EpiSpin: an Eclipse Plug-in for Promela and Spin

Master's Thesis

Bob de Vos

EpiSpin: an Eclipse Plug-in for Promela and Spin

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bob de Vos
born in Rotterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
http://www.se.ewi.tudelft.nl

EpiSpin: an Eclipse Plug-in for Promela and Spin

Author: Bob de Vos
Student id: 1312979
Email: B.deVos-1@student.tudelft.nl

Abstract

EpiSpin is an Eclipse plug-in that provides an editor in which *Promela* models can be developed and maintained. This editor includes syntax highlighting, reference resolving, code folding and an outline view to increase the readability of the code. It also provides instant error checking of syntactic and semantic errors and content completion for faster development of *Promela* models. To perform simulation and verification runs using SPIN, an interface is created in which the options can be specified and such a run can be started. The results are propagated back to the console window in EpiSpin. To check the interaction between processes, global variables and channels a graph can be created that shows the communication flow of a *Promela* model. This is called the Static Communication Analyzer.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Ir. C. Pronk, Faculty EEMCS, TU Delft
Committee Member:	Dr.ir. D.H.J. Epema , Faculty EEMCS, TU Delft
Committee Member:	Drs. L.C.L. Kats, Faculty EEMCS, TU Delft

Preface

This master's thesis presents EpiSpin: an Eclipse plug-in for the Promela language. For almost a year I worked on this project, experiencing up and downs. I learned a lot in the fields of software engineering and grammar engineering, but I also learned a lot about myself, getting to know my strong and weak points.

I could not have created this work without the help of some people. I would like to thank my daily supervisor, Kees Pronk, who encouraged me to work hard to make EpiSpin the best Promela tool that is available right now. He motivated me when I was stuck and helped me to get back on track when I was getting off. Next, I want to thank Lennart Kats, who especially helped me with the implementation specific part of my work, unveiling new Spoofax constructions that were just implemented and helped me out in times of need.

My personal highlight of this project was the possibility to present my work abroad. Speaking at the SPIN Workshop on Model Checking Software in Snowbird, Utah, USA was a great opportunity to develop myself on many different levels. I want to thank the Software Engineering department and the MoDSE group for supporting me on this trip.

This thesis concludes a period of five years of studying in Delft and now it is time to start a new adventure. I could not have been successful the past five years if my parents did not support me so well. For that, I want to thank them deeply.

Bob de Vos
Delft, the Netherlands
December 7, 2011

Contents

List of Figures

Chapter 1

Introduction

Promela [15] is a process modelling language in which verification models of hardware and software systems can be described. As these systems can grow very large, so can the models. Examples of such large models can be found in [19, 35]. This calls for state-of-the-art Integrated Development Environments (IDE) in which these models can be developed and maintained.

Current most popular editors for building *Promela* models are *Xspin* and its successor *iSpin* [14], both created by the *Promela* developers. These programs provide a plain text editor in which *Promela* models can be written. Coding in plain text without having any basic editor services like syntax highlighting is not considered very efficient. However, it is still the most popular tool used for developing *Promela* models. The popularity of this tool comes from its integrated support for the Simple Promela Interpreter, SPIN. SPIN can perform a simulation of a model, giving an overview of the statements that are executed or start a verification run in which several correctness criteria are verified. Integrating SPIN with a state-of-the-art IDE would create the perfect tool for both starting and experienced *Promela* developers.

Developing a new IDE from scratch would take a lot of time and effort. Several IDE's already exist that can be extended with a new language, such as Eclipse [9] and NetBeans [27]. It could be more efficient to extend such an existing IDE with the new language. For the *Promela* language two plug-ins for the Eclipse framework already exist. The first one is made by Rothmaier [33] in 2005 as part of his PhD work. Unfortunately, this plug-in is not maintained after his promotion and has never been distributed. Another endeavor was made by Kovse and his colleagues [23]. This work is more successful and is currently offered as the official Eclipse plug-in for *Promela*. Although this plug-in provides syntax highlighting, syntactic content completion and static error checking, it does still not use all editor services that are offered by Eclipse. Therefore we wanted to create a new and better Eclipse plug-in.

We created EpiSpin, an Eclipse plug-in with state-of-the-art IDE support for the *Promela* language with integrated support for SPIN. EpiSpin also includes the Static Communication Analyzer, which is a tool that uses the *Promela* source code to derive a graph showing the static communication flow of the model. Eclipse is chosen because of its support for multiple platforms and its support for creating new plug-ins. EpiSpin is developed in a different way compared to the existing Eclipse plug-ins. Instead of developing the plug-in in

Java directly, using the Eclipse plug-in tools, EpiSpin is created with the Spoofax language workbench [22]. This language workbench takes care of the underlying Java classes of the plug-in and allows you to concentrate on the language specific parts. Furthermore, it loads the newly generated plug-in dynamically in the same Eclipse instance, resulting in an agile development environment.

EpiSpin will contribute to a faster process of writing a model, correcting the model and performing a verification or simulation of the model. The various editor services increase the coding speed and maintainability of a *Promela* model and due to the instant feedback writing and correcting the model are merged into a single phase.

This thesis elaborates on the design of EpiSpin and will give you insight in its possibilities and limitations. In the remainder of this chapter a more detailed introduction will be given on the features of EpiSpin. In Chapter 2 background information will be given on the related topics of this thesis, such as *Promela* and SPIN, Spoofax and the traditional way of building Eclipse plug-ins using Java classes. Chapter 3 concentrates on the design choices that are made regarding the main EpiSpin features: the use of macros, the design of the Static Communication Analyzer, the possibilities and restrictions of starting simulation and verification runs from EpiSpin and the last section of this chapter explains which error checks for the *Promela* language will be implemented in EpiSpin. After justifying these design choices, the implementation of EpiSpin in Spoofax will be discussed in Chapter 4. This chapter elaborates on the implementation of the grammar first, followed by a discussion of the different editor services that will be implemented in EpiSpin. Chapter 5 shows how the different components of EpiSpin are tested. Discussion on related work can be found in Chapter 6 and this thesis ends with a concluding chapter in Chapter 7.

1.1 EpiSpin

In the following sections EpiSpin will be introduced by highlighting several aspects. First the editor will be discussed by dealing with the error checks and the other editor services. In section 1.1.2 the interface for performing verification and simulation runs will be introduced and more information about the Static Communication Analyzer can be found in 1.1.3.

1.1.1 The Editor

Error checking

In current *Promela* editors [14, 23] writing a *Promela* model and performing error checking on these models are two separate steps. First the program is written in a simple text editor, in iSpin or in Eclipse using the current Eclipse plug-in [23]. Then, the SPIN syntax checker is called and both syntactic and semantic errors are reported. One of the main features of EpiSpin is that error checking of the model is done in real time and error messages are shown immediately when invalid language constructions are entered.

Syntactic errors are reported directly by the parser, which is generated by Spoofax using the grammar definition. The complete *Promela* grammar of SPIN version 6 [13] will be implemented. This includes `proctype` definitions, “never” claims, repetition and conditional

statements, the use of channels and the recently added for-loop and inline "LTL" formulas. Implementing macros will be much more complicated. Macro definitions and macro calls are handled by the C-preprocessor in SPIN. Since it is not possible to call the preprocessor in real time, EpiSpin will not fully support the use of macros. This means that certain valid *Promela*-code will not be parsed by EpiSpin. Section 3.1 elaborates more on the design choices with respect to macros and gives an overview of which constructs are allowed and which are not.

Feedback on semantic errors is provided instantly as well. According to the source code of SPIN, there are approximately 200 semantic error messages that can be fired. Due to time restrictions it is not possible to include all of them, but most of these errors will be implemented. Section 3.4 elaborates on this issue. These error messages appear immediately when an erroneous string is entered. This means that users get direct feedback on the code they are writing, leading to fast error correction. The erroneous code (the `goto` statement in this case) is underlined by a red line and an error mark is shown in the margin. When the mouse is hovered over the red line or over the error mark, the error message is shown.

These error messages are specific to the error made and can also include references to user defined variables. How this is implemented in the source code of EpiSpin is explained in Chapter 4.

Editor services

Besides error messages, other editor services will be implemented as well. The first important feature is syntax highlighting. This means that keywords, identifiers, strings and other functional different parts of the code have a different font and color. This improves the readability of the model. For clarity and overview purposes, an outline view and code folding will be implemented. The outline view gives an overview of all processes and variables in the *Promela*-code. Code folding gives the opportunity to collapse multi-line language constructs into one single line. This will be possible for small blocks (like `do`- and `if`-statements) and for larger parts (like proctypes). These two features are especially useful for large programs, where one might to search through thousand lines of code to find a particular code fragment.

To keep track of the identifiers in a program, reference resolving will be implemented. With reference resolving, the declaration of an identifier can be found by clicking on an instance of that identifier. The last editor service that will be implemented is code completion. Both syntactic as well as semantic code completion will be implemented. Semantic code completion includes the completion of fixed language constructs. So opening brackets and parenthesis will automatically be closed and after typing `do`, `::` and `od` will be inserted. With semantic code completion, a list of possible identifiers or language constructs pops up, depending on the expected type of the next token. If a label is expected for example, all labels that can be used at that position will be listed. As the user types the token, the list shortens by removing all options that do not match the letters typed.

1.1.2 Verification and Simulation

After writing the model, the model can be checked. SPIN offers two main operations to check a model: verification and simulation. With verification, all possible states of the model are searched to find a particular state or a sequence of states, or to prove that a certain state does not exist. For example, a cycle can be detected in this way. In a simulation run, the program is executed in a non-deterministic way, interactively or by following a predefined trail.

To perform a verification or simulation run in EpiSpin an option file needs to be opened. This file shows the same options as in iSpin in a graphical way. Extra flags can be added to specify extra options such as multi-core options. However, there will be no checks on the validity of these options and the developer himself is responsible for using the extra flags correctly. In addition to current *Promela* editors, EpiSpin will save the changes that are made in the option file. These changes are saved to disk and reloaded when the file is reopened.

When a verification or simulation run is started, the output will be visible in the EpiSpin console window. For simulation runs, iSpin also provides a message sequence chart (MSC), in which the control flow of the program can be seen and a channel reader which shows the content of the channels at every step in the program. In iSpin these features are interactive, meaning one can step through the statements or click on elements of the MSC and the other components will highlight the same element. These interactive features will not be included by EpiSpin. However, there will be a similar static tool which derives a graph from the source code showing the dependencies between global variables, processes and message channels.

1.1.3 SCA

One of the innovative features of EpiSpin is the Static Communication Analyzer (SCA). Examples can be thought of in which it is useful to draw the communication structure of a *Promela* model. In education for example, this is useful for teachers who are grading students *Promela* models. In *Promela* models, processes can communicate with each other using global variables or channels. The SCA generates DOT code [10] which can be compiled to show a graph. This graph contains various shapes that represent variables, channels and processes. Arrows indicate read or write operations (between processes and variables) and send or receive operations (between processes and channels). This graph is generated by analyzing the source code of the model and transforming the different structures into DOT code. More about the design and implementation of the SCA can be found in Section 3.2 and 4.4 respectively.

Chapter 2

Background

This chapter provides more background information on the tools and languages that are used in this thesis. The *Promela* language will be introduced first by giving an overview of the most important language constructs that are used to build a *Promela* program. Then the traditional way of language development with Eclipse is discussed in Section 2.2. In the next section the Spoofax language workbench will be discussed by describing the languages and frameworks that are used in Spoofax and in the last section a comparison of the traditional Java based method and Spoofax will be made.

2.1 Promela and Spin

Promela [16] is a verification modeling language created by Gerard Holzmann [15]. It is used to build *Promela* models which can be interpreted by the SPIN model checker. SPIN can perform random and guided simulations of *Promela* models and it can generate a C program that can perform a verification of the model. Various safety and liveness properties can be verified such as deadlocks, non-executable code and finding non-progress cycles.

Promela programs consist of three different elements: processes, message channels and variables. *Promela* code can be interpreted by the C preprocessor and therefore it is also allowed to use macros in *Promela* models. Before a *Promela* model is parsed by SPIN, macro calls are first expanded by the C preprocessor.

The main part of a *Promela* model consists of proctypes with statements. An instantiation of a proctype is called a process, which is created on initialization of a run or it can be dynamically created by other processes. Statements are executed in sequential order inside a process, but processes can interleave with each other. The next statement that should be executed is chosen non-deterministically from all processes for which the next statement is *executable*. When a statement is not executable, it is *blocked*. Blocked statements are not skipped, but the process is put on hold until the blocked statement becomes executable, which may be the result of events happening in other processes.

Some of the most frequently used statements in *Promela* models are:

- Atomic sequences. When it is wanted that a sequence of statements is executed right after each other without the interference of other processes, this sequence can be

placed in an `atomic{}` or `d_step{}` block. Both are comparable but a `d_step` block is more restrictive by not allowing jumps in and out the sequence and statements in this sequence may not block (whereas a blocked statement is allowed in an `atomic` sequence: atomicity is then lost and is regained when the statement is executable again).

- Selection statements. A selection statement has the following syntax:

```
if
:: sequence of statements
:: sequence of statements
fi
```

  Only one of the sequences in the `if` construct is executed. This sequence is chosen non-deterministically between all sequences of which the first statement is executable. Therefore these first statements are called the *guards*.

- Repetition statements. These look the same as selection statements, but the `if`/`fi` keywords are replaced by `do`/`od`. Again, a non-deterministic choice is made to decide which statement will be executed. After a sequence of statements is executed, the construct is executed again, allowing for repeated execution. This repetition can be stopped when a `break` statement is executed or when a `goto` statement leads out of the repetition structure.

- Inline statements. To use an inline statement, an inline definition is required. When an inline call is executed, it is replaced by the text of the inline definition. If any parameters are used, their actual values from the call will textually replace the formal place holders inside the definition. The following example shows how an inline definition is used to add the value 1 to a variable.

```
inline add(a){
  a++;
}
init{
  int b;
  add(b);
}
```

In *Promela* local and global variables exist. The following datatypes are supported: `bit`, `bool`, `byte`, `pid` (which has the same range as a `byte`, but is used for specifying process identifiers), `int`, `mtype` (which is used for defining symbolic constants to send over message channels) and `unsigned`. It is also possible to declare user-defined data types and to make arrays of data (except for `unsigned` variables). Message channels are used to model the transfer of data from one process to another. They are declared locally or globally and

their declarations consist of a buffer size (how many messages can be stored in the channel) and an enumeration of data types, representing the fields that a message can contain. Message send and receive operations are allowed on channels, as well as different types of poll operations to read values from the channel without deleting the values in the channel. Channel send operations can only be executed successfully when the buffer of the channel is not full. In the verification and simulation options it can be specified which action should be taken when a message is sent to a full channel (blocking the statement or losing the message). A channel receive statement updates the arguments to the values of the message fields that are received, except when a message field is a `mtype` variable. In that case, the message field acts like a guard. When the `mtype` fields match, the receive statement is executed, otherwise it is blocked. Other variables can also act as guards in message fields by including the variable name in a `eval()` construct. This means the variable is evaluated and the receive statement only executes when the message value equals the value of the variable in the receive statement.

An example of a simple *Promela* model is given below. This example sends the numbers 0-9 from one process to another process which will print the numbers.

```
mtype = {msg, ack};
chan ch = [2] of {mtype, int}

proctype Sender(){
int i=0;
do                      //send i and wait for ack before i increased.
:: i<10-> ch!msg,i; ch?ack,eval(i); i++;
:: i==10-> break;
od;
}

proctype Receive(){
int i=0;
do                      //when i is received, print its value.
:: ch?msg,i-> ch!ack,i; printf("%d\n", i);
:: i==9-> break;
od;
}

init{
run Sender();
run Receive();
}
```

In Section 2.3 this example will also be used to demonstrate some Spoofax and Stratego code.

SPIN

The SPIN model checker is a tool used for verification of correctness properties of *Promela* models. It can also be used to perform (interactive) simulations of these models. In Figure 2.1 is shown how a *Promela* model generally is processed using SPIN. When a model is created (optionally using a graphical front-end) the SPIN syntax checker is called first. It returns a list of warnings and errors and optionally it can also return warnings about dubious constructs and give suggestions on possible improvements. Then a simulation can be done to check whether the model behaves as expected. Correctness claims can be checked by the verifier. These correctness claims can specify safety properties (such as invalid end states), liveness properties (such as non-progress cycles) and temporal logic properties. The temporal logic properties are specified in linear temporal logic (LTL) and allows users to make claims that state that a certain condition should always / never / eventually happen. In step 3, SPIN generates a verification program from the model. This verification program is compiled with user specified options (like the type of reduction algorithm that should be used) and is then executed to perform the actual verification. If a claim is violated, the verifier produces a trail of steps that leads to a counter example of the claim. This counter example can be fed to the simulator to inspect the cause of the violation in detail. Different search and storage modes can be chosen for verification runs. The user can choose between breadth-first and depth-first search modes. These searches can be exhaustive, where every state in the state space is checked, or more approximate such as a hash-compact or bitstate, where not all states might be visited.

Several graphical shells are built for SPIN. The SPIN developers created *Xspin* [16] and its successor *iSpin*. Both are developed with the Tcl [28] programming language and the Tk graphical toolkit. Tcl is used because it is relatively easy to build GUI's in and is has excellent support for integration with other software [1]. These properties result in a very interactive environment for exploring simulation and verification results in *Xspin* and *iSpin*. It is possible to step through the output, step back and jump to the corresponding line in the source code by clicking on an output step. This is especially useful for simulation results, since a simulation run with SPIN results in several outputs: an enumeration of the statements that are executed with for every step an overview of the values of the global variables and the contents of the message channels. Furthermore it is possible to see a message sequence chart, which is a flowchart that shows how and when channels are used by processes. *Xspin* and *iSpin* parse this output and show it in different frames: the main view shows the executed statements, and when one is selected, the message sequence chart, global variables overview and channel contents - which are shown in separate frames - are updated to show the information that applies for that step.

Unlike the verification and simulation environment, the editors of *Xspin* and *iSpin* are very basic. In the editors, only plain text can be typed in a default font. *iSpin* also places line numbers in textual form in the source code which makes it hard to copy a fragment of source code from *iSpin* since the line numbers are copied as well.

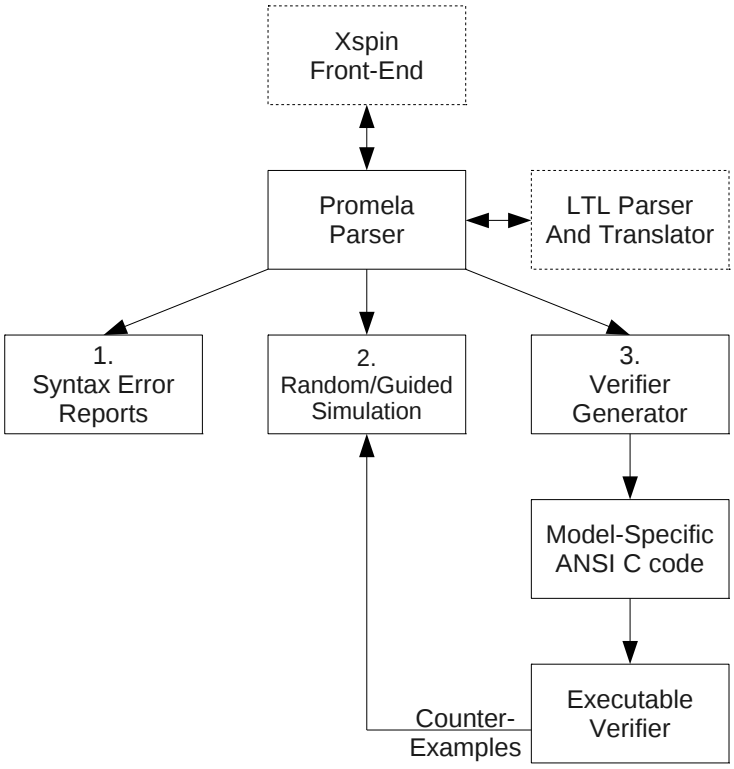

Figure 2.1: The structure of SPIN

2.2 Eclipse and its plug-in environment

Eclipse [9] is an integrated development environment (IDE) with an extensible plug-in system. It is used to develop applications in Java, but by installing plug-ins it can also be used for other languages like C, PHP or Perl. Eclipse is designed to be easily extensible by third parties. In fact, Eclipse itself is composed of various plug-ins. Eclipse and its plug-ins are mostly written in Java. To build a plug-in using Eclipse a new 'Plug-In Project' should be created from the Eclipse menu. The project contains initially one java class and one XML descriptor file: `Activator.java` and `plugin.xml`. The java class handles the instantiation and termination of the plug-in. The descriptor file gives an overview of the different components of the plug-in. These components are created in Java and are integrated in the plug-in by including so called *extension points* in the XML-file. An extension point then points to the Java class that is responsible for handling a certain component. Eclipse provides a list of possible extension points and has several templates for often used components.

Most plug-ins are built using several components. Most frequently used are:

- editors - In most plug-ins the editor component is the most important. The main graphical component of the editor is the large text box where the user will type its code. There is a Java class that has methods to declare the text box and methods to

handle editor services as code folding and content completion.

- wizards - A wizard can be created to guide users through the process of creating or importing new files. In the Java class all the actions that need to be performed are implemented (for example creating a new folder as well).
- views - A view is a visual component within Eclipse, like the package manager (which shows the hierarchy of the workspace), the console and the error log. An example of a view that is more language specific is JUnit (which provides an interface to perform Java unit tests from Eclipse).
- perspectives - A perspective in Eclipse is used to define the initial layout for a plug-in. It describes which views are shown and what their positions are on the screen when the perspective is opened. A perspective can also be opened automatically when a new project is created.
- preference pages - Using the preference page extension point, plug-in developers can add their own preference pages to the general Eclipse preferences window. On the preference page users can specify options to customize the editor services.

