cited May 10, 2009]. Available at: < http://sourceforge.net/projects/scstudio/ >.

[5] A Scenario Oracle and Formal Analysis Toolbox: Homepage. [online], [cited May 5, 2009]. Available at: < http://www.irisa.fr/distribcom/Prototypes/SOFAT/index.html >.

[6] A Scenario Oracle and Formal Analysis Toolbox: License. [online], [cited May 5, 2009]. Available at: < http://www.irisa.fr/distribcom/Prototypes/SOFAT/Licence.txt >.

[7] *MSCan*. rwth-aachen.de, [cited May 5, 2009]. Available at: < http://aprove.informatik.rwth-aachen.de/~kern/index.html >.

[8] J. Babica. Message Sequence Charts properties and checking algorithms. [pdf], 2009.

[9] Jean Bovet. *ANTLRWorks*. [cited May 7, 2009]. Available at: <http://antlr.org/works/index.html>.

[10] IBM Corporation. The Telelogic SDL Suite. [online], [cited May 5, 2009]. Available at: < http://www.telelogic.com/ >.

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

[12] ANF DATA spol. s r.o. and Masaryk University the research center Institute for Theoretical Computer Science (ITI), Faculty of Informatics. The sequence chart studio. [online], [cited May 5, 2009]. Available at: <http://scstudio.sourceforge.net/>.

[13] PARR Terence. *The definitive ANTLR reference : building domain-specific languages*. Pragmatic Bookshelf, Raleigh, N.C, 2007.

[14] PARR Terence. ANother Tool for Language Recognition. [online], [cited April 14, 2009]. Available at: < `http://antlr.org` >.

Appendix A

Contents of Attached CD

The attached CD contains the following items:

- a PDF and LaTeX version of the thesis;
- source code of the parser (Z120.g, Context.cpp, Context.h).