Other components include toolbar buttons, pop-up windows and menu entries. Beside all those visual components, extension points can also refer to classes that perform actions, like a class that calls an external program and processes its output.

The existing Eclipse plug-in for *Promela* by Kovse [23] is created using these extension points. Since a *Promela* model is not parsed in this editor (but only scanned for syntax highlighting), it is hard to implement semantic editor services, since nothing can be said about the meaning of the different tokens. Syntactic editor services are more convenient to be implemented, as is done in the plug-in from Kovse.

These editor services are implemented to increase readability, maintainability and coding speed for end users of the plug-in. These editor services need to be written in Java as well. Syntax highlighting, code folding, content completion and the outline view will be discussed in the next paragraphs.

Syntax highlighting is coloring the source code such that groups of tokens with different functions get the different colors. This is obtained by scanning the source code that is typed in the editor. A scanner is called from the editor Java class and it detects different kinds of tokens such as comments, strings and keywords and marks each token with an attribute. These attributes are then interpreted by another class (usually from the JFace toolkit [8]) which is responsible for the actual coloring.

Code folding works in a similar way. Code folding is the ability to collapse multi-line language constructs into one line. First the start and end positions of the language constructs that may be folded are stored. Then this information is passed to the editor class which uses other Java classes to handle the actual folding.

Implementing content completion (showing a pop-up when certain tokens or language constructs are typed) is done in three steps. First the editor should check which part of the string is already typed. Then the appropriate completions should be found by calling a class that extends the `IContentAssistProcessor` class. The possible completions depend on

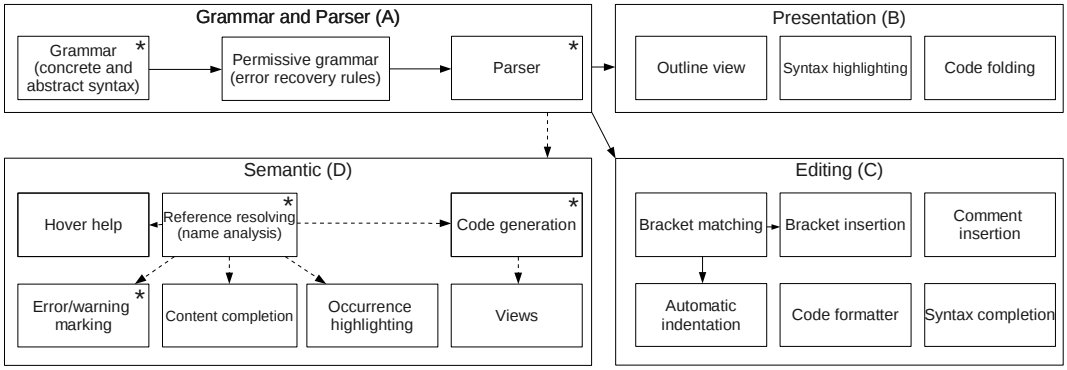

Figure 2.2: Relations between the IDE components. Asterisks indicate components that are also part of general compilers. From [22].

the part of the string that is already typed, but they may also depend on the position of the string: when a language has different scopes for example, not all content proposals might be allowed at all positions. The third step is to actually show those completions which is again handled by the editor class.

The outline view is implemented like any other view. The outline view shows an overview of the structural elements of a program. The Java class that describes this view needs to implement a special interface for outline views. When an editor is selected, it is asked for a class with this interface to build the outline view.

2.3 Spoofax

Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support [22] and can also be used to build IDE support for existing languages. A language workbench is piece of software that integrates several aspects of language development - such as parsing, transformation and code generation - in one environment. Spoofax is loaded as a plug-in in Eclipse. To develop a language, a Spoofax project should be created. Each project can be deployed as an Eclipse plug-in which can be installed on every machine running Eclipse.

With traditional language development, first the compiler and then the IDE is created. Spoofax aims for integrating these two phases by partly deriving editor services for the IDE from the grammar. This relation can be seen in Figure 2.2. This figure gives an overview of compiler components and IDE services. The solid arrows show generative dependencies, which means that one component can be derived from the other, and the dashed arrows show usage dependencies, meaning that one component calls another. The figure shows that the presentation components (i.e. editor services that increase readability of the source code) can be directly derived from the grammar. These editor services can then be manually customized.

```
Module+                                -> Start {cons("Modules")}
Active? "proctype" ID "(" ")"
  Priority? Enabler? "{" Sequence "}" -> Module {cons("Proctype")}
Active? "proctype" ID "(" One-decl (Sep One-decl)* ")"
  Priority? Enabler? "{" Sequence "}" -> Module {cons("ProctypeParam")}
Visible? Type {Ivar ","}+              -> One-decl {cons("Decl")}
ID  VarInit?                           -> Ivar {cons("Var")}
(Step)* Stepfs                         -> Sequence {cons("Steps")}
"do" Options "od"                      -> Stmnt {cons("Do")}
("::" Sequence)+                       -> Options {cons("Option")}
Send                                   -> Stmnt {cons("Send")}
Varref "!" SendArgs                    -> Send {cons("StandardSend")}
```

Figure 2.3: A fragment of the EpiSpin SDF grammar.

Syntax definition

Spoofax uses the declarative syntax definition formalism SDF [11] to define the grammar. SDF grammars combine lexical and context-free syntax into one formalism and it is also possible to define concrete and abstract syntax in the same production rules. A production rule consists of tokens $p_1 \cdots p_n$ which are matched to a symbol s. It is common to annotate a rule with a constructor name, which is shown in the abstract syntax tree (AST) when this rule is matched. When a Spoofax project is built, a parser is generated from the grammar which is run in a background thread and executes every time a change is made. This 'live' parse produces an AST which can be analyzed for semantic editor services. The AST is represented in Spoofax using *terms* of the the ATerm language [4]. A constructor is represented like $ConstructorName(t_1,...,t_n)$, where the argument terms can be other constructors, constants or a list of terms. A list is represented using square brackets: $[t_1,...,t_n]$. Another useful aspect of the ATerm language is the ability to annotate terms with another term. This can be used to annotate identifiers so identifiers with the same name (in different scopes for example) can still be distinguished.

A fragment of the EpiSpin grammar is shown in Figure 2.3. Quoted strings represent terminal symbols and the tokens starting with a capital on the left hand side are called *sorts* and represent the non-terminals that may occur at that position. The first word on the right hand side is the name of the matching sort and the string in `cons` represents the constructor name, which is used the representation of the AST. Constructions followed by `*` and `+` are lists (with at least zero or one elements) and a structure like `{Ivar ","}+` indicates a list of `Ivar`s, separated by commas. Figure 2.4 shows the AST of the following code (which is a simplified part of the code from Section 2.1):

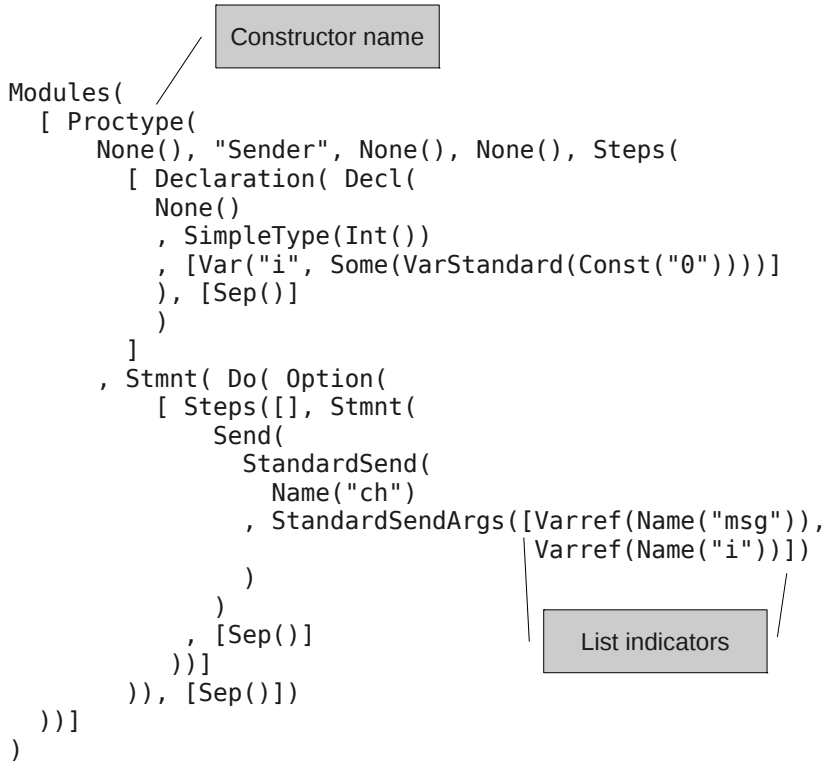

Figure 2.4: An AST of a small *Promela* program parsed by EpiSpin.

```
proctype Sender(){
  int i=0;
  do
  :: ch!msg,i;
  od;
}
```

When the parser is finished, all syntactic editor services that can be directly derived from the grammar (i.e. the editor services that are shown in the presentation and editing block in Figure 2.2) are updated immediately. The semantic editor services are updated after the semantic analysis.

Semantic Analysis

During semantic analysis the AST is annotated with semantic information. This information is used later on when transformations are applied on the AST to define the editor services. Both the analysis and transformations are defined using a domain specific language that is built for Spoofax. This language is a declarative, rule-based editor descriptor language to

define the observer (which is the strategy that is executed every time a change is made in the source code) and to define the presentation and editing services. It also describes what transformations should be used to start the analysis and to perform the semantic editor services. The *implementation* of these transformations is done with the Stratego language [5]. Stratego is a transformation language in which rewrite rules can be specified. These rules have the following form:

```
r: t1 -> t2 where s
```

When a rule r matches term $t1$, this term is transformed into a term $t2$ if the condition s holds. These rules are generally applied to constructor nodes in the AST. An AST can be traversed using Stratego traversal rules. These rules are composed from basic combinators such as sequential composition and several of these traversal rules are provided by the Stratego standard library.

When an AST is obtained from the parser, it will be analyzed first. This analysis forms the basis for all semantic editor services. During analysis, scopes are identified and variable occurrences are bound to their declaration. Other properties of the AST (that can be used for error reporting for example) are identified as well. When the AST above is considered, this analysis phase will annotate some of the variables to get unique identifiers in the AST. When a variable `i` is declared and used in multiple scopes, this renaming process makes sure that the variables have different annotations in both scopes so they can be distinguished when the AST is traversed for defining editor services. In this case, `Var(name, initializer)` is the constructor of a variable declaration and `Name(name)` is the constructor of a variable occurrence. The rewrite rule for the `Var` constructor looks like

```
rename(|type):
  Var(name, init) -> Var(<rename-var>(name, type), init)
```

It matches the constructor `Var` and binds the two terms of the constructor to the variables `name` and `init`. It is transformed into a new `Var` constructor with the same initialization term, but with a name that is returned by the strategy `rename-var`:

```
rename-var:
  (x,t) -> y
    with y:= x{<new>}
      ; rules(
          RenameId :+ x -> y
          TypeOf   :+ y -> t
          VarLookup :+ y -> x
        )
```

This strategy should be called on a tuple with two terms: a name (which is bound to `x`) and a type (which is bound to `t`) as is done in the `rename(|type)` strategy above. This tuple is transformed into a new term `y`, which is the term `x` annotated with a unique string (obtained by calling the `new` strategy) to make sure the term `y` is unique. Furthermore, this strategy stores three *dynamic rules*, which are instantiated on run-time. The transformation

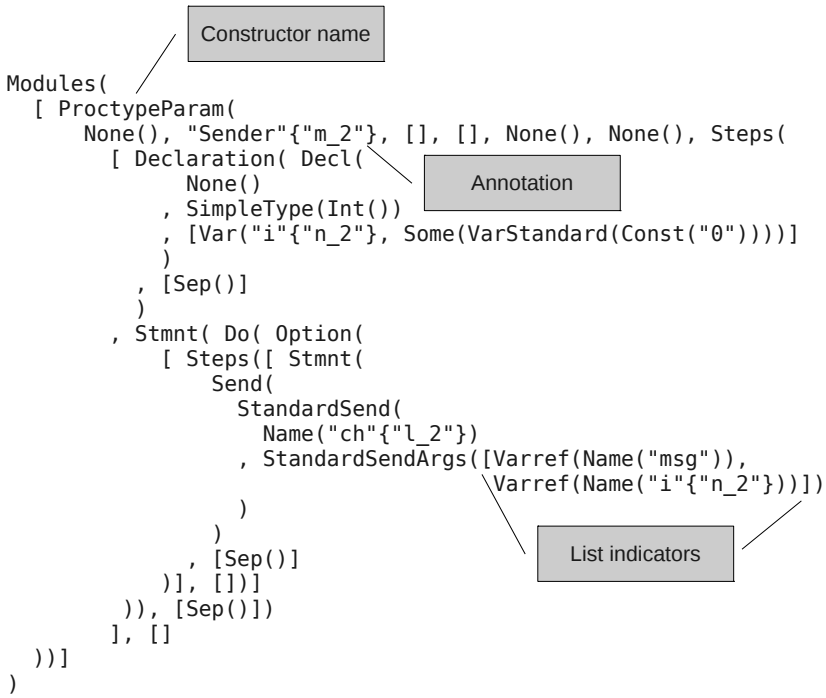

Figure 2.5: The analyzed AST of the small *Promela* program.

from the old name to the annotated name is saved in the dynamic rule `RenameId`. These dynamic rules can be called later on a certain term to get the transformed value. So when an occurrence of the variable (which is represented by the `Name` constructor) is renamed, this will simply look like

```
rename:
  Name(x) -> Name(<RenameId> x)
```

The analyzed AST of the above AST is shown in Figure 2.5. This AST differs from the original one by the added annotations and some refactorings. For example, note that the `Proctype` constructor is transformed in `ProctypeParam` with empty lists for the third and fourth term, representing zero arguments. Error reporting, reference resolving and content completion is now done using this traversed AST.

Semantic editor services

Semantic error checking in Spoofax - as any other editor service - is done using transformations. For error checking the AST is transformed into a list of tuples, each tuple containing an error message (a string) and the position of the error marker (a term). Such a rule should have the following form:

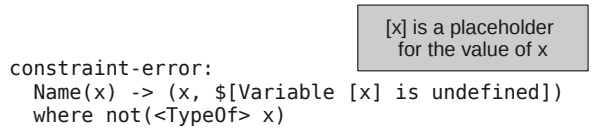

Figure 2.6: Error constraint strategy for undefined variables

```
check:
   context -> (position, $[error message])
   where s
```

The AST is traversed and when a term `context` is matched and the condition of strategy *s* holds, the transformation is performed. The new tuple consists of a term that indicates the position of the error marker and a string (indicated by `$[]`) representing the error message. These tuples are collected by a control rule which is called by the observer in the main editor descriptor file. This observer is executed every time a new AST is created. By pointing the observer rule to the control rule, error checking is performed after every small change in the source code. Besides error checking, warnings could be collected as well. Reporting warnings follows the same steps as reporting errors.

This error mechanism is also used to detect undefined variables. During analysis the variable name in a variable declaration is renamed (as shown in the `rename-var` example on page 14). In this strategy a dynamic rule `TypeOf:+ y -> t` is defined. When an applied occurrence of this variable is found, it is checked if the dynamic rule `TypeOf` succeeds for this variable. If not, this means there is no declaration for that variable and the transformation to show the error should be applied. The Stratego code for generating this error message can be seen in Figure 2.6.

Reference resolving uses a similar construction. In the reference resolving descriptor file a control rule is specified. This control rule transforms the selected term (which is the term representing the code that is clicked on) into its target term using the information that is stored during analysis. When an occurrence of a variable is clicked on, the following strategy is called:

```
editor-resolve:
  (Name(x), position, ast, path, project-path) -> <VarLookup> x
```

where `VarLookup` is the third dynamic rule from the example on page 14.

Content completion is the transformation of an incomplete token or expression into a list of possible completions. When content completion is triggered (which can be done manually or automatically after typing a predefined completion trigger) the control rule for content completion is called and it will start another traversal of the AST, encapsulating the incomplete AST node by the COMPLETION constructor. When the COMPLETION constructor is matched in the next traversal, it is transformed into a list of possible tokens that can occur at that position.

2.4 Comparison of Spoofax and Java based plug-in development

One of the main reasons that Spoofax was built in the first place was that language developers often found themselves writing a lot of standard code in Eclipse. Spoofax has been designed to factor out this standard code into the generic Spoofax libraries. The code that needs to be written by users is mostly language specific and much more compact compared to standard Eclipse Java classes. Developers need to write Stratego code for analysis, transformation and code generation, instead of using several Java frameworks and template engines for these features.

When comparing the parsers, Spoofax generates a scannerless generalized LR (SGLR) parser [36] based on the SDF grammar, while the Java based approach uses regular expressions to match the source code. The SGLR parser supports error recovery automatically, which needs to be implemented manually in Java based Eclipse plug-ins. From the syntax definition basic editor services are derived such as syntax highlighting, outline view and code folding which can be customized by the developer. This leads to an agile development process where the language can easily be extended. Syntax highlighting in Java based Eclipse plug-ins is done by matching predefined strings to the parsed tokens to check for keywords.

Finally, Spoofax loads the new editor dynamically in the same Eclipse instance, instead of opening a second Eclipse instance. This allows developers to test the language and develop the language in one window, which also contributes the agile development process.

Chapter 3

Design

This chapter will focus on the design choices that are made for EpiSpin. The important aspects that needed to be designed from scratch were macro support, handling verification and simulation runs from within EpiSpin and the Static Communication Analyzer (SCA). Macro support is available in *Promela*, but is handled by the C preprocessor before it is parsed by SPIN, which is not possible in EpiSpin. Section 3.1 elaborates on this aspect. Section 3.2 discusses the design choices related to the Static Communication Analyzer: a tool that generates DOT code showing the static relations between processes, global variables and message channels of the current model. The following section discusses how to perform SPIN verification and simulation runs from within EpiSpin and the final section explains which error checks are implemented in EpiSpin.

3.1 Macros

One of the grammar elements that could not be copied one-to-one from the *Promela* grammar definition [13] to the SDF syntax definition of EpiSpin was the macro elements. When the SPIN syntax checker runs, the source code of a *Promela* model is first preprocessed by the C preprocessor before being parsed by SPIN. The preprocessor scans the source code for macro directives and macro calls, evaluates conditional compilation directives, substitutes macro calls and removes all directives. The result of this process is pure *Promela* code without macros. This code is then parsed by SPIN.

When source code is processed by an IDE that is developed in Spoofax, it is always directly parsed according to the SDF grammar of the Spoofax generated parser. This means that the syntax of macro definitions and macro calls should be included in the EpiSpin syntax definition. By including macro statements in the language, the grammar becomes context-sensitive. However, Spoofax only supports context-free grammars, which means that the syntax of macros cannot be included one-to-one in the syntax of *Promela*.

Macro support should be included since it is an important language construct in large *Promela* models. If there is no support, the parser would flag a syntax error and continues parsing after recovering. When the unsupported source code fragment is too long, or when there are too many unsupported fragments, the parser may not be able to recover and

parsing of the complete file may fail. To prevent this from happening, different solutions are considered. For some of these solutions the source code of Spoofax needs to be modified in order to change the way programs are parsed by a Spoofax generated parser. This is an undesirable solution in this project since it requires a lot of time understanding and modifying the Spoofax source code. However, this solution will be briefly discussed in the next section. Other options like calling the preprocessor manually from within EpiSpin, or including only limited parts of the macro grammar are discussed afterwards.

3.1.1 Calling cpp in the background

The most obvious solution to remove macro calls from the source code is by calling the C preprocessor. This needs to happen before Spoofax parses the code. The basic idea is as follows: When the *Promela* code is entered in EpiSpin, the C preprocessor is called on the source code. The C preprocessor returns the processed source code, which is then free of macros. This processed code is then parsed and editor services are applied using the Spoofax generated parser and transformation rules. Two options need to be considered: the resulting editor services are mapped back on the original source code or the original source code is shown next to the processed code.

Let us first take a closer look at the first option, where the editor services - that are applied to the preprocessed code - are mapped back on the original source code. When the user writes a *Promela* model, the C preprocessor would be called in the background every time the user makes a change, just like the parser. The preprocessed source code is then saved in another file, ready to be parsed and checked by EpiSpin. That is where it gets complicated. Spoofax generated Eclipse plug-ins show editor services in the graphical user interface of Eclipse. This means that the preprocessed file should be opened for the editor services to be visible. But what is opted for in this case is not to show the editor services in the preprocessed file, but to map the editor services on the original source code. The positions of these editor services should be translated to the positions in the original files. To establish the difference between the old and the new positions, Spoofax needs to know which macro calls are expanded. However, Spoofax does not know anything about expanding macro calls, as observed before. Such a solution would therefore be impossible. Furthermore, such a feature would only be useful for developing languages that need to call an external preprocessor. Spoofax is primarily developed for creating new domain-specific languages which do not use a preprocessor.

The second option is to keep the original source code separate from the preprocessed code by showing the preprocessed code and its editor services in a separate window. This enables users to see both versions of the code simultaneously. The main disadvantage is still that models that contain macros may not be parsed immediately and error checks and other editor services are only available in the preprocessed versions of the source code, which means that the user gets its feedback on the model in a different window (within Eclipse) than from where the model is edited. In the worst case scenario, the original file only shows plain text with error markers indicating syntax errors that could not be recovered. Although this is not optimal, it allows the user to use macros without any restrictions.

3.1.2 Combining macro grammar with main grammar

As described above, the main problem when parsing *Promela* source code containing macro calls is that the positions where one is allowed to place macro calls need to be included in the grammar. Macro calls can occur in every possible position that can be parsed as a separate token, so it is not desirable to include all these positions. Adding optional macro calls around all the tokens in the grammar would increase the complexity of the EpiSpin source code tremendously and would introduce a lot of ambiguities as well. However, a lot of these macro call positions are rarely used [29]. Although full macro support is not possible using this solution, macro directives can be allowed at certain positions, covering the common positions for macro calls. These common positions are different for the different macro directives, which can be divided into three groups:

- `#include` for file inclusion,
- `#ifdef`, `#ifndef`, `#if`, `#else` and `#endif` for conditional compilation,
- `#define`, with or without parameters, used for defining macros.

The first two groups and the last group will be discussed separately.

#include and conditional compilation `#include` directives are normally used as separate statements, usually at the start of a file. Adding this directive to the *Promela* grammar would not lead to a parsing conflict since it is different from all other tokens. Therefore this directive is included in its full form and can be placed where a module (a code block in the global scope, such as a `proctype`) or statement can be placed. The same applies to conditional compilation directives. They are also treated as modules or statements.

According to [29], conditional compilation directives are mostly used in C

- as inclusion guards at the beginning and end of a file,
- to support the conditional compilation of sequences of statements,
- to support function signature alternatives.

After investigating some existing *Promela* models which make much use of macro directives (such as the work described in [35]), it turns out that the first two use cases hold for the *Promela* language as well. The third use case is only encountered once and is therefore not considered important enough to be implemented in EpiSpin. Therefore a conditional compilation directive can be used where a module or statement is used.

This fixes the positions where these macro directives can be placed, but the syntax of these directives still needs to be decided. For `#include` statements this is clear, since only a filename or a library may be used there, but in conditional macro constructs everything is allowed by the C preprocessor: it should only be syntactically correct after evaluation by the C preprocessor, but this is not checked by the C preprocessor. So any code that is not syntactically correct can still be included since the preprocessor only needs to decide whether the condition of the conditional macro holds and whether the possibly incorrect code should be

included in the source code before it is sent to the language parser. In EpiSpin, the complete macro is immediately parsed. This means that no incomplete source code can be included because the parser would not be able to parse it. It is therefore only allowed to include complete statements and modules in the conditional compilation directives. This makes sense since these directives are only allowed at positions where a new statement or module starts, so incomplete statements can never be combined with other incomplete statements to form a complete statement.

#define Including macro definitions using the `#define` directive is similar to including the `#include` directive and has the same limitations. But calling a macro definition introduces some additional complications. There are two different macro definitions: object-like macro definitions (like `#define SIZE 5`) and function-like macro definitions (like `#define SIZE(x) (5*x)`). Object-like macro calls cannot be distinguished from identifier references, which leads to reduce/reduce conflicts. The easiest solution would be to prohibit object-like macro calls in places where variable references are allowed, but this is undesirable since variable references can be used in statements through the whole source code.

Again, some solutions are possible here. The first one is to use heuristics to distinguish a macro call from an identifier reference. A heuristic that can be used here is the one that macro definitions are usually written in upper case. When an identifier is found that is written in upper case, it is parsed as a macro call. A big disadvantage of this method is that models (with or without macros) can no longer make use of fully capitalized identifiers anymore. Since this would impose a restriction on macro-free code as well, it is not implemented.

Another option to investigate is to partially implement the use of macro calls. Macro calls are often used to replace numerical constants. It is easy to allow macro calls in places where only constants are expected (when declaring or using an array variable for example). This would not introduce any ambiguity with the rest of the source code.

In addition to these partial solutions, a work-around will be implemented to allow all kinds of macro calls and other kinds of macro usage that is restricted by the EpiSpin grammar. This is done by allowing one particular conditional compilation directive not to be parsed by EpiSpin:

```
#define _EPISPIN_ 1
#if _EPISPIN_
/* any code */
#endif
```

This means that the *any code* part is not parsed, so literally any code can be typed over there. Because the condition of the if-statement always evaluates to true, this code is always included by SPIN. The downside of this work-around is that the user cannot profit from editor services and error checks in this part of the code. Although that is not desirable, it allows users to use unsupported macro constructs by 'switching off' parsing for a small part of the code, which is, in most cases, better than having a lot of syntax errors.

Function-like macro calls do not cause any ambiguity with the *Promela* language. Therefore these kind of macro calls can be used as statements or modules.

3.1.3 Conclusion

In this section, different approaches to solve the macro problems were discussed. Full macro support is not possible by calling the C preprocessor internally or by including the complete macro syntax in the *Promela* syntax. Therefore, only the following statements are allowed to be used:

- `#include` statements as freestanding modules or statements
- `#ifdef`, `#ifndef`, `#if`, `#else`, `#endif` statements as freestanding modules or statements,
- `#define` statements as freestanding modules or statements,
- Object-like macro calls are allowed at all places where a constant can be placed (like in array and channel definitions and other calls to arrays),
- Function-like macro calls are allowed as freestanding modules or statements,

A work-around to use *Promela* code in combination with macros, is to insert this code between `#if _EPISPIN_` and `#endif`. This code will be included for verification or simulation when the C preprocessor is called, but it will not be parsed by EpiSpin, restricting the user from profiting from any of the editor services. Finally, it is also possible to call the C preprocessor on the source code and open the preprocessed code in a new window. This preprocessed code will then be parsed and checked by EpiSpin.

Although full support is not possible, by using the above solutions it should be possible to deal with most of the macro code. Especially when you are writing a new program using EpiSpin, you can easily take those limitations into account and use the work-arounds. When an existing model is loaded, there is a chance that the program cannot be parsed immediately and should be modified to be parsed with EpiSpin. This can be done the easy way, by enclosing the unsupported macro constructs with `#if _EPISPIN_` and `#end`, or by rewriting the problematic constructs, for example by replacing macro calls by their definitions.

3.2 The Static Communication Analyzer

In *Promela* models different processes may need to communicate with each other to perform a task together or when different processes have different roles (one process can represent a server and another a client, for example). An important *Promela* construct for processes to communicate with each other is channels. The values of all the variables that are allowed in *Promela* can be sent as a message over such a channel. To determine which processes use a particular channel, one needs to scroll through the source code to find all instances of that particular channel. The Static Communication Analyzer (SCA) is a tool that finds

those relations automatically by traversing the abstract syntax tree of the source code. This tool is particularly useful when an existing *Promela* model needs to be analyzed.

Channels can be defined globally or locally and can be passed as a formal parameter when running another proctype. By scanning the source code of a *Promela* model, the channels and processes can be identified and their relations can be shown in a graph. Besides channels, global variables are interesting to show as well. They can be accessed and updated by all processes, which is why it is useful to show them in this graph. Using Stratego transformation rules, these *Promela* language constructs can be filtered and transformed into DOT [10] source code. The DOT language is chosen because of its simplicity and it provides the items necessary to draw a communication structure, such as different types of shapes, colors, lines to connect the shapes and labels to identify variable names.

Channels in *Promela* are complicated structures. They can be used in multiple ways and in multiple positions. A channel can have a buffer to store messages, but the buffer size can also be zero (a rendezvous channel). The simplest way a channel can be used is by declaring one and send (or receive) a message to or from it as a statement in a process. What is more difficult is to identify channels that are sent as a parameter in a process definition, or as a message in a channel operation. Basic ways to declare channels are as stand-alone local or global variables, but they can also be encapsulated in user-defined types and in arrays, making it hard to track them. In different sections these cases will be discussed, starting with the basic cases and going on to the more complex ones.

3.2.1 Basic cases

Let us start this subsection by identifying the *Promela* operations that are considered as a basic case. More difficult cases will be discussed later. The basic cases are:

- Definition of globally and locally defined channels with any buffer size and any message type except channels (these will be considered later)
- Exclusive read and write accesses from processes to channels
- Definition of other globally defined variables
- Channel send and channel receive operations of the above-mentioned channels
- Running a process, without passing a channel as a parameter

This communication analyzer only analyzes the source code in a static way. This means that the code is analyzed without evaluating variable references or conditions. So for example, even when a global variable is assigned a new value in a block of a selection structure that is not executed, there will be an arrow in the graph indicating that there is some interaction between that proctype and that global variable. The AST needs to be traversed once to derive the communication graph. When one of the above statements is encountered, the corresponding DOT code is created. The SCA only shows a relation between a variable or a channel and a proctype once, so when a variable or channel is accessed multiple times by a proctype, there will still only be one arrow.

The following notations will be used to draw the communication graph:

- Channels are represented by boxes, global variables by points and processes by circles. The `init` process will have another color than the other processes and global and local channels will be distinguishable by using rectangular boxes (local channels) and parallelograms (global channels),
- When a proctype reads a global variable, there is an arrow from this process to this variable,
- When a proctype writes to a global variable, there is an arrow from the variable to the process,
- When a proctype has exclusive rights to a channel, the connecting arrow is shown in bold print,
- When a proctype contains a channel send operation, there is an arrow from the proctype to the channel,
- When a proctype contains a channel receive operation, there is an arrow from that channel to the corresponding proctype,
- When a proctype is started by another proctype, this is indicated by dashed arrows.

3.2.2 Channels as formal parameters in proctype declarations

After these basic cases are completed, the communication analyzer can be extended to include more difficult cases. These will be discussed in the next subsections and will include:

- Channels as formal parameters in proctype declarations,
- Channels in channel message fields,
- Encapsulated channel definitions.

When SPIN is analyzing a model, every different channel is given a unique number in order to track down the channel. Consider the following *Promela* code:

```
init{
  chan ch = [0] of {int}; // Declaration of a rendezvous channel
  run procA(ch);
  ch ! 42;  //sends the value 42 to channel ch
}

proctype procA(chan c){
 c ? 42; //read the value 42 from channel ch
}
```

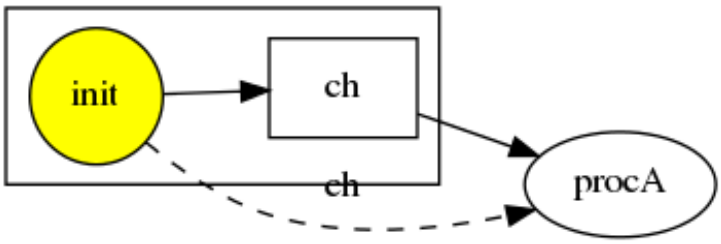

Figure 3.1: The communication graph of the example

In this example, channel `ch` will get the id 1. When `procA` is run, the value 1 is given as a parameter. Together with the type information in the process declaration, variable `c` is known to be a channel with an id of 1, which is the same channel as the channel with identifier `ch`. So although there are two channel identifiers, there is only one channel declared in the model. In an ideal case, the communication graph only shows one channel as well, resembling the model as closely as possible. However, this would mean that the channel variables need to be tracked or evaluated to see whether the two channel definitions point to the same channel. Although tracking variables was not the intention of this static tool, for this situation it can be easily done since the positions where variables are passed are easily recognized (they are always passed in a `run` statement and are always assigned to a new variable name in the process declaration). This contrasts with - for example - channels that are sent as a message to another channel, which can be received - and thus assigned to a new variable - at many different places.

Now consider the example again. There is an init process, with a locally defined channel, and it starts another process using that channel as a parameter. The proctype declaration then assigns that channel to the variable `c` and a send and receive operation is done on the channel. Figure 3.1 shows the communication graph. Besides the already-mentioned elements, two additional components can be found: a clustering box and a label. The clustering box clusters a process with all of the channel declarations done in that process. The label of a dashed arrow shows the names of the parameters that are sent to instantiate a new process.

Note that assignments to (channel) variables are not shown in the graph. This means that if a second channel is defined and is then set to point to the first channel, no code will be generated. So when a communication graph of the following example is generated, channel `ch2` is represented as a second channel alongside channel `ch`.

```
init{
  chan ch = [0] of {int};
  chan ch2 = ch;
  ch2 ! 42;
}
```

When a message is sent via channel `ch2`, it is actually sent via the same channel that is assigned to channel `ch`, but this will not be evident from the graph. To implement this, it would be necessary to track down every variable to see when it has been updated and

according to what conditions, which is beyond the purpose of this tool. In practice, it happens rarely that a channel variable is updated, since it would leave another channel unreachable and there is no use pointing to the same channel many times within the same process.

3.2.3 Channels in message fields

In this subsection another way of passing channels between proctypes is discussed. When a message is sent over a channel, the message itself can be a channel identifier (in that case the channel id is sent). When a channel id is sent to another channel as a message, it is never clear where the message is received because the next statement that is executed is decided non-deterministically. Let us clarify this again by giving an example:

```
1  chan globchan = [2] of {chan};
2
3  active proctype proc1(){
4    chan ch = [2] of {int};
5
6    globchan ! ch;
7
8    if
9      :: globchan ? ch;
10     :: ch ! 42;
11   fi;
12
13 }
14
15 active proctype proc2(){
16   chan ch1 = [2] of {int};
17
18   if
19     :: globchan ? ch1;
20     :: skip;
21   fi;
22
23   ch1 ! 23;
}
```

There is one global channel and there are two processes. The first process sends a local channel over the global channel. It is impossible to predict where this channel will be received: right after it is sent in the first process (at line 9) or in the second process (at line 19). As a consequence, it is also not known through which channel the number 23 is sent in the second process: the channel as initialized at line 16 in the first case, or the channel that is assigned to `ch1` at line 19 in the second case. There are two local channels in the graph for which the interaction between the global channels and the local channels

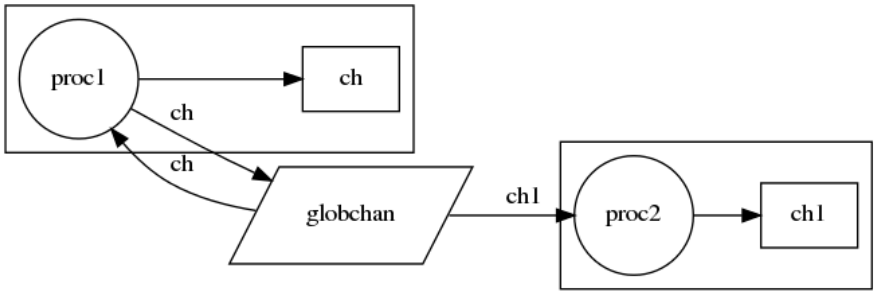

Figure 3.2: The communication graph of the example with channels sent as message parameters

is clarified using labels. These labels show the variable names that are sent or the variable names that are used to store a message from a channel. In Figure 3.2 the communication graph corresponding to the above example is shown. It can be seen that a channel `ch1` is declared in `proc2`, but that there is also a channel coming from `globalchan` that is bound to `ch1`. As in the source code, it is not clear when this variable is bound, and thus whether `ch1` points to the same channel as `ch` or not.

3.2.4 Encapsulated channel definitions

In the last two sections it was shown how channels were passed along in a *Promela* model. These channels were declared as stand-alone variables. It gets more complicated when a channel is used as part of a user-defined type, or in an array of channels. Channels from arrays may be used the same way as stand-alone channels, since a variable reference `channels[1]` points to a channel the same way a single identifier does. Still, it is hard to show all channels from an array separately, because it is not always possible to resolve the index number of the array. If only ordinary numbers were used, it would be easy, but array indices can be built using complicated expressions, making it impossible to statically determine the index number. Therefore it is decided not to show all channels separately, but only the array variable. As a result of this decision, the graph indicates communication between a process and an array of channels, instead of a channel itself. Although it is not optimal, it still gives an idea of the processes that are using the array.

Another special case of channel declaration is the use of channels from within user defined types. Like in *C*, users can define a new type which consists of elements of other types. When such a user-defined type is defined multiple times, all channels from the different definitions should be shown separately. In this case, that is possible, because the channel name is just preceded by the name of the user-defined variable reference. Consider the following code fragment:

```
typedef Node{
  chan queue= [0] of {int};
}

active proctype proc1(){
  Node n;
  n.queue!2;
}
```

When the channel of Node `n` is now used to send a message, the communication graph will show an arrow from the process `proc1` to the channel `n.queue`.

So far, only the basic operations of encapsulated channels are discussed. But these user-defined types with channels can also be sent as a parameter to another process or as a message via another channel. Arrays are not allowed to be sent as a parameter or in a message field due to restrictions in the *Promela* grammar. When a user-defined type that includes a channel is sent as a parameter to another process, it should behave the same as when a stand-alone channel is sent. Therefore every user-defined type should be checked for including channel definitions. When there is such a channel, all communication with these channels in the newly defined process should point to the local channel as defined by the first process.

3.2.5 Conclusion

The Static Communication Analyzer will show the following information considering the use of channels and global variables in processes. It will

- show the relations between channels and processes,
- show the relations between global variables and processes,
- be able to track a channel that is given as a parameter to another process and recognize this as a earlier defined channel,
- be able to show locally defined channels and indicate channel names that are send as a message to another channel,
- give the relations between processes and channel arrays (not with their elements)
- show the relations between channels in user-defined types and processes.

3.3 Performing verification and simulation runs

Since EpiSpin is an Eclipse plug-in, the general layout of the editor is fixed by Eclipse. The appearance of the editor services are a little bit more customizable, but there are certain conventions which are closely followed. For example, the way errors are indicated (by a red

mark in the side line and a red line underlining the error) could be changed, but every Eclipse user is confident with this way of providing feedback so that is why these conventions are followed in EpiSpin.

To call the SPIN verifier or simulator a series of commands need to be executed. For verification first C code needs to be generated. Then this C code of the verifier needs to be compiled and finally it should be executed. There are several flags that can be specified with each of these commands to customize the verification and simulation runs. The program iSpin [14] is a basic *Promela* editor that has a verification and simulation panel. In these panels it is possible to select some options. When a verification or simulation run is started, iSpin derives the correct commands according to the options selected and runs them.

The strongest point of iSpin is the way output is represented. For verification runs, the output is just textual, but the output of a simulation run consists of four different frames that are shown in iSpin. It has a main textual frame describing the steps that are taken and a message sequence chart showing the created processes and their usage of channels. Furthermore there are two frames showing the input of the channels and the values of global variables at the selected step. These frames are connected with each other and when a step is selected (in the textual output or in the message sequence chart) the other frames are updated. iSpin is written in the Tcl [28] programming language and uses the Tk graphical toolkit for its user interface. Tcl/Tk provides the tools to connect all frames with each other and step through the source code interactively. When the same simulation is performed using the command line, all steps are described sequentially, but detailed information is only printed for the last step. A static version of the message sequence chart can be generated by specifying the `-M` flag.

In the optimal case, all popular features from iSpin will also be implemented in EpiSpin. But since EpiSpin is a plug-in for Eclipse, it is restricted to the tools that are supported by Eclipse. The standard way to display output in Eclipse is by sending text to the Console window. Additional (output) windows should be handled in the code itself. Spoofax is intended for developing transformation languages, i.e. languages that transform the source code into other code. These transformations are done by Spoofax using Stratego rules. To perform a verification or simulation run in EpiSpin, it is necessary to call SPIN, collect the output and display it in EpiSpin. In Spoofax there is no support for interacting with an external program directly. However, Spoofax provides two possibilities to use your own defined Java programs. The first one is by using the standard Eclipse extension points (which will be used to implement the user interface). The other possibility is to write new strategies in java and import those in the Stratego code. Using java it is possible to call external programs like SPIN.

In EpiSpin, the option screens for verification and simulation look the same as in iSpin. This layout of these iSpin panels is copied as much as possible because most users will be familiar with this layout. The option screens are made using the Eclipse Standard Widget Toolkit (SWT). SWT integrates well in Eclipse and accesses the user interface facilities of the operating system and gives a native appearance to the option screens. The option screens can be implemented in two different ways: as a separate window that is independent of any *Promela* model, or as a file in the Eclipse project, which can also store data and is linked with the *Promela* file.

The first implementation is most similar to how it is done in iSpin and in the existing Eclipse plug-in [23]. This implementation requires building a new SWT window where the verification and simulation options can be specified. In this window a verification or simulation can be started on the currently opened *Promela* model.

The second implementation option is very different. It requires a data file which extension is associated with a certain user interface within Eclipse. This means that when such a file is opened in Eclipse, it shows the user interface that is associated with that file extension. This user interface could consist of verification and simulation options. The content of the file could be used to save data. In this case, it would be possible to store the selected options in XML or plain text in the file. When a user reopens that file, the options are loaded from the file and automatically selected in the user interface. In Eclipse it is possible to split the screen, so both the *Promela* code and the option screen can be viewed at the same time. A disadvantage of this method would be that every *Promela* file needs its own option screen, even when you only want to do a single verification or simulation. Especially when a large batch of existing *Promela* files is imported, this can take a lot of time.

Despite this downside, this last method is being chosen as the one that will be implemented. EpiSpin is primarily designed to create and edit new *Promela* models and therefore it will not happen often that batches of existing programs will be imported in EpiSpin. Furthermore, the possibility to save the choices in the option screens would be a useful addition compared to current *Promela* editors.

When the choices in the option screens are selected, a verification or simulation run can be started by pressing the 'Run' button. A Java event handler then generates a SPIN command by checking the choices and setting the appropriate flags accordingly. This command is then executed from the Java program and the textual output is sent to the Eclipse console window. iSpin displays the simulation results in different frames and allows the user to step through the different steps and inspect the buffer and data values for every moment. Such a feature will not be implemented in EpiSpin because it would be too complicated. SPIN only returns text when it is called using the command line. This text contains a list of all the statements that are executed and some detailed information about the last step. It would be possible to parse this text and make the filename references clickable, but it would not be possible to see detailed information about every step, since this is not returned by SPIN. For the same reason it is not possible to rewind to the first step and execute the program step by step again (as iSpin can): there is no information being returned about these steps.

A special way of running a simulation is by doing it interactively. In a standard simulation SPIN chooses non-deterministically which statement is executed. When an interactive simulation is performed, the user can select the next statement that needs to be executed. This feature is also implemented in iSpin where a window pops up with the possible choices and a similar system will be used to implement this in EpiSpin.

3.4 Error messages

This section elaborates on the design choices related to the implementation of error messages in EpiSpin. The error messages in EpiSpin are based on the error messages that are

found in the source code of SPIN. It is not possible to copy all error messages one on one because of several reasons that will become clear in this section. The traditional way of displaying errors of *Promela* models is by calling the SPIN syntax checker, either using the command line or the syntax check button in *iSpin* or *Xspin*. The SPIN executable is then called and the source code is parsed and checked for errors. A syntactic error is found when the source code can not be parsed according to the SPIN grammar. Semantic errors are found by checking conditions on the parsed code and if a condition fails, the corresponding error message is printed by calling the `fatal()` or `non-fatal()` function, using the error message as a parameter. The first function is called when an error is found that requires the SPIN program to abort immediately and the `non-fatal()` function is called when parsing and error checking may be continued. Syntactic errors are also printed using these functions. The error message is then 'syntax error' followed by the wrong or unexpected token.

In EpiSpin these checks are performed using different methods. Syntax errors are directly returned by the Spoofax generated parser and semantic error checking is performed by analyzing the abstract syntax tree. As long as the parser can recover from the syntax errors, parsing and error checking is continued, reporting all errors that are made. Getting instant feedback increases the coding speed.

All errors that are found in the SPIN source code are compile-time errors and therefore the errors that are shown in EpiSpin are also compile-time errors. To decide which semantic error messages will be implemented in EpiSpin, all calls to `fatal()` and `non-fatal()` are collected and the error messages are analyzed. Approximately 200 error calls are found, which can be roughly divided in two categories: language dependent and language independent errors.

Language independent errors are errors that can occur in most programming languages. These include:

- using undeclared identifiers such as labels, variables and functions;
- using a declared identifier in a wrong way, like jumping to a variable name, omitting a required array index or using a function name in an expression;
- declaring identifiers multiple times in the same scope (while this is not supported);
- calling functions with the wrong amount of parameters;
- using fields of objects or user-defined structures that are not declared;
- writing to read-only variables an or reading from write-only variables.

All errors from the SPIN source code that relate to one of these errors are implemented in EpiSpin. These situations cover 50 errors. Spoofax provides good mechanisms to implement these kind of errors efficiently using name analysis and dynamic rules (see Section 4.3). Together with syntax errors, these are the errors that are most easily made, usually by making a typing error or forgetting an identifier name. Therefore a lot of effort is put in including all of these errors in EpiSpin.

All other error messages are *Promela* dependent errors that are fired when *Promela* specific structures are used wrongly. 30 of these errors are errors that impose additional

constraints on the syntax. These errors could also have been handled by the parser, but it would have made the grammar grow enormously. Such errors are duplicate `else` statements in a repetition or conditional structure, using the `break` keyword outside a repetition structure or using more than one `run` statement in an expression. These are errors that are especially made by starting *Promela* developers. Getting direct feedback on these errors will contribute to a faster understanding of the *Promela* language and are therefore all implemented.

The remaining 120 error messages cover cases that are very specific for certain language constructs. Especially when using the `d_step` structure a lot of errors can be made (approximately 20 semantic errors). Various constructions are prohibited in this structure and so is jumping into and out of a `d_step` block. From these specific errors, approximately 70 errors are only encountered very rarely because the structure is not often used in *Promela* or because multiple conditions should fail together. A lot of these rare situations relate to memory issues. Examples of these kind of errors are too many `mtype` variables or too many statements in a `d_step` sequence. These kind of errors are not implemented in EpiSpin yet. They would occur only in very limited cases and are due to time restrictions assigned for future work. The other 50 very specific error messages are implemented because of the frequency of use of the involved construction. A channel operation is used more often than an unless statement, for example. This frequency is determined by analyzing 300 *Promela* models of 50 different problems that are found in the BEnchmarks for Explicit Model checkers (BEEM) database [31]. A list of error messages that are implemented in EpiSpin can be found in Appendix E.

Chapter 4

Implementation

Creating a new plug-in using Spoofax starts with creating a new Spoofax project in Eclipse. A new Spoofax project contains a small language with editor services that illustrates various Spoofax features. From there on, files can be added, deleted and customized. Three different languages are used to program in a Spoofax project: the SDF language for syntax definition [11], the Stratego language to describe analysis and transformation [5] and the Editor Descriptor Language which is a domain-specific language built for Spoofax to specify which transformations should be applied to define the editor services [22].

Figure 4.1 is a copy of Figure 2.2, but it is modified to show which IDE components that Spoofax provides are used in EpiSpin. All components from block A, B and C are included in EpiSpin. The editor services reference resolving, content completion and error marking which can be found in block D are also implemented. Code generation is used to generate DOT-code for the Static Communication Analyzer. Solid arrows indicate components that are derived from other components and they can be customized afterwards. The grammar definition and all components from the semantics block are completely written by the developer. When a component has an incoming dashed arrow it means that it uses another component. Hover help, occurrence highlighting and views are not implemented in EpiSpin. Hover help provides a pop-up with information about the code that is hovered

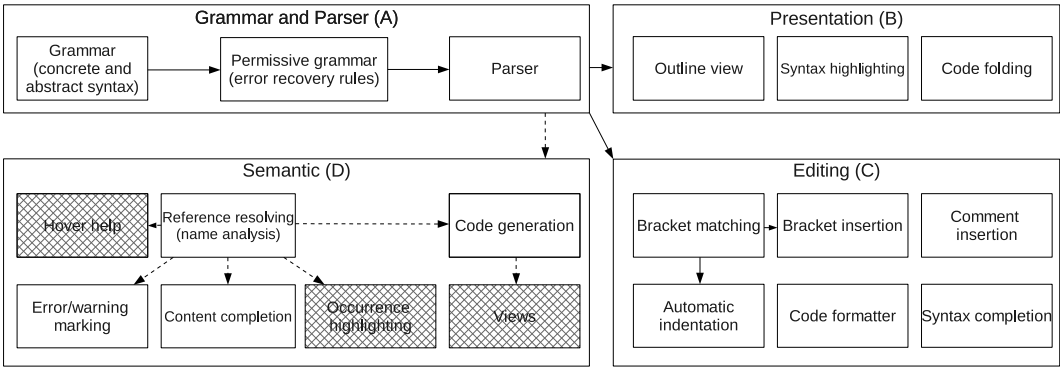

Figure 4.1: Relations between the IDE components. Based on [22].

over. The *Promela* website [13] has a clear explanation of all language constructs so it is not necessary to provide extra information. Occurrence highlighting is not yet implemented in Spoofax, and can therefore not be implemented in EpiSpin yet. Views are used to show transformed code, which is not done by EpiSpin. Although the source code can be transformed into DOT code for the Static Communication Analyzer, this code will not be shown in a view. The DOT code itself is not useful for EpiSpin users and is only used as an intermediate step to create the graph.

This chapter covers the implementation of these components. First the grammar files will be discussed in the next section, followed by a section about analysis and a section covering the editor services. Appendix F enumerates the files that are necessary to build the EpiSpin project and explains how EpiSpin could be built using the source code.

4.1 Grammar

As any other language, the *Promela* language is constantly evolving. Since the introduction of the *Promela* language and the publication of the book *The Spin Model Checker* [15] in 2003, various changes are made to the language. For EpiSpin, the *Promela* grammar of version 6 [13] is used as a basis, but this grammar definition is not complete. Therefore *The Spin Model Checker* is also used to get more information on some of the language constructs and for some problems even the source code of SPIN (version 6.0.1) is explored.

A syntax definition in SDF can be broken up in two main parts: the definition of the lexical syntax and the definition of the context-free syntax. The lexical syntax is defined in the file *Common.sdf* which is generated when a new Spoofax project is generated. It already contains definitions for identifiers, constants, strings, comments and layout and only needs small modifications to match the *Promela* grammar. When a production rule is matched as `LAYOUT`, this rule is neglected by the parser and will not be visible in the abstract syntax tree. Examples of constructs that will be treated as layout are comments, the newline character, spaces and tabs.

The context-free syntax is defined in the file *Promela.sdf* and includes all production rules. A rule is an enumeration of sorts (non-terminals) and lexicals on the left-hand side which are matched against the sort on the right-hand side. A rule is also identified with a constructor name, which will appear in the AST when that rule is matched.

In the next sections different aspects of the grammar will be discussed. First refactorings of the original *Promela* grammar are discussed. Then it is explained how separators are implemented in the EpiSpin grammar, followed by a discussion on ambiguities. The final topic of this section is about implementing macro definitions.

4.1.1 Grammar rewriting

To create the EpiSpin grammar, some changes are made to the *Promela* grammar as used by SPIN. Most of these changes are small: some non-terminal symbols can be replaced by their alternatives when the non-terminal is only used once. This way, both the amount of production rules and the depth of the AST are reduced. An example of this refactoring can be found in Figure 4.2 (before) and 4.3 (after). In both figures, the upper left part shows

```
start      : [ module ] +
module     : proctype
           | init | ...
proctype   : PROCTYPE name '(' [ decl_lst ] ')' '{' sequence '}'
init       : INIT [ priority ] '{' sequence '}'
```

```
Module +                                          -> Start{cons("Modules")}
Proctype                                          -> Module{cons("ProctypeModule")}
Init                                              -> Module{cons("InitModule")}
"proctype" ID '(' Decl-lst? ')' '{' Sequence '}'  -> Proctype{cons("Proctype")}
"init" '{' Sequence '}'                           -> Init{cons("Init")}
```

```
Modules(
  [ ProctypeModule(
      Proctype(
      , "foo"
      , Steps(
          Stmnt(
            FullExpr(
              AnyExpr(
                Const("skip")
            )),
[Sep()])))])])
```

Figure 4.2: A fragment of the *Promela* grammar used by SPIN and the AST that is built with this grammar to construct the abstract representation of the smallest *Promela* model that exists: an init declaration with only a skip statement.

```
start      : [ module ] +
module     : PROCTYPE name '(' [ decl_lst ] ')' '{' sequence '}'
           | INIT [ priority ] '{' sequence '}' | ...
```

```
Module +                                          -> Start{cons("Modules")}
"proctype" ID '(' Decl-lst? ')' '{' Sequence '}'  -> Module{cons("Proctype")}
"init" '{' Sequence '}'                           -> Module{cons("Init")}
```

```
Modules(
  [ Proctype(
    , "foo"
    , Steps(
        Stmnt(
          FullExpr(
            AnyExpr(
              Const("skip")
          )),
[Sep()])))])
```

Figure 4.3: A fragment of the EpiSpin grammar after modifying the *Promela* grammar and the AST that is built for an init declaration with only a skip statement.

a small fragment of the *Promela* grammar using the Yacc format [18] where non-terminals are shown in lowercase and terminals are shown in uppercase. The bottom left part shows the same grammar using the SDF notation where non-terminals start with a capital and terminals are listed between quotes. Also every rule is annotated with a constructor, which is used on the right-hand side to build the AST of a small *Promela* program with only a `proctype` declaration. It can be seen that Figure 4.2 has more production rules and one extra constructor in the AST compared to Figure 4.3. Although the effect for end-users is hardly noticeable, it has some advantages for the development of EpiSpin. With less production rules the analysis strategies are less complicated, which leads to compacter code. Several of these small modifications are made in the *Promela* grammar, but there is one major refactoring that will be discussed separately.

Linear Temporal Logic (LTL) claims specify conditions about states of the program that should always / never / eventually happen. These claims are written as modules in the global scope using normal expressions combined with special temporal logic symbols which are prohibited or have a different meaning outside LTL claims. The *Promela* grammar as shown on the SPIN website [13] does not include inline LTL claims, while they are supported since SPIN version 6. Therefore the source code is consulted to see how SPIN handles inline LTL statements.

The most important observation is that the SPIN source files - in this case the grammar file *spin.y* which is used by Yacc to generate a parser and the file spinlex.c which is used by Lex [25] to generate a lexical analyzer - use global variables to keep track whether the parser is in an LTL module or not. So when the beginning of an LTL module is parsed (indicated by the `ltl` keyword), a global variable is set. When the next token is requested from the lexical analyzer, this variable is checked. If it is set, identifiers such as *always*, *next* and *until* - which have a special meaning in LTL claims - are returned by the lexical analyzer as keywords instead of identifiers. Besides identifiers, symbols can have a special meaning within an LTL claim as well, such as the `->` symbol which defines an implication in LTL claims, but is used as a separator in sequences as well. Some symbols, like the `<>` symbol, are forbidden outside an LTL claim and are only matched correctly by the lexical analyzer inside an LTL claim. This way, characters can be parsed differently inside or outside an LTL statement.

Spoofax generates a parser based on the SDF syntax definition. This parser is a Scannerless Generalized-LR (SGLR) parser [36]. Scannerless parsing means that there is no scanner to divide a string into lexical tokens. Instead, lexical analysis is integrated in the context-free analysis of the entire string. This means that when a token is parsed, the context is taken into account. For example, consider the `->` symbol again. The grammar for both LTL expressions and normal expressions can contain this token without introducing conflicts because the meaning is determined during the context-free analysis. So instead of using global variables for communication between the scanner and the parser, this is now handled by the context-free analysis immediately. Therefore part of the grammar for normal expressions is duplicated and used to define the grammar of LTL expressions to get a clear distinction between both kind of expressions. The corresponding rules from the EpiSpin grammar are:

```
"ltl" ID? "{" LTLFullExpr Sep? "}"  -> Module{cons("Ltl")}
LTLExpr                             -> LTLFullExpr {cons("Expr")}
LTLAnyExpr                          -> LTLFullExpr {cons("AnyExpr")}
```

`LTLExpr` and `LTLAnyExpr` are the same as `Expr` and `AnyExpr`, but `LTLAnyExpr` is expanded with LTL specific syntax. The complete syntax can be found in Appendix D.

4.1.2 Statement Separators

The *Promela* language has two separators: `;` and `->` which can be used interchangeably. The general rule is that separators *should* always be used between two steps and *may* also be used after the final step. A step can be a statement or a declaration in a local scope. Modules and declarations in the global scope do not have to be separated. The SPIN syntax checker is not as strict as the grammar suggests and allows some exceptions to these rules and is even somewhat unpredictable in some cases.

Two of these exceptions are allowed deliberately. The SPIN developers found out that two syntax errors were made very often. The first one is about steps that end with the } character. There are about six of these steps and most of them are used to end a special sequence, for example an atomic sequence. In the scope of a proctype such a 'hidden'

sequence is considered as one step, and thus it needs a separator when it is followed by another step. But *Promela* developers are not used to write separators after the } sign, because this character is mostly used to end a big global structure like a `proctype` or a `never` declaration after which a separator is not necessary. Because of this they tend to omit the separator after a statement as well, although it is mandatory at that point. Examples of structures after which a } symbol is expected according to the grammar are channel declarations (`chan ch = [1] of {int}`) and atomic blocks (`atomic{...}`). This made SPIN developers decide to allow forgotten separators after such structures to help *Promela* developers.

The other case when a missing separator is allowed, is after the `else` keyword. The `else` keyword is normally used in selection structures as a guard (see Section 2.3) and looks like:

```
if
:: n>2 -> ...
:: else -> ...;
fi
```

As in the first case, the separator after the `else` keyword (in this case the `->` symbol) was often forgotten by *Promela* developers, because it is uncommon to write a separator after the `else` keyword in other languages. Therefore it is tolerated to omit a separator here as well. For both cases a few lines of C-code are added to the parser, checking if the last token is the `else` keyword or the } sign. In these cases a semicolon is added by the lexical analyzer and the code is accepted. In the case of EpiSpin this is not possible since the parser is directly derived from the grammar. Therefore this tolerant behavior of the SPIN parser should be handled in the EpiSpin grammar.

Another issue regarding separators is that the SPIN parser is somewhat unpredictable when it comes to parsing multiple separators sequentially. When a separator is actually used to separate two statements, there is no restriction by the parser on the amount of separators that can be written. But when a separator is used to terminate the last statement of a sequence (and therefore is not mandatory), only two or three separators are allowed. After the last statement in a selection or repetition structure more than two separators are prohibited, but after the last statement in a `proctype` at the most three separators are allowed. Since no structure was found in this, it is decided to allow an unlimited amount of separators at all positions, being less strict than the SPIN parser. Since there is no semantic meaning for modelers to use more than one separator, it is hardly ever used in *Promela* models. Therefore, it is not expected that this difference in strictness would lead to any problems.

The issues mentioned above are implemented in the EpiSpin grammar by expanding the original grammar at some points. The original grammar has the following rules regarding steps that form a sequence:

```
sequence : step [ ';' step ] *

step     : stmnt [ UNLESS stmnt ]
           | decl_lst

stmnt    : ATOMIC '{' sequence '}'
         | GOTO name
```

In the EpiSpin grammar two additional distinctions are made: A step can be a normal *step* or it can be the final step of a sequence: *stepfs*. *stepfs* does not need to have a separator and for *step* it depends on whether it ends with the } character. Because of this difference, the need for a separator can not be determined in the *sequence* rule, but should be determined in the rules for *step* and *stepfs*. Only a *step* that does not end with a } character should be followed by a separator. Therefore a second distinction is made: statements ending with a } (called *stmnt*) and statements not ending with that character (*stmntz*). The rules for a sequence and its steps in EpiSpin are then:

```
sequence : [ step ]* stepfs

step     : stmnt sep+
         | stmntz sep*
         | stmnt UNLESS stmnt sep+
         | stmntz UNLESS stmnt sep+
         | stmnt UNLESS stmntz sep*
         | stmntz UNLESS stmntz sep*
         | one-decl sep+

stepfs   : stmnt sep*
         | stmntz sep*
         | stmnt UNLESS stmnt sep*
         | stmntz UNLESS stmnt sep*
         | stmnt UNLESS stmntz sep*
         | stmntz UNLESS stmntz sep*
         | one-decl sep*

stmnt    : GOTO name

stmntz   : ATOMIC '{' sequence '}'
```

The first thing that is noticed is that the EpiSpin grammar is much longer. The rule `step : stmnt [ UNLESS ] stmnt` is split in six lines now, since all combinations of *stmnt* and *stmntz* with `unless` should be written down explicitly. It would be possible to get the first two alternatives of the *step* rule by introducing optional sorts in the third and sixth alternative, but it is chosen to split those in different rules so statements with and without `unless` get different constructors. It could be seen that steps that end with a statement that

should be followed by a separator end with a non-empty list of separators. Steps ending with the } character are followed by a possibly empty list of separators, just like all final steps of a sequence (*stepfs*).

Another difference between the original grammar and the EpiSpin grammar involves the declaration list. This modification is made to avoid ambiguities and will be discussed in the next section.

4.1.3 Ambiguities

Another motive for modifying the original SPIN grammar is to avoid ambiguities. Ambiguity arises when multiple production rules of the syntax definition can be matched against a certain input string. Using an LALR(1) parser, this leads to shift/reduce and reduce/reduce conflicts in the parser which should be solved first. With Generalized-LR (GLR) parsers, like the EpiSpin parser, a parse forest is created. This is a compact representation of all possible parse trees for an ambiguous input string [36]. There is no lookahead with GLR parsers, but all possible parse trees will be added to the parse forest. Some of these branches then fail when more tokens are parsed. But when there is an ambiguity in the grammar, multiple accepting parse trees will be available in the parse forest. In the abstract syntax tree the ambiguous code fragment is represented as a list in an `amb()` constructor, where every list element is a partial tree from the parse forest. To resolve this ambiguities the parser should be instructed which production rule should be matched. Sometimes it is possible to rewrite the grammar so the ambiguity is removed completely.

An example where the grammar is rewritten to avoid ambiguities is by modifying the rule *decl_list*. Consider the following fragment of the SPIN grammar:

```
sequence: step [ ';' step ] *
step    : decl_lst
decl_lst: one_decl [ ';' one_decl ] *
```

When the source code is parsed and it contains two declarations in a row (both are a *one_decl*), there are two ways this can be parsed: it can be parsed as one step, containing a *decl_lst* with two *one_decl*s, or it can be parsed as two steps, each containing a *decl_lst* with one *one_decl*. This ambiguity can easily be avoided by removing the list from the *decl_lst* rule. In fact, that complete rule can be removed when its right-hand side is directly placed in the *step* rule (as described in beginning of this section):

```
sequence: step [ ';' step ] *
step    : one_decl
```

Other situations require explicit instructions for the parser to avoid ambiguities. The SDF language provides several disambiguation constructs to help the parser decide which of the production rules should be matched (and in which order) [11]. All of them create a filter that removes the subtrees that are identified by the disambiguation construct from the parse forest. So disambiguating in SDF is done by removing the branches that are wrong, instead of choosing the branches that are correct.

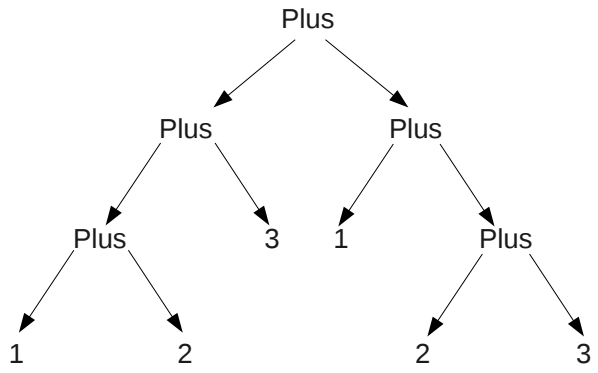

Figure 4.4: The parse forest corresponding to `1+2+3`

Disambiguation constructs are mostly used for disambiguating expressions. As in most languages, it is possible to recursively combine several operators with each other in expressions in *Promela*. All these operators have different priorities and associativity that should be taken into account. In SDF it is possible to specify the associativity by annotating production rules with the {`left`}, {`right`} or {`non-assoc`} attribute. Based on this attribute, a filter is created that removes part of the parse forest. Consider an addition operation `1+2+3` and the syntax rule `AnyExpr "+" AnyExpr -> AnyExpr {cons(Plus),left}`. The corresponding parse forest can be found in Figure 4.4. There are two different subtrees that represent a parse for `1+2+3`. Since the `left` attribute is specified, the left subtree is the correct one in this case. The `left` attribute makes sure that all subtrees of `Plus` for which the right-most subtree is again `Plus` will be removed from the parse forest.

Another way of resolving ambiguities is by specifying priority rules:

```
context-free priorities
AnyExpr "*" AnyExpr -> AnyExpr {cons(Times)}
    >
AnyExpr "+" AnyExpr -> AnyExpr {cons(Plus)}
```

These rules state that the `*` operator has a higher priority than the `+` operator. When an expression `1+2*3` is parsed, a parse forest is created with `Times` as a child of `Plus` and the other way around. With the priority rules specified, all `Plus` constructors in the parse forest that are direct children of `Times` constructors will be filtered.

Furthermore it is possible to annotate a rule with the {`reject`} attribute to specify that a certain left-hand side may not be matched to the right-hand side. This is often used in the lexical syntax definition to restrict keywords to be parsed as normal identifiers: `proctype -> ID` {`reject`}. Two other disambiguation constructs are mentioned in Subsection 4.1.4.

4.1.4 Macros

As described in Section 3.1 it is not possible to implement full support for macro calls. Implementing macro calls requires the use of the C-preprocessor to substitute the macro calls with their definitions. This transformation can not be done in EpiSpin since it is not possible to include the syntax for macro calls in the grammar. Therefore macro definitions can only be used as modules and statements and macro calls can only be used at the position where a variable reference or a constant is expected.

Implementing the macro syntax in the EpiSpin grammar was the most complicated part of the grammar definition. Macro definitions are the only language construct for which the newline character has a special meaning. It is used as a terminator after a macro definition and as a separator in the conditional compilation structure `#if` *cond* ... `#endif`. In EpiSpin the newline character is defined as a layout character, which means that the parser ignores this character when it is parsing the source code. For macro definitions the newline character is part of the production rules and should therefore not be ignored. The newline character is also important when multi-line macro definitions are considered: when a newline character is preceded by a backslash character, the macro definition continues on the next line.

To resolve ambiguities regarding newline characters another disambiguation construct is used: follow restrictions. A follow restriction `ID -/- [a-zA-Z0-9_]` states that an identifier `ID` can not be followed by a letter, number or underscore. This means that an input string with only these characters can never be parsed as two separate identifiers.

The SDF rules for a macro definition (in the global scope) are:

```
"#define" ID ANY? (ENDMAC ANY)* ->Module{cons("MacroDef"),avoid}
"#define" IDFUN "(" {ID ","}* ")" ANY (ENDMAC ANY)*
                                ->Module{cons("MacroFunDef"),prefer}
```

For these rules, three new lexicals are defined:

- `ANY`, which is a list of any characters except the backslash and newline character. This restriction is specified using a follow restriction that states that `ANY` may not be followed by anything but a backslash or newline character. This means that every other character that is typed on that line is part of the macro definition.
- `ENDMAC` is the backslash character.
- `IDFUN` is an identifier name, just like `ID`, but it has different follow restrictions. `IDFUN` can only be followed by a `(` character.

Furthermore, both rules have a special attribute: {`avoid`} and {`prefer`}. This means that for every macro definition it is first tried to match it as a function-like macro definition. If this fails, the standard macro definition is tried. The function-like macro definition has a restriction stating that the first character after the identifier must be an opening parenthesis. This restriction is necessary to correctly differentiate between `#define(foo)` and `#define (foo)`. The first example should be parsed as a function-like macro and the second example should be parsed as a normal macro definition. If this restriction would not have been

```
1 editor-analyze:
2   (ast, path, project-path) -> (ast2, errors, warnings, [])
3   with
4    !path;
5    editor-init;
6    ast2 := <analyze> ast;
7    errors   := <collect-all(constraint-error, conc)> ast2;
8    warnings := <collect-all(constraint-warning, conc)> ast2

9 analyze = desugar-all; rename-all
```

Figure 4.5: The observer rule 'editor-analyze'

applied, both examples would be parsed as a function-like macro, since the {prefer} attribute forces the parser to check this rule first and both examples could be matched against that rule. The conditional macro definitions are built using similar constructs.

4.2 Analysis

In this section the analysis of the abstract syntax tree (AST), which is returned by the parser, is discussed. Before this AST is used to define the editor services, it is analyzed and modified first. During this analysis phase the AST is annotated with semantic information which is used by the semantic editor services. But first the AST is modified by rewriting certain language constructs to simplify the abstract representation. This is called *desugaring*. This analysis of the AST is performed every time a new AST is created, so every time a small change is made in the *Promela* model. This is done by defining an *observer* in Spoofax. The observer is the strategy that is executed every time a new AST is created. Following the example Spoofax project, this strategy is called `editor-analyze` and is shown in Figure 4.5 for EpiSpin. It matches a tuple with the AST, the local path from the project root to the *Promela* file and the path of the project. This tuple is transformed in a tuple of the analyzed AST and three lists: errors, warnings and notes. Notes are not used in EpiSpin and therefore the last list is just an empty list. On line 5 an internal Spoofax strategy is called on the path term that is mentioned in line 4. For now, only the code on line 6 is interesting. It defines the new AST as the return value from the `analyze` rule that is applied on the original AST. This rule is found on line 9 and states that first the `desuguar-all` and then the `rename-all` rules are applied. What these rules do is explained in the following sections.

Desugaring

Desugaring is a transformation that is used for removing syntactic sugar from the abstract representation. Syntactic sugar is the syntax in a language that exists to simplify certain complicated constructs. With desugaring, syntactic sugar is rewritten into its alternative

```
Active? "proctype" ID "("
  ")" Priority? Enabler? "{"Sequence "}"  -> Module {cons("Proctype")}
Active? "proctype" ID "(" One-decl (Sep One-decl)*
  ")" Priority? Enabler? "{"Sequence "}"  -> Module {cons("ProctypeParam")}
```

Figure 4.6: EpiSpin grammar concerning proctype-declarations.

form. Transforming higher-level constructs into more general constructs is considered as desugaring as well. Desugaring can make later analysis and transformations simpler since only a subset of the language has to be analyzed. Desugaring is started when the `desugar-all` rule is called. The AST is then traversed bottom-up and the rule `desugar` is tried on every term, applying transformations to matching terms. The source code itself is not changed during analysis, only the abstract representation is.

In EpiSpin desugaring is mostly used for transforming very high-level language constructions into more general ones. An example of such a case is found in the `proctype` declaration. The EpiSpin grammar matches `proctype` declarations with and without parameters to two different constructors, as can be seen in Figure 4.6. A `proctype` declaration without parameters can be seen as a specific case of a `proctype` declaration with zero parameters. Therefore, the following transformation is applied to desugar the `Proctype` constructor:

```
desugar:
Proctype(active, name, prio, enabler, seq*) ->
ProctypeParam(active, name, [], [], prio, enabler, seq*)
```

After analysis, all `Proctype` constructors are replaced by `ProctypeParam` constructors, which are having empty lists as argument terms. The reason that a `ProctypeParam` constructor has two fields for parameters is because a list of *One-decl*s separated by *Sep*s can only be expressed in SDF by splitting the first or the last element from the rest of the list (since there is one declaration more than the amount of separators).

When a `proctype` declaration with more than one parameter is matched, two other desugar rules are used, which are shown in Figure 4.7. The first rule is for combining the first parameter with the list of parameters and is only executed when the argument list is not empty. This check can be seen on line 5. If the argument list is not empty, `arg` and `args*` are put in a new list and the sublists are removed by calling the `flatten-list` strategy, as is seen on line 6. With the next rule the separator constructor is removed from this list since the separator constructor has no semantic meaning. This is done by matching a tuple of a separator and a declaration and only returning the declaration.

```
1 desugar:
2 ProctypeParam(act, name, arg, args*, prio, enabler, seq*) ->
3 ProctypeParam(act, name, [], args2*, prio, enabler, seq*)
4   where
5     not([]:=arg);
6     args2*:= <flatten-list> [arg, args*]

7 desugar:
8  (Sep(), Decl(vis, type, ivar*)) -> Decl(vis, type, ivar*)
```

Figure 4.7: Desugar rules for a proctype declaration.

Name analysis

After desugaring, the AST is traversed top-down for name analysis by calling the `rename-all` strategy. Name analysis is also performed by specifying transformation rules. When a term is matched, the corresponding `where` clause is executed and if it succeeds, the transformation is applied. Name analysis has two main functions. The first objective of name analysis is to obtain an AST where all identifiers are unique such that two identifiers with the same name, but in different scopes, can be distinguished from each other. The other objective is to store identifiers and add other semantic information - usually by defining dynamic rules (see Section 2.3) - which are used for error checking and other semantic editor services.

Renaming is started by calling the `rename-all` rule from the main analysis file. This rule starts several traversals of the AST. But before real name analysis is performed, first two other transformations are applied concerning macro `include` statements and inline declarations. The Stratego library contains a useful strategy for importing other source files called `open-import`. When a filename and a rule are given to this strategy, the file is parsed and the rule is applied. This strategy is used to include source files that are included in a *Promela* file using the macro `include` statement. In one traversal all `include`- statements are matched and the included files are parsed. Then the `desugar-all` and `rename-all` rules are called on the parsed file to perform desugaring and name analysis on the included file first. This way, identifiers from other files are stored and can be used in the current *Promela* file.

In the same traversal in which `include` statements are analyzed, inline declarations and inline calls are analyzed as well. During this traversal for all inline declarations a transformation is defined (using a dynamic rule) from the name of the declaration to the declaration itself. All inline calls are transformed into the corresponding declaration using the previously defined transformation. This way the new AST contains an expanded inline call which can be renamed during name analysis taking all scope rules into account. The result of this transformation can be seen in Figure 4.9 for the *Promela* code in Figure 4.8. The left part of the Figure shows the abstract representation of the code directly after parsing. The right part shows the representation after analysis (so after name analysis as well, which will

```
inline example(x, y){
       y = a;
       x = b;
}

init{
       int a, b;
       example(a, b);
}
```

Figure 4.8: A Promela fragment using the inline construct

be discussed in the next paragraph). Fragment A shows the content of the inline declaration `example` and Fragment B shows the inline call with parameters `a` and `b`. After analysis, the contents of the inline declaration are removed (as can be seen on the second line of the right AST) and the code in Fragment B is transformed into the code of Fragment A which results in Fragment C. The actual parameters `x` and `y` are transformed into the actual parameters `a` and `b` as indicated by the arrows, resulting in expanded inline calls that can be analyzed during name analysis.

The general structure of renaming [22, 37] is already explained in Section 2.3 on page 13. It shows how an identifier in a variable declaration is annotated using the helper strategy `rename-var`. This strategy stores three dynamic rules that define a transformation: `RenameId` is a mapping from the old name to the new name, `TypeOf` stores the type of the variable and `VarLookup` maps the new identifier back to the old one. When a variable occurrence is matched, another strategy is called which transforms this variable to its new value by calling the `RenameId` rule. A variable occurrence can only be successfully renamed when the dynamic rule `RenameId` is defined before, i.e. when the variable occurs after the declaration. In *Promela* this behaviour is wishful for variables, but not for all structures. Consider a `proctype` declaration and a `run` statement. The `run` statement may be placed before the `proctype` declaration in a *Promela* model. It is not possible to do the renaming of both constructs in one traversal since it might happen that a `run` statement occurs before a `proctype` declaration, meaning that the dynamic rule that maps the old `proctype` name to the new name is not yet defined when a `run` statement is matched. Therefore renaming is done in two traversals, using `renamep1` for the first traversal and `renamep2` for the second traversal. During the first traversal all variable declarations that should be visible in the entire scope (in contradiction to variables that should only be visible after their declaration) are stored and renamed. These are the `proctype` variable definitions (not the sequence inside the `proctype` declaration), named `never` claim and LTL definitions, labels and `mtype` definitions. In the second traversal the remaining variable declarations, all variable occurrences and the scoped sequences (like the one of a `proctype` declaration) are renamed.

When writing rename rules, the different scopes of the language should be taken into account as well. This is implemented in Stratego using dynamic rule scoping [6, 22]. The following example illustrates how this is done. Consider the rename rule in Figure 4.10. On line 4 and 7 the start and end of a scope are defined. Any definition of the rule indicated on line 5 - `RenameId` - that is made from within the scope is no longer visible after the

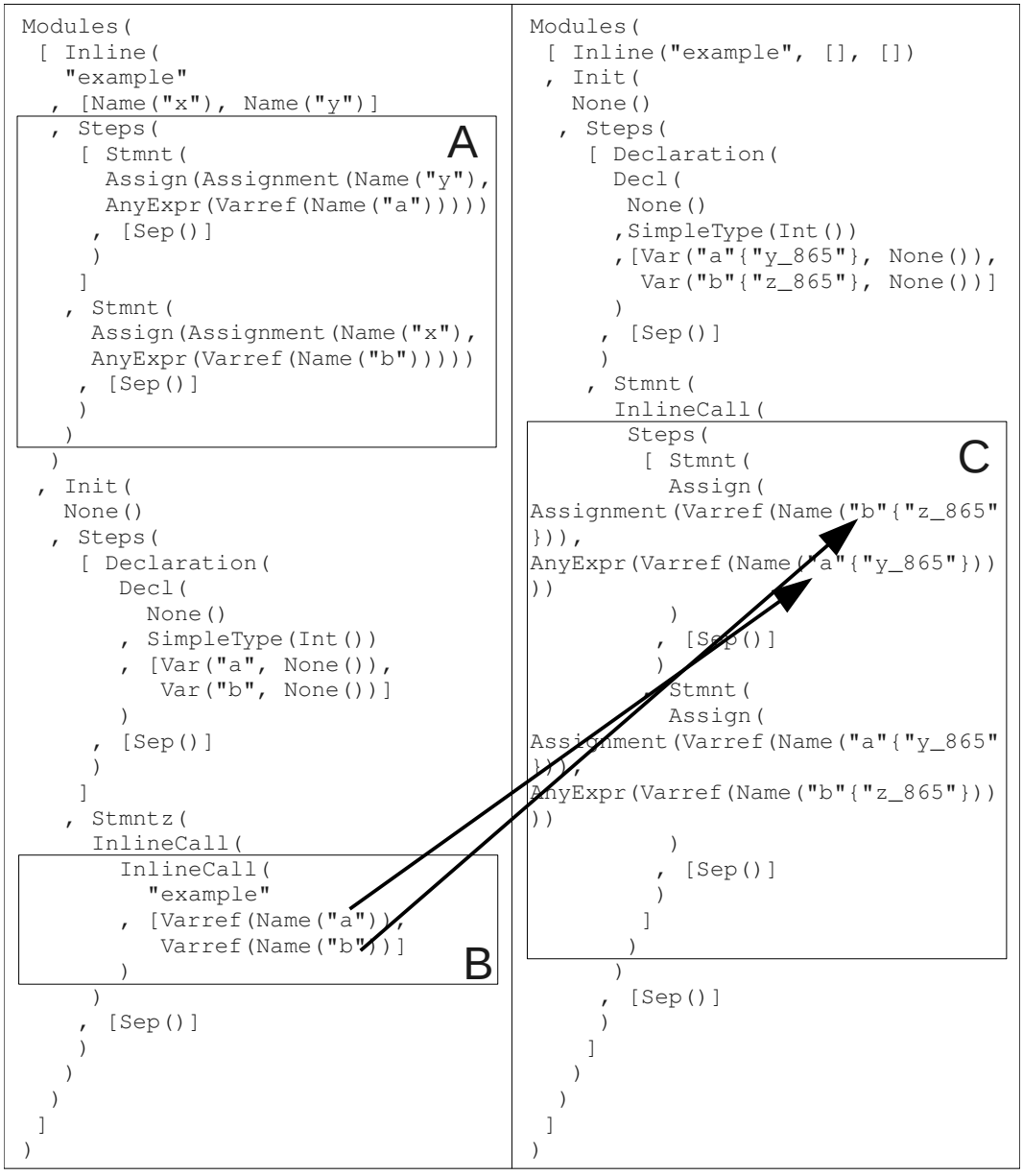

Figure 4.9: AST's of the Promela code in Figure 4.8 before (left) and after (right) analysis

```
1 renamep2:
2   i@Init(prio, seq*) -> Init(prio, seq2*)
3   with
4     {|
5       RenameId
6       : seq2*:= <alltd(renamep1); alltd(renamep2) > seq*
7     |}
8     ;
9     rules(Active := i)
```

Figure 4.10: Rename rule for the init construction

scope has ended. When an `Init` constructor is matched, the sequence term is replaced by an analyzed sequence term as shown on line 6. This transformation is again done in two traversals: first the strategy `renamep1` is applied top-down, followed by the strategy `renamep2`. The strategy `renamep1` can only match a label constructor in this scope, since the other three possible constructors can only be found in the global scope. When during the second traversal a variable is matched, only a `RenameId` rule that is defined in the same scope can be applied. On the last line (and outside the scoped definitions) the rule `Active` is defined, which just stores the init construction. This rule is also defined when an `active proctype` is matched and is used by error checking to check if there is a runnable process in the model. Besides `init`, the following structures have their own local scope: `proctype` declarations, `never` claims, `trace` and `notrace` claims and the inner sequences defined between `{...}`. The rename rules for these constructs are very similar to the one just shown.

So name analysis is done in three traversals. In the first traversal included files are analyzed, then the identifiers that should be accessible in the complete scope are stored and renamed and in the last traversal all other identifiers are stored and variable occurrences are renamed.

4.3 Editor services

After analysis every variable in the AST can be uniquely identified using its annotation. Also a lot of semantic information is stored in dynamic rules. Now error checking and other editor services only need to traverse the AST and match a relevant constructor, check some constraints using the dynamic rules and return the appropriate format for that particular editor service. In this section the implementation of all editor services will be discussed, starting with the syntactic editor services such as the outline view, code folding and syntax highlighting. After that the implementation of semantic editor services will be explained: error checking, content completion and reference resolving.

4.3.1 Syntactic editor services

The syntactic editor services that are implemented in EpiSpin are the components from the Presentation block in Figure 4.1: the outline view, syntax highlighting and code folding. All of these editor services are directly derived from the grammar. For syntax highlighting the default coloring scheme is used that is generated by Spoofax: purple keywords, blue strings, dark green numbers and lighter green comments. This is a familiar scheme for regular Eclipse users since it is used to highlight Java code as well.

All syntactic editor services are defined using the Editor Descriptive Language that is developed for Spoofax. For syntax highlighting token-based coloring is implemented. This way a color can be specified for every kind of token (a keyword, number, identifier). The color should be given in RGB values. The elements in the outline view and the structures that should be foldable are specified by enumerating sorts and constructors. Spoofax then processes these lists and constructs the outline view and the folding mechanism.

When building the outline view, Spoofax is considering the structures that are listed in the *Promela-Outline.esv* file. For every structure, the first identifier that is encountered is used to identify the structure in the outline view. When a language structure is found that is used inside another element that is shown in the outline view, the inner structure is shown indented in the outline view. For proctype declarations, this works very well because a proctype declaration has a name and this name is the first identifier of the proctype and thus appears correctly in the outline view.

The *Promela* language also contains structures that do not have an identifier in their declaration such as the `init` structure. The problem is that `init` is a keyword and is therefore not included in the AST. However, Spoofax still tries to display an identifier and therefore the first identifier that is used or declared inside the `init` block will be shown in the outline view. Only when there is no identifier the name of the constructor is displayed. Unfortunately, this behaviour can not be changed in the current Spoofax version without making changes to the Spoofax source code itself. Since the outline view is directly built after the parsing phase is completed, it is not possible to workaround this restriction by using special tricks during semantic analysis. Even with this downside, it is chosen to implement the outline view in EpiSpin, since most of the *Promela* structures are shown correctly. All module elements are included, except `mtype` variables. Furthermore global and local variables are shown and so are labels.

Code folding is implemented in Spoofax in the same way. All language constructs that are usually written on multiple lines are foldable. These are all constructions in which a sequence of statements can be written, together with the different C-blocks that can be used in *Promela*, LTL claims and multi-line macro definitions. The implementation for code folding in Spoofax works correctly, except for the case when a syntax rule starts with an optional non-terminal and that non-terminal can not be matched in the source code. Spoofax then places the folding sign one line too high. The outline view and code folding are illustrated in the screenshot in Figure 4.11. On the left-hand side (part of) a model which prints the Fibonacci sequence is shown. On the first line a minus symbol with a line below it is shown in the border. Clicking on this sign folds everything along that line, which is the complete proctype in this case. Note that the folding sign for the proctype declaration

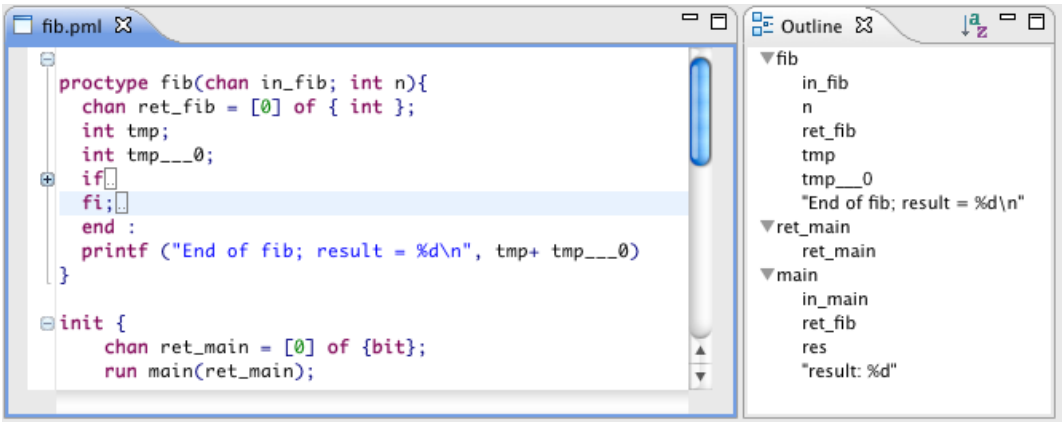

Figure 4.11: This screenshot shows code folding and the outline view of EpiSpin

is one line off because there is no (optional) `active` keyword for this declaration. A folded `if` structure can be found half way the source code. On the right-hand side the outline view is shown, showing two proctypes (*fig* and *main*) and the init declaration with their local declarations. Both the `end` label and the init declaration are not represented correctly in the outline view since both are keywords and thus not identifiers. Therefore the first identifier (or the string of the `printf` statement) that is found is printed. In the outline view the last element of the *fib* process is the string of the `printf` statement instead of the name of the label and on the line below, the `init` declaration is indicated by the name of the first identifier in the declaration, being *ret_main*.

4.3.2 Semantic editor services

When considering the semantic editor services, a distinction can be made between the error checking mechanism and the other editor services. Error checking is performed with every change that is made in the model, while content completion and reference resolving are services that are explicitly requested by the user and are only possible for certain constructs. In the next section error checking will be discussed and the subsequent sections will cover content completion and reference resolving.

Error checking

Error checking should be done after every change in the model, and after the analysis of the AST. Lines 7 and 8 of Figure 4.5 showed how error and warning messages are collected. The (predefined) strategy `collect-all` collects all subterms of `ast2` for which the strategy `constraint-error` succeeds and concatenates all these transformed subterms using the predefined rule `conc`. This list is saved in `errors` and returned by the observer rule `editor-analyze` after which it is handled by Spoofax internally. The strategies `constraint-error` and `constraint-warning` are defined manually for every error and warning message that are included in EpiSpin. Since errors and warnings are implemented

exactly the same way and only a few warning messages are defined, only the implementation of error messages will be discussed. In the next paragraphs different approaches for the `constraint-error` strategy will be discussed for different kinds of errors.

In section 2.3 the general structure of `constraint-error` strategies has been discussed and an example has been given given on how to find and report undefined variables. This is one of the most basic error messages that should be included when real time error checking is supported. Using the same mechanism an undefined label in a `goto` statement is reported. For this, the following strategy is written:

```
constraint-error:
  g@Goto(name) -> (g, $[Label [name] is undefined])
    where not(type-of)

type-of = ?Goto(name); <TypeOf>name
```

When a `goto` statement is matched, and the rule `type-of` does not succeed (so the dynamic rule `TypeOf` is undefined for the name in the `goto` statement), a tuple is returned containing a target term on which the error message should be attributed and the error message that should be shown. In this case, *g* is the target term which is the complete `Goto` constructor (`g@Goto` stores the constructor in the variable *g*) and the error message is shown between `$[...]` which indicates a string. The name of the label will be evaluated since it is a variable placed between `[...]`. This error message is very simple since the main part of the constraint checking is already done during analysis by storing all defined variables.

A related kind of error is shown when identifiers are defined multiple times. During name analysis an identifier is annotated with a unique string, but when the identifier already exists, the annotation will consist of the string “double”. When the AST is traversed for error checking, constructors that contain an identifier are checked for this annotation and when it is found, an error message is created. Such a strategy then looks like:

```
constraint-error:
  Label(name{an}, stat) -> (name, $[This label is declared before])
  where
    "double":=an
```

New in this example is `name{an}` in which the annotation is explicitly matched to a term as well, which is then checked against the term “double”.

Another type of error is calling a previously defined structure that needs parameters, such as calling an inline declaration or instantiating a new proctype. It is an error to call these structures with the wrong amount of parameters. The `constraint-error` rule for matching `run` statements that contains such an error is:

```
constraint-error:
  r@Run(name, args*, prio) -> (r,
            $[[name] is called with the wrong amount of parameters])
  where
    not(<eq> (<length>args*, <AmountParam>name))
```

In the `where` clause the length of the arguments list is compared to the rule `<AmountParam>name` which is defined for every proctype during renaming and maps the proctype name to the length of the parameter list. When the lengths of these lists are unequal, the transformation is made and the error tuple is collected.

Every language construction has it own errors. Variables in channel statements are checked whether the type of the variable is a channel and variables with an index are checked to be of the array type. Also several checks on user defined types are done. User defined types allow *Promela* users to define and use their own types, like:

```
typedef Node{
  chan ch = [1] of {int};
  int id;
}

Init{
  Node n;
  n.ch!42;
  n.id = 0;
}
```

One of the most complicated constraint to check in EpiSpin is whether the field *ch* is actually defined for the variable *n*. This is done in several steps, as shown in Figure 4.12. A variable of a user defined type is parsed as an `ObjectElement` with the variable name as the first term and the field name as the second term. Note that the variable name can also be a field of another user-defined type. Then the type of this first part is requested by calling the `get-type-of` rule. On this type the rule `bagof-UTypeVar` is called, which returns a list of all variables that are defined for that type. Finally, it is checked whether the field name is in that list. Note that the first part of a user-defined variable can contain errors as well, and that is why the `constraint-error` strategy is called recursively on this first term. The returned list is concatenated to the error tuple that is created for the field term and that is the result of the error check strategy on a user-defined variable.

Another group of errors arises when certain language constructs or keywords are used at a prohibited place, although it is syntactically correct. This includes a `break` statement outside a `do` loop or an `enabled(...)` expression outside a never claim. These cases are checked very differently compared to the previously described procedures. This is explained using the `break` statement. During name analysis, all `break` statements that are found within a `do` structure get an “approved” label. Then during error checking all `break` statements that do not have such a label are transformed to an error message.

```
constraint-error:
  ObjectElement(uref, idt) -> <flatten-list>[insideErrors,
             (idt, [No field [idt] defined in structure [<pp>uref]])]
  where
    insideErrors:=<collect-all(constraint-error)>uref;
    typename := <get-type-of> uref;
    not(<elem>(idt, <bagof-UTypeVar> typename))
```

Figure 4.12: A constraint-error strategy for checking for the use of non-existing fields in user-defined types

The last error that is discussed is a special kind of error. It is an error that does not belong to a certain language construct, but relates to the complete program. An example of such an error is the "No runnable process" error which is shown when no `init` construction and no active proctype is defined. The constraints of the corresponding strategy are not hard to check, but there is no obvious position where to show this error. Therefore the target term in the `constraint-error` strategy will be the most outer constructor, the `Modules` constructor and in practice Spoofax always shows this error on one of the first lines.

Content completion

Content completion is an editor service that shows the user a list of possible options on how to complete the current line (i.e. where the cursor is) of the code. Syntactic and semantic content completion can be distinguished and although they are shown together in the same proposal window, they are both implemented differently. Syntactic content completion is responsible for completing static structures of the *Promela* language, such as proctype structures or repetition structures. The templates of these structures are written in the file *Promela-Completions.esv*. These templates are built using strings, placeholders and layout characters (`\n`, `\t`). Furthermore it is possible to specify the sorts (non-terminals) where such a completion suggestion may be offered. The templates for the proctype and repetition structure looks as follows (note that the new line in the Module template has no semantic meaning, but is only there to fit the page width):

```
completion template: Module = "proctype " <name> "(" <>
                              "){\n\t" (cursor) "\n}"
completion template: Stmnt = "do" "\n:: " (cursor) "\nod;"
```

Both templates contain the strings that form the structures and some tab and newline characters to increase the readability. `<name>`, `<>` and `(cursor)` are different placeholders, indicating positions where user input is allowed. Using the Tab key on the keyboard the cursor can be moved to the next placeholder, ending at the `(cursor)` position. Both templates start with the names of the sorts where they can be placed. When content completion is

requested at a position where a only a statement is allowed, only the templates starting with `Stmnt =` are shown. Content completion can be requested manually in Eclipse by using the Control + Spacebar combination on the keyboard, but can also be started using pre-defined triggers. When such a trigger is typed in the *Promela* model, content completion is started.

Semantic content completion offers user-defined identifiers instead of pre-defined structures. It uses the AST to get semantic information for the current position and based on that information a list of proposals is shown. With semantic content completion, two main decision phases can be distinguished. First it should be determined what kind of structure is being completed (a variable reference or a `run` statement for example) and then the possible identifiers for that structure should be established. When (semantic) content completion is triggered, the source code is usually in a state where it contains syntax errors. Consider for example the `goto` statement, which grammar rule is `"goto" ID -> Stmnt{cons("Goto")}`. Content completion is wanted for this structure after typing the `goto` keyword. Incomplete structures are normally not shown in the AST, so therefore a special `COMPLETION` term is used to show the incomplete structure in the AST. In this case, the constructor `Goto(COMPLETION(""))` is inserted in the AST, which is then given to the `editor-complete` strategy. On this point, Spoofax also defines the *candidate sorts*, i.e. the sorts that can occur at the position of the completed token. This process of establishing which sort is expected at the current point is handled by Spoofax internally. Error recovery [20] is used to estimate how the code should be completed. In this area a lot of active research by the Spoofax developers is going on to optimize this process. Currently, users of Spoofax need to help the parser occasionally to decide which rule should be matched. An example of such a case is when the `run` statement is completed. The syntax of this statement is `"run" ID "(" {AnyExpr ","}* ")" Priority? -> AnyExpr{cons("Run")}`. Content completion is triggered when the `run` keyword is entered, but the parser is not able to parse this as a partial `run` statement due to the high amount of missing elements. This means that content completion fails for this case. A workaround here is to include another rule in the syntax definition, matching only the `run` keyword and the proctype name: `"run" ID -> AnyExpr{cons("Run")}`. Now this rule can be matched during content completion to complete the identifier. The parenthesis and optional parameters are completed as well, but this will be discussed in the next paragraph, where the process of deciding which identifiers will be shown is explained.

So now the source code is parsed and an AST is built with the `COMPLETION` node inside. The candidate sorts are determined and are used to determine which syntactic templates should be included in the completion list as well. For semantic content completion the identifiers are determined by calling the `editor-complete` strategy. This strategy first applies desugaring and renaming on the AST, as is done after a normal parse. Then the AST is traversed again by applying the `complete-proposal` strategy topdown. This strategy matches the `COMPLETION` token and returns a list with completions. Because there are different kinds of identifiers (proctype names, variables, labels) and the *Promela* language supports different scopes, it is not possible to simply return all the declared identifiers as a list of possible completions. Both aspects should be taken into account. Therefore a lot of information that is needed for content completion is already collected during name analysis. This will be explained using an example. Consider again the `goto` statement. When a `goto`

statement is used inside a proctype, all labels that are declared inside that proctype should be shown in the completion list. The only moment to obtain such scope specific information is during renaming. Therefore two additional elements are added to the renaming process. First a scoped dynamic rule `Labels` is created which stores all labels that are created. This rule can only be used from within the current scope, meaning that when this rule is called, only the labels from the current scope are shown. Secondly, a new renaming strategy is added that matches the `COMPLETION` token. This strategy creates an unscoped dynamic rule to store all the labels from the current scope. This rule is used by the `complete-proposal` strategy to create the proposal list by requesting all elements of that rule. For the `goto` statement, this looks as follows:

```
1 renamep2:
2  Goto(COMPLETION(_)) -> <id>
3  where
4    vars:= <all-keys-Labels> ;
5    rules (CompletionVarsGoto:= vars)
```

On line 4, all labels that are stored in the `Labels` rule are stored in a variable `vars` and on line 5 these elements are stored in a new dynamic rule `CompletionVarsGoto`, which is used by the `complete-proposal` strategy. The reason that `complete-proposal` does not directly call `all-keys-Labels` is because the completion strategy does not have direct access to that rule because that rule is scoped. Only during name analysis this rule is directly accessible. The `complete-proposal` strategy is then:

```
complete-proposal:
  Goto(COMPLETION(_)) -> <CompletionVarsGoto>
```

During the renaming phase, it is also possible to customize the identifiers that are returned for completion. This is useful for the `run` statement, where not only the name of the proctype should be completed, but parenthesis and possible parameters should be supplemented as well. The list of names of the proctypes is retrieved in a similar way as described above, using a dynamic rule that stores all proctype names. But before this list is returned, all elements are extended by parenthesis and the names of the formal parameters by calling a strategy that performs this transformation on each element of the list.

Besides for `goto` and `run` statements, semantic content completion is implemented for

- normal identifiers in expressions, showing all global and local identifiers;
- positions where only channels can be used, such as in channel poll operands or after exclusiveness declarations. In these cases, only previously defined channels are shown;
- user-defined types. When a variable of a user-defined type followed by a dot is typed, completion is triggered as well. The completion list shows all variables that are defined for that type.

Reference resolving

Reference resolving is the easiest semantic editor service to implement, since most of the work is handled internally by Spoofax. With reference resolving the cursor is moved to the declaration of an identifier by clicking on the identifier when holding the Control key on the keyboard. Reference resolving is handled by the *Promela-References.esv* file which only points to a strategy `editor-resolve`. When a user control-clicks on any element in the source code, this strategy is called. When the constructor of the clicked element can be matched against one of these `editor-resolve` rules, a node is returned. Spoofax provides position information for all elements in the AST. When such an element is stored in a dynamic rule, this position information is remembered as well. Spoofax extracts the position information of that node and moves the cursor to that position. For all identifiers such a rule is written. For variables and labels these rules look like:

```
editor-resolve:
  (Name(x), position, ast, path, project-path) -> <VarLookup> x

editor-resolve:
  (Goto(x), position, ast, path, project-path) -> <VarLookup> x
```

In both cases the original name of the identifier is returned using the dynamic rule `VarLookup` which is defined during name analysis.

4.4 Static Communication Analyzer

The Static Communication Analyzer is a communication graph that shows the interactions between proctypes, channels and global variables. It is compiled from DOT [10] code that is generated by EpiSpin when ”Generate Dot Code” is selected from the Transform menu. The strategy `generate-dot` that is coupled to this action is then executed. This strategy creates a new file with the `dot` extension and the same filename as the file for which a graph should be generated. The DOT code is generated by transforming the analyzed AST of the *Promela* model into DOT code. This process is started by the calling the strategy `to-dot` on the AST. About 50 of these strategies exist, all matching another constructor of the AST. The outer constructor is always the `Module()` constructor, containing a list of all modules in the model. Therefore the first strategy that is matched is always the same:

```
to-dot:
  Modules(m*) ->
    $[digraph{
      [m'*] }
     ]
  with
    m'* := <map(to-dot<+success)> m*
```

A transformation is applied by matching the `Modules()` constructor and returning a string. This string will be printed in the file generated in the `generate-dot` strategy. In Stratego a string is placed inside `$[...]`, and elements between square brackets are variables that are evaluated. They need to hold a string value in order to be printed correctly. The text `digraph{}` is the printed DOT code that declares a new directed graph. The element inside this graph is `m'*`, which is a list of modules transformed by the `to-dot` strategy. Almost every `to-dot` strategy only passes the value of the `to-dot` strategy applied to the subterms, without printing new dot code itself. Take for example a repetition structure, which itself does not need to be transformed in DOT code, but can contain other elements that should be included in the graph. The `success` strategy is only executed when the `to-dot` strategy fails and matches anything and returns an empty string. This strategy ensures that every Module returns a string when it is transformed, even when there is no `to-dot` strategy defined. `Typedef` and `mtype` structures are ignored completely for example.

The rest of the strategies basically do the same trick as in the example above, using one or more helping strategies to filter out the names that should be printed. The implementation of the SCA will be discussed using the same order as maintained in Section 3.2: first the basic cases are discussed, being the definition of channels and other variables, running a proctype (without giving a channel as parameter) and using these variables and channels in expressions (without sending channels over channels). Then channels in message fields and as actual parameters in `run` statements are discussed and finally the implementation of encapsulated channels is explained.

4.4.1 Basic cases

The strategy that transforms global variables into DOT code is simple. A global variable definition is a module and at a certain moment `to-dot` is called on that module, matching either a channel declaration or otherwise a variable declaration:

```
to-dot:
  GlobalDecl(Decl(vis, SimpleType(Chan()), ivar*)) ->  dotcode
  where
    dotcode:=$[[<map(to-dot-chan)> ivar*]]


to-dot:
  GlobalDecl(Decl(vis, type, ivar*)) -> dotcode
  where
    dotcode:=$[[<map(to-dot)> ivar*]]
```

For both of these rules the identifier term `ivar*` is extracted, which is a list of `Var()` terms that contain the identifier and the initialization value. This term is transformed using the `to-dot-chan` or `to-dot` strategy, which is the strategy that actually returns DOT code. Since channels and other variables are represented differently in the graph, separate strategies are used. The transformation `to-dot-chan` is:

```
to-dot-chan:
  Var(name, init) -> $[[<with-anno>name] [esc]"[name]"[esc2];]
  where
    esc:="[shape=hexagon, style=filled, fillcolor=limegreen, label=";
    esc2:="]"
```

Two remarks should be made about this fragment (and will apply to coming fragments as well). First, the annotation of an identifier (which is created during name analysis) is normally not printed unless this is specified explicitly. The annotation is required to distinguish two identifiers with the same name (but from different scopes) in the DOT code. The rule `with-anno` ensures that the annotation is printed. Secondly, attributes of node declarations in DOT code are placed between square brackets, which are also used in Stratego strings to evaluate a variable. Therefore a workaround is used to actually print these square brackets by using string variables (`esc` and vbesc2) that contain these brackets. The attributes in the DOT code specify that a channel is represented by a hexagon shape that is filled with the color lime green. The default name that is shown in the shape is overridden by specifying a label, which is just the name of the variable (instead of the name with annotation). So for the declaration of a channel *ch*, the resulting DOT code looks like `chk_2201 [shape=hexagon, style=filled, fillcolor=limegreen, label="ch"];`.

The second basic case is the implementation of proctype structures in the communication graph. *Promela* proctypes are only interesting to include when they are actually instantiated, which is done by using the `run` statement or by using the `active` keyword in front of a proctype declaration. When a *Promela* model is run, only the active proctypes are instantiated immediately. The other proctypes are instantiated at the moment a `run` statement is executed. With the generation of DOT code, the same is done. Only proctype declarations with the `active` keyword are matched and directly printed when the list of Modules is traversed. Other proctypes are only transformed at the moment that they are instantiated by the `run` statement. When a `run` statement is matched, the name of the proctype that is holding the run statement (or "init" when the statement is executed in an `init` block) is returned followed by an arrow and the name of the proctype that is being started. This strategy is called `to-dot-run` can be found in Figure 4.13. To avoid duplicate arrows, first it is checked whether this relation is not printed before. This is done using dynamic rules: on line 5 it is checked if the rule *GetRuns* was not defined before for the `run` statement *r* in proctype *name*. If it was not defined, line 6 defines it. On line 2 and 3 the transformation can be seen with the arrow between the two proctype names. The name of the current proctype is `name` and is given as a parameter to this strategy. To transform the called proctype as well, it is requested first by calling the `GetProctype` rule on line 8 and save it in `proc`. This rule returns the proctype constructor with the name `idt`. On line 9 a strategy is called that modifies this proctype. For now, it is only important to know that the proctype constructor that is returned by the `get-modified-proctype` strategy on line 9 (and saved in `proc2`) is a copy of the called proctype and that it contains the `active` keyword. On line 10 this proctype is transformed using `to-dot` and the result is saved in `proctypedot` that is printed on line 3. Since `proc2` now contains the `active` keyword, it can be transformed using the

```
 1 to-dot-run(|name):
 2   r@Run(idt, params*, prio) ->
 3      $[[<with-anno>name] -> [<with-anno>idt] [esc]; [proctypedot]]
 4   where
 5     not (<GetRuns> (r, name));
 6     rules(GetRuns:+ (r,name) -> <id>);
 7     esc:="[style=dashed]";
 8     proc:= <GetProctype> idt;
 9     proc2:= <get-modified-proctype(|params*)> proc;
10     proctypedot:=<to-dot>proc2
```

Figure 4.13: This strategy prints the relation between two proctypes and the DOT code for the called proctype

same strategy that is used to generate DOT code for active proctypes.

The last basic case covers the interaction between a proctype and a global variable or a channel. In the SCA an arrow is drawn from the proctype to the variable for read and send operations and the other way around for write and receive operations. These operations are found by applying the `to-dot` strategy recursively on a term and its subterms, until a send or receive statement is matched, an assignment is matched or an expression is found, in which case the expression is searched for variable usage and run statements. When a send or receive statement is matched, an arrow between the current proctype and the statement is printed, provided that this relation has not been printed before. When an assignment statement is matched, the left hand side is printed using the `to-dot-var-write` strategy which prints an arrow from the variable to the proctype, provided that the variable is global or represents a channel and provided that the relation is not printed before. The right-hand side is an expression and every expression is searched for variable occurrences. These variables are then printed using the `to-dot-var-read` strategy, which performs the same checks as the write-version of this strategy, but prints the relation the other way around. Also `run` statements are searched and the strategy `to-dot-run` is applied to the `Run()` constructors as described above.

4.4.2 Channels as proctype parameters and message types

This subsection covers the case where a channel is declared in one proctype and is send as a parameter in a `run` statement to instantiate a new process. Both proctypes should interact with the same channel, although it might be assigned another name in the called proctype. To make sure both proctypes point to the same channel in the graph, a duplicate is made of the called proctype and this duplicate is then modified. This is what is done on line 8 and 9 in Figure 4.13. The `get-modified-proctype` not only inserts the `active` keyword, but also replaces the names of formal channel parameters with the names of the actual parameters.

This is illustrated with the example from page 25 where a channel *ch* is declared in an `init` block and is then used as a parameter to instantiate a process from proctype *procA*:

```
init{
  chan ch = [0] of {int}; // Declaration of a rendezvous channel
  run procA(ch);
  ch ! 42;  //sends the value 42 to channel ch
}

proctype procA(chan c){
 c ? 42; //read the value 42 from channel ch
}
```

In order to make sure that interactions between *procA* and *c* are displayed as interactions between *procA* and *ch*, the proctype declaration of *procA* is modified. `get-modified-proctype` traverses the sequence of this proctype declaration and every time a variable *c* is encountered, this variable name is transformed into *ch*. When the `to-dot` strategy is called on this sequence later on, it matches a send statement that involves the `ch` variable instead of the `c` variable and an arrow is drawn between *procA* and *ch*.

Channels in message fields

The case where a channel is sent as a message over another channel is only supported very limited as explained in Section 3.2. The only way this is shown in the graph is by a label that is attached to the arrow between the proctype and the channel. This label is inserted by the strategy that matches a send or receive statement as discussed in previous section which is responsible for printing the arrow. It scans all the arguments that are sent or all variables of the receive statement and collects all channel variables. These channel names are then labeled to the arrow so it is visible how these channels are transferred, as shown in Figure 3.2.

4.4.3 Encapsulated channels

Channels can be hidden inside an array or inside a user-defined type. In Section 3.2 it is decided that array variables are treated the same as ordinary variables and that the different elements of the array are not displayed one by one. The only difference is that encapsulated channels are shown in dark green. Channels from user-defined types are handled the same way as normal channels. The label of such a channel includes both the variable name and the field name(s), like *n.ch*. To support this in the graph, strategies should be made that match an `ObjectElement`, which is the constructor that is matched when a variable of a user-defined type is used and is the equivalent of the `Name()` constructor that is matched when a regular variable is used. These strategies perform the same operations as for the `Name()` constructor, printing errors between proctypes and variables. Because channels in user-defined structures behave the same as regular channels, all strategies that are called for printing DOT code for channels can be reused.

Chapter 5

Testing

The domain of software testing is an important subdomain within the domain of software engineering. Testing ensures that a software system meets its requirements. Testing EpiSpin is done using the integrated testing language of Spoofax [21]. First the parser is being tested as is explained in the first section. This is done by creating a test set that covers all rules of the EpiSpin grammar and by parsing a batch of existing *Promela* models. Section 5.2 explains how the editor services are tested. First error reporting is testes using similar steps. All error messages are tested separately and then the same batch of files is used to check if the expected amount of errors is found for all models. After that, reference resolving and content completion is tested.

5.1 Parser testing

For EpiSpin a new grammar is written that is based on several sources: *The Spin Model Checker* [15], the grammar on the manual page on the SPIN website (see Appendix A) [13] and the SPIN source code. Testing the EpiSpin parser is done by comparing its behaviour with respect to the SPIN syntax checker.

When comparing the EpiSpin parser with the SPIN parser, it is desired that those two are as similar as possible. Comparing parsers is done by comparing their languages, which is the set of input strings that a parser accepts. If the language of the EpiSpin parser contains input strings that are not in the language of SPIN, the EpiSpin parser is said to be *incorrect*. When the opposite is true (the EpiSpin language misses input strings that are in the language of SPIN), the EpiSpin parser is *incomplete* [24]. Although both cases are undesired, it is hard to avoid them. And it is even harder to test for these properties.

In theory, completeness can be discovered by parsing a lot of programs from a test set. This test set must be representative (large enough to experience all possible constructs of the *Promela* grammar) and all test cases from this test set must be accepted by the SPIN parser. By parsing more and more programs from this test set, and modifying the parser when a program is not accepted, the EpiSpin parser is complete when all programs from the test set are parsed. In practice, it is very hard to create a test set that covers all possible constructs of the *Promela* grammar. Therefore several heuristics are used to test the grammar using

smaller test sets. The first heuristic is rule coverage.

Rule coverage

With rule coverage a test set is being parsed that uses all rules from the grammar at least once. A test case is said to cover a grammar rule if that rule is used at least once while parsing that test case [12]. The EpiSpin grammar consists of 337 rules, including all combinations of optional non-terminals for a rule in the SDF grammar. For example, an SDF rule `"run" ID "(" {AnyExpr ","}* ")" Priority? -> AnyExpr {cons("Run")}` can be evaluated in four different ways: with or without a parameter list and with or without a priority expression. To create a test set to cover all rules, the upper limit of this set is 337. In practice, the test set could be much smaller since multiple rules are used to parse a single test case. For the mentioned rule, at least an expression, a module (a `proctype` or an `init` block) and the starting symbol are matched, which then should not have to be considered separately.

To test all rules of the EpiSpin grammar, a set of test cases is written manually. This is done by starting at the first grammar rule and create the simplest test case possible that matches this rule. All other rules which are also used to parse this test case are also marked as test. For every sequential rule that is not covered yet a test case is constructed. This leads to a total of 270 test cases. This number could have been smaller if not the simplest test case possible was constructed, but when multiple language constructions were taken together into one single test case. This way much more rules could have been tested in one test case. To keep the construction of test cases relatively easy, and to separate different language elements from each other, the test cases are kept as short as possible. This principle complies to the principle of unit testing [] in which all smallest testable parts are tested separately. When a test case fails to be parsed, it is immediately clear which part of the language is incomplete.

Using rule coverage for testing the EpiSpin grammar two difficulties should be dealt with [12]. First, the *Promela* and EpiSpin grammar admit constructs that are not syntactically valid. This is done to make the grammar more compact and additional error checks are implemented to eliminate these constructs (this is also discussed in Section 3.4). A basic example is that the grammar allows a `break` statement outside a repetition structure, while this is not allowed. For rule coverage only the syntax errors that are reported by the parser are interesting. Therefore any additional constraints on the syntax that are checked in a later state are irrelevant. This means a test case like `init{np_}` can be used to cover the rule `"np_" -> AnyExpr cons("NP")`, even when the `np_` variable may only be used inside never claims. The second difficulty when using rule coverage regards to ambiguities. When a grammar is ambiguous there is no guarantee that an ambiguous input string will be parsed using a certain rule. Because of the generalized parser that is used, EpiSpin always reports ambiguous input strings. If such an ambiguity is encountered, this can be solved by applying one of the disambiguation constructs as described in Section 4.1.3.

Integrated testing with Spoofax

The tests described above are performed using the Spoofax testing language [21] in which the syntax, semantics and editor services of EpiSpin are integrated in the testing language. This means that syntax highlighting, error marking and content completion can be used in the testing language. In the testing language it is possible to write a fragment of *Promela* code and to specify assertions for this fragment. These assertions can be syntactic, such as '`parse succeeds`', '`parse fails`' or '`parse to` *AST*'. Also syntactic assertions can be specified like '`1 error`' which checks if exactly one error is given for the test case. The tests are evaluated immediately in a background thread and failed test cases are reported using error and warning messages. It is also possible to run all test suites in a batch test runner. Multiple tests can be specified in the same file. When these tests are similar and share the same code this common code can be factored out using setup blocks. In a setup block the general code is placed and the several test cases are then evaluated by concatenating the current test case with the setup block and parsing the complete construction.

For testing the EpiSpin grammar different files are created that have a different setup block. For testing the different modules, no setup block is required, since they are all put in the global scope. For testing channel operations, a setup block with an `init` block and a channel declaration is made. Multiple setup blocks can be used to encapsulate test cases, so a setup block at the bottom of the testfile is concatenated to the end of each test case. Seven test files are created for testing modules, expressions, channel operations, the different variable references, variable declarations, the different sequences and steps and finally all single statements. The default assertion for the test cases (i.e. when no assertion is specified) is to check if the code fragment is syntactically and semantically correct. When a structure as `init{np_}` is tested, the assertion is explicitly specified to `parse succeeds`, which means it only checks for syntactic errors from the parser. Two examples of test cases are:

```
setup[[
  init{skip;
]]

test anyexpr conditional expr[[
  (1>2 -> 1 : 2)
]]

test anyexpr np[[
  np_
]]parse succeeds
```

Setting up the tests

Other ways of testing are considered as well. Another coverage property that could be tested is called context-dependent rule coverage [12]. To explain the concept of context-dependent

rule coverage, first the term *direct occurrence* should be explained. The direct occurrence of a non-terminal *N* in a certain grammar is the amount of occurrences of that non-terminal on the left-hand side (where terminals and non-terminals are used to construct the grammar rules). A test set is said to achieve context-dependent rule coverage when the test set covers all production rules of a non-terminal for all its occurrences. This means for example that all data types should be tested at all places where they might occur (in global scope, in a proctype declaration). For the EpiSpin grammar this means that at the most 7556 test cases need to be written. This number is calculated by taking the sum of all occurrences of the 45 non-terminals multiplied by their amount of alternatives. Especially the many recursive rules for `AnyExpr` and `LTLAnyExpr` lead to this high number (68 occurrences of the `AnyExpr` non-terminal and 43 alternatives to evaluate it). Again, it is possible to combine multiple rules in one test case and decrease the amount of necessary test cases, but the amount of combinations that should be tested to ensure context-dependent rule coverage is too high to create the test sets manually.

Instead, a large batch of available *Promela* files are used to test the parser. Three sources are used to obtain *Promela* models: the database from the Benchmarks for Explicit Model checkers [31], which contains 235 *Promela* models of 57 different problems that are relatively simple and do not use macros. The second source is the database from Alberto Lluch [26], containing 19 large programs that use macros in several ways. Due to the restricted support for macros in EpiSpin some of these programs might not be parsed correctly or not be parsed at all (that is, if error recovery fails). The final *Promela* models that are used are found on the SPIN website itself. When downloading the source code, a folder with 33 example protocols and algorithms is included.

These models are first parsed using the SPIN parser to ensure that they are valid *Promela* models. During testing it should be taken into account that last year Spin version 6 was released. This version includes some changes in the syntax compared to version 5. This means that some older models might not be parsed correctly anymore. Performing the tests leads to 219, 11 and 32 models of the three different sources that are parsed successfully. From the models that could not be parsed, about half of them are successfully parsed with SPIN version 5. The rest contains other syntax errors. These files are ignored for now and are used for semantic tests later on. The correct files are now tested by EpiSpin.

Prior expectations are that all programs without macro usage are parsed correctly. Models that do contain macros are tested twice: once after they are preprocessed by an external preprocessor before they are parsed by EpiSpin and once by parsing them directly. This concerns 30 models from the Lluch and SPIN database.

Performing the tests

Parsing the 270 test cases to obtain rule coverage is done successfully. No constructions were rejected and no ambiguities were found.

Running the test suites on the macro-free models, one test case failed. In this file, an expression `c=='\n'` was found. No support for character symbols was found in any of the *Promela* grammars and was therefore not implemented. By adding a rule in the lexical syntax that a single character surrounded by single quotes should be parsed as a constant

Table 5.1: This table shows the amount of correctly parsed models

	Spin	Epispin directly	EpiSpin after CPP
Macro files	30/36	20/30	29/30
No macro	232/249	231/232	x

value, the test file succeeds. Furthermore, from the 30 preprocessed files, 29 parse correctly. The failing test case includes two errors: first, it uses the `accept` keyword word as an identifier. Considering the fact that this program was correctly parsed by SPIN, this unveiled that the `accept` word was incorrectly marked as a keyword in EpiSpin. The same applies for `progress` and `end`, which are labels that have a special meaning during verification. It turns out that these label names can be used as identifier names as well (not in the same scope, obviously). The other error was a parsing ambiguity which is solved by adding a new priority rule to the syntax definition. When these 30 models are parsed immediately by EpiSpin, 20 of them parse correctly. Note that even when a file is parsed, this does not mean there are no semantic errors. The rejected models use macros in a way that is not included in the EpiSpin grammar or use keywords in macro definitions, which is restricted in EpiSpin.

The results of this test are summarized in Table 5.1. A distinction is made between models that contain macros and models that are free of macros. When testing a batch of models, it can not be assured that every possible construction of the language is tested. To know this for sure, the test sets should be constructed systematically. As calculated in this section, for context-dependent rule coverage this means that over 7000 combinations of rules should be tested, which is way too much to do manually. Using as much as possible existing models to test the parser is a good alternative. With the used models in this section, three different syntax errors were found and solved.

5.2 Editor service testing

For testing the editor services a distinction is made between testing error reporting in the first part and testing the semantic editor services reference resolving and content completion in subsection 5.2.2.

5.2.1 Testing error reporting

To test if the semantic errors are shown in the correct cases, the same two testing methods as for grammar testing are used: testing for rule coverage using unit testing and testing for completeness using batch testing. In this case, rule coverage means that every error message should be shown at least once. Semantic errors depend on the state of the model. An error should be shown when a set of constraints succeeds (in Stratego an error message is included when a set of constraints succeeds), but should not be shown in any other case. For every error message a separate test case is made containing the erroneous code that triggers the error message. It is easy to create test cases for which the error is shown, since this is just a very specific case which can easily be reconstructed. Showing all test cases

for which the error is not shown is not possible, since this is almost the complete language (except for that specific case). Instead, for every error message several test cases that should not show an error are constructed by identifying some corner cases.

Unit tests

All language independent errors (see Section 3.4) relate to the definition and usage of identifiers in a wrong way. In the *Promela* language several pairs of identifier definition and identifier usage can be established:

- Label & `goto` statement
- `Proctype` definition & `run` statement
- `Inline` definition & `inline` call
- `Typedef` of a new structure & the definition of a variable of that structure
- Variable definition & variable usage

The last pair does not refer to two specific language constructs, but to a set of them. A variable definition can occur in the global scope, in the local scope and as a formal parameter in a `proctype` definition. Variables are expected in 20 different rules of the grammar, for example in assignment operations, channel operations and expressions.

For each of these pairs, test cases will be constructed that test general conditions like undefined identifiers or redefinition of identifiers. Where applicable, the amount of formal parameters in the definition (in a proctype for example) are compared to the amount of actual parameters in the call (in a run statement for example). Both test cases that do expect an error and test cases that do not expect an error are constructed by varying one or more of the following properties:

- the scope can be varied of the definition and the usage of the identifier. A label that is placed in an inner scope should still be visible to `goto` statements in an outer scope, while variables declared in an inner scope should not be visible outside that scope.
- using recursion different test cases are made. A `run` statement should be able to call the same proctype in which it is used, while an inline call can not be made recursively to the inline definition.
- the order of the declaration and using the identifier can be changed. A label could be defined after it is used in a goto statement, but a variable can not.
- the amount of parameters is varied for a call, using less and more parameters than are allowed by the definition.

Variable occurrences are tested for all 20 rules for 5 different positions: in the global scope and in the local scope before they are used (no errors are expected here), in both scopes after they are used (an 'undefined variable' error is expected at all variable occurrences) and

Table 5.2: The amount of test cases for each definition / occurrence pair

labels / `goto`	12
`proctype` / `run`	15
`inline` definition / call	5
`typedef` / variable def.	3
Variable definition / occurrence	54
Total	89

as a parameter in a proctype definition (no errors are expected). Finally, all rules are tested when no definition is made at all. For a faster generation of the test cases, several of these 20 grammar rules are tested in one single test case, leading to 9 test cases for each of the 6 positions of a definition. The total amount of test cases considering variable definition and usage is thus $6x9 = 54$. The amount of test cases for the other definition / occurrence pairs can be found in Table 5.2

The second group of errors imposes additional constraints on the syntax. For every error a test case is written that should raise an error and one which should not. For example:

```
test break inside do[[
init{
  do
  :: break;
  od
 }
]]

test break outside do[[
init{
  break;
}
]]1 error
```

This means that 60 test cases are added for these kind of errors, since there are about 30 of these errors.

The remaining error messages are tested by specifying one or more test cases for that error message. These error messages are all very specific to a certain *Promela* construction and therefore it is hard to generalize over all of these errors. For some of them, only two test cases are needed, such as for 'An array can not be indexed with itself'. One test case triggers the error by trying to perform an expression like `arr[arr[1]]` and the other test case uses an array correct: `arr[1]`. Other error messages need more test cases to cover more corner cases. For example, to test if an array variable is not used without an index, multiple test cases are written that cover all ways a variable may be used, which include variables in user-defined types as well. In total, almost 200 test cases are created for unit

testing the error messages: 89 for testing the pairs of definition and occurrences, 60 for the errors that impose an additional constraint on the syntax and 48 for the remaining specific error messages.

Batch testing

The second part of the test cases contain the same programs as described in last section. Only the programs that were successfully parsed by EpiSpin are used for testing the error checking mechanism. Furthermore, the programs that were rejected by the SPIN syntax checker because of semantic errors (in SPIN, syntactic and semantic error checking is interleaved) were also tested. This means 238 macro-free models (232 from last section plus 6 models which contain a semantic error) are tested and 20 macro containing models. By specifying the amount of errors that are expected for each test case (which is obtained by parsing the program with SPIN), these test cases will reveal incompleteness and incorrectness of the error messages.

Performing the tests

Running the tests does not give any big surprises. The 200 unit tests all give the expected amount of errors, and so do the 238 macro-free models. Of the 20 macro containing models that were parsed correctly, four of them wrongly report errors. 2 different error messages were shown: "Type is undefined" and "Inline call is undefined". The first error occurs when a macro like `#define Foo bit` is defined, followed by a variable definition `Foo f;`. The variable definition is parsed correctly, since it accepts any identifier as its type to support user-defined types. Since `Foo` is not a user-defined type, but a macro definition, this identifier is not included in the list of user-defined types and therefore the message "Type is undefined" is shown. This could be improved by checking all macro definitions if they expand to a predefined type. If so, the macro identifier is included in the list of user-defined types. For the second wrongly reported error message, a similar problem arises. A call to `Bar(x)` will be parsed correctly, since it is recognized as an inline call. However, it could also be a call to a function-like macro `#define Bar(x) x+1`. Again this could be solved by adding the macro identifier to the list of inline definitions. Both solutions are assigned for future work.

5.2.2 Testing reference resolving and content completion

The other two semantic editor services that are implemented are reference resolving and content completion. Both can be tested with the integrated testing language in Spoofax and are discussed in the next two subsections.

Reference resolving

For reference resolving the same pairs of definitions and occurrences could be made as defined in previous section. Every occurrence that is clicked on should move the cursor to the definition, except for the inline call. Since this call is expanded during analysis, the

```
test goto[[
  init{
    [[lab]]: skip;
    goto [[lab]];
  }
]]resolve #2 to #1
```

Figure 5.1: Test case showing how reference resolving is tested

AST does not contain this call any more, and therefore no reference resolving could be performed on this inline call. The other pairs are tested using the Spoofax testing language. To test if the label in a `goto` statement correctly resolves to the label definition, the test case from Figure 5.1 is executed. In this test case, two placeholders can be found, indicated by the double squared brackets. The test condition that is specified states that the test case succeeds if the second placeholder resolves to the first one.

Ten different cases should be tested for reference resolving. The pairs *label* - `goto`, `proctype` - `run` and *user-defined type* - *declaration of user-defined type* are tested using the same structure as above. Furthermore, the seven different ways a variable could be used are included: `i`, `arr[2]`, `foo.i`, `foo.arr[2]`, `arr[2].i`, `foo.i.i`, and `foo.arr[2].i`. When running the tests, surprisingly one test case fails. The test case that makes sure that the identifier in a `run` statement resolves to a proctype declaration fails. When the same test is applied manually by control-clicking on the identifier, the cursor moves to the proctype declaration. Therefore this failed test case is considered as a false negative. The reason for this false negative is unknown and might be a Spoofax related issue.

Content completion

Content completion is tested in a similar way as reference resolving. The same test cases are used, but they are adapted to look like the one in Figure 5.2. The test condition now states that the identifier in the placeholder should complete to the string “lab” in this case. The test cases with user-defined variables are used to test the completion for the variables of a user-defined type, like:

```
typedef Foo{
  int i;
  int arr;
}
init{
  Foo f;
  f.
}
```

```
test goto[[
init{
lab: skip;

goto [[l]];
}
]]complete to "lab"
```

Figure 5.2: Test case showing how content completion is tested

When the dot is typed, a completion list containing `i` and arr is shown. All testcases succeed for content completion.

Summary

This chapter tested EpiSpin using approximately 1000 test cases. Half of them are complete models that are tested for parsing and for correct error reporting. The other half contains unit tests to test all grammar rules, all error messages, all reference resolving constructs and all content completion constructs. These unit tests are systematically composed by using the most simple example that covers the grammar rule or triggers the error message. Corner cases are identified for every rule or message which are also tested.

Chapter 6

Related Work

Model Checking [17] is a technique used for state space exploration of a model of a system to determine whether it meets a given specification. Several languages and tools exists to perform this task. For the *Promela* language [15], SPIN [16] is the native tool that can perform a simulation and verification of a model. Other tools exist that use SPIN or are based on SPIN and these tools will be discussed in the next sections. First the stand-alone editors will be discussed and in Section 6.2 the existing Eclipse plug-ins are shown. In Section 6.3 a comparison is made between EpiSpin and the editors that most closely relate to EpiSpin.

6.1 Existing stand-alone editors

First the stand-alone editors are discussed that use SPIN or are highly related to SPIN. These are *iSpin* and *Xspin*, JSPIN, ERIGONE and SPINJA.

6.1.1 Spin, iSpin and Xspin

SPIN is the command line tool that is used for syntax checking and performing simulations. With the 45 different flags that can be used for this command, an extensive amount of options can be specified for simulation runs. When a simulation is started, all executed statements are printed sequentially and an overview is given of the values of the global variables and the messages in channels.

SPIN is also used to generate a verifier in C, which can then be compiled by a C compiler. Several flags can be specified that optimize the output program for different search modes or storage modes. When the verifier is started, it prints the amount of states searched and memory used and when verification is finished, a summary is given which shows which properties were checked and if one of them failed. If so, a counter-example is provided.

Xspin and *iSpin* [14] both integrate a text editor with the SPIN model checker using the Tcl language with the Tk toolkit [28]. Since *iSpin* is a newer version of *Xspin*, with mainly graphical changes, only *iSpin* will be discussed here. Using Tcl/Tk, the editor and SPIN are tightly integrated with each other. When a simulation or verification is performed in *iSpin*, the result can be clicked on to highlight the corresponding line in the source code.

For simulation runs, three windows are provided that give the values of global variables, the statements executed and the contents of the channels. When a statement is selected in the second window, the other two windows are updated to give the information for that statement. Furthermore, a Message Sequence Chart (MSC) is drawn while the simulation runs to show which channel send and receive statements are executed, which messages are sent or received and in what order. Just like the window with the executed statements, the other windows are updated when the elements of the MSC are clicked.

Unfortunately, the editor itself is a very basic plain-text editor, in which even the line numbers are shown in the editor. When the file is saved, *iSpin* filters out these numbers, but if just a fragment of source code needs to be copied, the line numbers are copied as well. Furthermore, no single editor service is provided, which causes the source code to look chaotic for large programs.

6.1.2 jSpin

JSPIN [2] is a development environment for *Promela* created by Moti Ben-Ari. While *iSpin* and *Xspin* focus on professional users by providing an extensive list of options that can be specified for verification and simulation runs. JSPIN on the other hand is primarily designed for starting modelers, like students in a model checking course. JSPIN has a simple user interface in which *Promela* models can be edited and SPIN can be executed in its various modes with a single mouse click or keypress. The output can be customized by specifying which information should be printed. JSPIN is written in Java instead of Tcl/Tk (like *iSpin* and *Xspin*), because of Java's portability and wide use in computer science education. JSPIN does not process the source code of a *Promela* model itself, but calls SPIN just like *iSpin* and *Xspin* do. The result of a syntax check, verification or simulation run is propagated back to JSPIN and formatted so it matches the output format specified by the user.

6.1.3 Erigone

The same developer created the ERIGONE model checker [3], which is a simplified re-implementation of SPIN. It is implemented in Ada 2005 and is built for the same reason as JSPIN: to provide a simple tool for starting modelers. ERIGONE can be installed by opening one single file and contains a detailed documentation. Furthermore, the source code is well structured such that users can modify the algorithms used. ERIGONE supports a subset of the *Promela* language and is still in development to include more constructions from the *Promela* language. ERIGONE consists of three subsystems: a compiler, a model checker that implements the same algorithms as SPIN does and an LTL translator, all written in Ada.

6.1.4 SpinJa

Another model checker for *Promela* is SPINJA [7]. SPINJA is designed to behave similarly to SPIN, but is more extendable and reusable because it is written in Java and uses an object-oriented approach. SPINJA does not use SPIN but has its own parser and verification and

simulation methods. Like ERIGONE, SPINJA supports a large subset of the *Promela* language and allows you to perform verification of the models by checking different properties (absence of deadlock, assertions, liveness properties) by using different search modes. Also random, interactive and guided simulations of the model can be started. From the viewpoint of EpiSpin, the most interesting part of this plug-in is its parser. The rest of the SPINJA implementation covers the search algorithms, which is not relevant in the scope of this thesis. SPINJA uses an external tool *JavaCC* that generates a parser in Java by processing a grammar file. This grammar file is written by the SPINJA developers and specifies the *Promela* grammar in a *JavaCC* compatible format. Furthermore, they implemented half a dozen of semantic error checks in their grammar by specifying `MyParseException`s when certain conditions fail. JavaCC creates the Java code to handle these exceptions. The semantic errors seem rather random and it is unclear why only these are included. A *Promela* model is parsed when the compilation command is executed. When a syntax error is present in the model, parsing fails and a text is printed in the console explaining what token was found and what was expected.

6.2 Eclipse plug-ins

This section discusses two existing plug-ins from Rothmaier and Kovse.

6.2.1 Eclipse plug-in from Rothmaier

In 2005 Gerrit Rothmaier developed an Eclipse plug-in as part of his PhD research on how to use SPIN and Eclipse for optimized high-level modeling and analysis of computer network attack models [33]. Unfortunately, this plug-in was never distributed after his dissertation. The plug-in was made to integrate the cTLA2PC translation tool, which translates cTLA specifications to *Promela*, into Eclipse [34]. Therefore syntactic and semantic error checking is implemented for the cTLA language. The parser is based on the ANTLR [30] parser construction kit. Semantic analysis is done by traversing and transforming the syntax tree and symbol table. The plug-in provides editing, translation, simulation, debugging and verification of specifications. One of its key features is the ability to use the Eclipse debugging mechanism for simulation runs of the translated cTLA specifications. Breakpoints can be set in the editor and if that line is executed during simulation, the simulation will be stopped and the user can investigate the values of variables at that point and step through the rest of the simulation. Finally, the translated specifications can be verified from within Eclipse.

6.2.2 Eclipse plug-in from Kovse

The work that is the closest related to EpiSpin is the Eclipse plug-in that is made by Tim Kovse and his colleagues [23]. This plug-in contains a *Promela* editor with syntax highlighting, code folding, error markers and static content completion of keywords. The error markers are shown when the SPIN syntax checker is called. This plug-in is created by manually writing all Java classes that take care of the editor, the error marking, the preference pages and all other components of the plug-in. It has an implemented token scanner and

when a keyword is found, it is given a special font. These fonts can be customized by the user for eight different groups of keywords, which is not possible in EpiSpin. The different groups are: meta terms, declarators rule, control flow, basic statements, predefined, embedded C code, omissions, and comments.

Code folding is implemented for structures enclosed by brackets and for code between reserved pairs of keywords, such as `if-fi` and `do-od`. A Java class that handles code folding is called every time a change is made to the model. This class calculates the start and end points of the foldable structures and interacts with the internal Java interface that handles the code folding internally.

Error markers are shown when the SPIN syntax checker is called. The result of the syntax check is returned to the plug-in and when an error is found, the corresponding message and line number are extracted and communicated to the internal Eclipse class that creates a new marker. The error message is also shown in the Problems view in Eclipse.

When content completion is requested, a list of all keywords that starts with the typed letter(s) is shown, regardless the position of the cursor. This is completely handled by a class that implements the `IContentAssistProcessor` interface.

Using preference pages it is possible to specify verification and simulation options. The same verification options can be specified as in the *Xspin* editor. The settings can be exported and imported for later use. For simulation, only two options can be customized: the seed value and the initial steps skipped. Next, in the editor a verification run or a guided or random simulation can be started. Similar to EpiSpin, the verifier or SPIN is called and the output is shown in the console window.

This plug-in also contains integrated support for the *st2msc* tool, which is a conversion tool made by the same developers. It converts a SPIN trail into a message sequence chart. In a special window in the plug-in, the input and output file can be specified and the chart is generated.

6.3 Comparison of EpiSpin with related tools

This section will summarize the above sections and compare EpiSpin with its nearest competitors. EpiSpin's most important design target was to provide an Eclipse plug-in with state-of-the-art editor services and an interface to perform SPIN simulation and verification runs. The tools that relate the most with EpiSpin are *iSpin* and the Eclipse plug-in from Kovse. Too few is known about the other Eclipse plug-in from Rothmaier to make a fair comparison. SPINJA and ERIGONE are re-implementations of SPIN and focus more on the implementation of the search algorithms, than on the implementation of a sophisticated editor. JSPIN can be seen as a simplified version of *iSpin*, including only the basic options for performing simulation and verification runs.

EpiSpin, *iSpin* and Kovse's plug-in can be compared on two main points: the editor and the simulation and verification environment. The editor of *iSpin* is very basic and can hardly be compared to the two Eclipse plug-ins, which provide much better editors to code in. Having only syntax highlighting would create a much more clear overview of the code. The two Eclipse plug-ins both include several editor services, as can be seen in the top half

Table 6.1: Overview of the features of the existing plug-in by Kovse and EpiSpin

	Kovse	EpiSpin	iSpin
Syntax highlighting	✓	✓	
Customization of syntax highlighting	✓		
Code folding	✓	✓	
Outline view		✓	
Content completion			
Syntactic	✓	✓	
Semantic		✓	
Error markers	On demand	Instantly	On demand (only textual)
Reference resolving		✓	
Verification	✓	✓	✓
Save verification settings	✓	✓	
Simulation			
Guided & Random	✓	✓	✓
Interactive		✓	✓
Save verification results			✓
Extensive browsing of the output			✓
Graphical tools	st2msc	SCA	Automata view

of Table 6.1. Syntax highlighting and code folding are similar in both plug-ins, but content completion is much more advanced in EpiSpin. It takes into account the position of the cursor and can complete to entire structures instead of only to keywords. Also, identifiers can be completed as indicated by the *semantic content completion* field in the table.

The only disadvantage of EpiSpin is its limited amount of support for models that contain macros. Although the mostly used macro constructions are included as explained in Section 3.1, it is still possible that models that contain exceptional use of macros are not parsed correctly.

Furthermore, EpiSpin has automatic bracket insertion and automatic indentation. These two features may not seem very magnificent, but they certainly increase the coding speed. EpiSpin also contains an outline view which is very useful when modeling large programs and reference resolving to quickly find the declaration of a identifier. But probably the biggest improvement of EpiSpin over the existing plug-in is the instant feedback on syntactic and semantic errors. Writing source code is an accurate process in every programming language and a (syntax) error is easily made. The sooner this error is pointed out to the developer, the sooner the erroneous code can be repaired. By finding all this editor services which are not included in Kovse's plug-in and in *iSpin*, it is safe to say that the editor of EpiSpin is better than the existing editors.

The second part of EpiSpin is the interface to perform simulation and verification runs. Like *iSpin*, EpiSpin opens these option screens in separate tabs of the editor, while Kovse's plug-in uses the Eclipse preference window in which a new page for SPIN options is created.

This preference window is a new window that 'hangs' above the editor, which means that the source code can not be edited while the preference page is open. Another difference is that in EpiSpin and *iSpin* a simulation or verification run can be started in the same window as where the options are specified, while in Kovse's plug-in such a run is started from the editor, which means the preference window with the options should be closed first. Another big issue in the plug-in from Kovse is that the complete editor freezes as long as a verification run is not finished. In EpiSpin and *iSpin* intermediate information about the explored state space and memory usage is given.

When the specifications of the three editors are investigated, it can be seen that the most important features are implemented in all of them: verification, guided and random simulation. Unfortunately, interactive simulation is not supported by Kovse's plug-in. An interesting feature that is implemented in the two Eclipse plug-ins, but is not shared with *iSpin* is the ability to save the verification settings to make it easier to perform the same verification at a later time again. On the other hand, *iSpin* has the ability to save the outputs of a verification, which is not implemented in the Eclipse plug-ins. For EpiSpin, this is assigned for future work. Finally, *iSpin* shows different information windows during simulation which are all connected which each other, while EpiSpin and Kovse's plug-in only show the textual output.

Comparing the three editors on their performance during simulation and verification, it can be said that EpiSpin works just a little smoother than the existing Eclipse plug-in. Comparing EpiSpin with *iSpin*, it makes no difference whether a verification run is done in EpiSpin or *iSpin*. When the output of verification runs need to be saved, *iSpin* might be more appropriate and when the settings need to be saved, EpiSpin is more valuable. For simulation runs, most *Promela* models could be perfectly simulated in EpiSpin. Only when a real-time graph of the channels used or a thorough inspection of the global variables is needed, it might be better to perform the simulation run in *iSpin*, which can be opened from within EpiSpin.

Taking both the editor and simulation and verification interface into account, it can be said that EpiSpin increases the coding speed and the maintainability of a *Promela* model by including all these sophisticated editor services. Simulation and verification is just as easy and fast as in *iSpin*, so with the more advanced editor, EpiSpin is more complete and a better tool compared to Kovse's plug-in and *iSpin*.

Chapter 7

Conclusions and Future Work

This chapter will conclude the work done that is done in previous chapters. In the first section a summary and conclusion is given about EpiSpin A discussion of the extensibility, portability and maintainability of EpiSpin will be discussed in Section 7.2, followed by a reflection on some design choices in Section 7.3. This chapter ends with some recommendations for future work in Section 7.4 and with a small evaluation of the complete project in Section 7.5.

7.1 Conclusion

This thesis presented the design and implementation of EpiSpin, an Eclipse plug-in for editing and verifying *Promela* models. Its largest improvement over current tools is the editor including instant error checking and various syntactic and semantic editor services. EpiSpin is created with Spoofax, a language workbench for agile development of languages which provides them with state-of-the-art editor services. Spoofax takes care of the language independent code that is needed to build the framework of a new Eclipse plug-in so language developers can focus on the implementation of the language specific features.

The grammar rules of EpiSpin are specified in SDF, the syntax definition formalism for defining context-free grammars. EpiSpin accepts the same language as the latest SPIN release does. By using rule coverage and testing a batch of 260 programs, 3 syntax errors were found and fixed. All macro-free models are parsed successfully now. Due to the context-sensitive nature of the macro syntax it is not possible to include the full macro grammar with SDF rules. For that, the complete grammar should be written in a context-free form, which is not possible. Therefore, only a limited amount of macro constructions are supported in EpiSpin. After performing the research described in Section 3.1 it is decided to implement the following macro constructions in the EpiSpin grammar:

- `#include` statements as freestanding modules or statements
- `#ifdef`, `#ifndef`, `#if`, `#else`, `#endif` statements as freestanding modules or statements,
- `#define` statements as freestanding modules or statements,

- Object-like macro calls are allowed at all places where a constant can be placed (like in array and channel definitions and other calls to arrays),
- Function-like macro calls are allowed as freestanding modules or statements,

When opening a set of 30 macro containing models in EpiSpin, 20 of them were parsed successfully. The other models were not parsed because they included macro constructions that are not supported by EpiSpin.

The second main component of EpiSpin is the interface to start a simulation or verification run. The same simulation and verification properties can be specified as in *iSpin* and the output is shown in the EpiSpin console.

The last main component that is included in EpiSpin is the Static Communication Analyzer (SCA). The SCA shows the communication structure of a *Promela* model by showing the relations between processes, global variables and channels. This tool generates *dot* code that can be compiled to a graph. This *dot* code is generated by traversing the AST and transforming each constructor that needs to be shown in the graph into DOT code. This code should be manually compiled to a graph to see the communication structure of the *Promela* model.

EpiSpin supports more editor services than the Eclipse plug-in from Kovse. Besides syntax highlighting, code folding and syntactic content completion as is included in the plug-in from Kovse, EpiSpin provides instant feedback on syntactic and semantic errors, an outline view, reference resolving and semantic content completion. Also the EpiSpin simulation interface is much more detailed than the one from Kovse. It includes interactive simulation, which is not possible by Kovse's plug-in. Even more important is that the integration of the option windows is much better in EpiSpin because information is given about the state space and memory during verification runs, while the plug-in from Kovse freezes as long as the verification runs. Spoofax includes more editor services than the plug-in from Kovse and than *iSpin* and the features that are also implemented in one of these existing editors, perform just as well in EpiSpin. Therefore EpiSpin could be used as an improvement over *iSpin* and Kovse's plug-in.

7.2 Discussion

Extensibility

EpiSpin is created using the Spoofax language workbench instead of the traditional way of writing all features in Java code directly. The total number of lines of code that need to be written by the EpiSpin developer is much smaller than if EpiSpin was developed in Java directly. This is the result of Spoofax that takes care of all language independent code such that the remaining language specific code can be written as concise as possible. This results in the fact that EpiSpin is easily extensible. When a new language construct is added, its grammar rules should only be added to the list of SDF grammar rules to be parsed by EpiSpin. To add it to the outline view or implement code folding for that structure, only one line has to be added. Then for semantic editor features, on average 4 or 5 lines should be written for the analysis and the same amount per editor feature. This means that a new

language construct could be implemented with all editor services using less than 30 lines of code.

Portability

Regarding the portability of EpiSpin, four other software programs are needed on the machine to run EpiSpin perfectly. The most important one is Eclipse obviously, which is necessary to install EpiSpin in. Having only Eclipse installed, but not the later mentioned tools, only the editor and all its editor features can be used. When a simulation or verification run needs to be started, the SPIN executable should be available in the default class path of the system. For verification runs, also the *gcc* compiler needs to be available. Both the SPIN and *gcc* commands are hard coded in the EpiSpin source code. This could be improved by allowing users to specify these paths. The last component that is used by EpiSpin is *dot*, which comes with the *graphviz* package [10]. *Dot* is needed to compile the generated dot code from the SCA (which could be generated without *dot* as well) into a graph. Although EpiSpin needs four components to run perfectly, *gcc* and SPIN are expected to be installed on most machines already since EpiSpin is developed for SPIN users. Eclipse is hard to avoid when creating an Eclipse plug-in, so *dot* is the component that most EpiSpin users need to download additionally if they want to use the SCA.

This means EpiSpin can easily be ported to a Mac OSX computer or any other Unix based system. For Windows PC's, *gcc* and SPIN only run in an emulated Linux-like environment, such as Cygwin [32]. To use EpiSpin with all its features, it should be used from within the Cygwin environment. This is not sufficiently tested yet and therefore not recommended to use.

Maintainability

Maintaining EpiSpin is easy for the lead developer, moderately difficult for developers that have a good understanding of Spoofax and its used languages and hard for other developers. Especially the Stratego language has a steep learning curve, but also the formal ideas behind analysis and transformation of the AST takes some time to fully understand. Tricks can be applied to improve the support for some editor services, such as allowing the `run` statement in the grammar without an identifier or parenthesis to improve content completion for that structure. An error message is then implemented to raise a syntax error when the `run` statement is not completed at all. These kind of tricks will be recognized by experienced Spoofax developers, but may seem random to other developers. By improving the comments in the SDF and Stratego code, the maintainability could be improved, but inexperienced Spoofax developers should first develop a thorough understanding of Spoofax.

7.3 Reflection

Looking back at the design of EpiSpin, some design choices are reconsidered in this section. One of the most important questions to ask is: was it a good choice to create EpiSpin using the Spoofax language workbench? As described above, the learning curve of Stratego is

steep. Although a manual describing most of the basic features of the Stratego language is available, it is still hard to completely understand how the language is put together. Spoofax self is in constant development. Spoofax has a stable version which is almost a year old and this means that for using the testing language for example, the unstable version should be used. The unstable version comes with minor bugs, but can also contain major changes. Once, for example, a change was made in the way position information for terms was stored internally. This resulted in the fact that certain editor messages in EpiSpin started to show up at places where they first did not (and should not). A point of recommendation to the Spoofax developers would be to communicate these kind of changes better, and not only in the issue tracker and in the SVN log. Especially since language developers that use Spoofax depend on the unstable version to work with the newest features. After all, it was a good choice to use Spoofax. Spoofax code is concise and clearly divided in different files in which it is easy to find certain code fragment at a later time (see Appendix F for an overview of all files and folders in Spoofax). Furthermore, the task of creating a scanner and parser is totally taken over by Spoofax, which creates a scannerless generalized LR parser based on the SDF grammar. Finally, the meta-editor that is dynamically loaded within the same Eclipse instance contributes to an agile development environment, which is pleasant to work with.

Something that could have been done differently, is the inclusion of separators in the AST. Now, all most separators that end a statement are shown in the AST (see Figure 4.9 for example). This AST can not be viewed by end users, but is used often during the development of EpiSpin to check if a certain transformation is applied successfully. By removing all separators in the desugaring phase, some constructors will have less terms and the AST will look more clear and the analysis rules more compact.

Another point of discussion that has arisen during the design of EpiSpin is whether the current implementation of the option screen could be improved. The main issue is that every *Promela* file in the Eclipse workspace can have its own file with the specified options. When this file is opened in Eclipse, not the filename is shown, but a predefined name that has been chosen during development. When multiple of these option files are opened, it gets unclear very easily which option file belongs to which *Promela* file. No solution has been found yet to this problem and therefore the question was asked if this way of implementing the option screen was justified. I am convinced this is the best solution. This way, the option screen integrates nicely with the rest of the Eclipse windows and it is able to save the options automatically. By using the split-screen opportunity of Eclipse it is possible to have the source code and the option screen in one Eclipse window. When multiple option files are opened at the same time, it might be more convenient to close the ones that are not used at the moment. In general, only a few different *Promela* files are investigated at the same time so this should not lead to big problems.

7.4 Future work

In this section recommendations for future work are given. Most of these issues are already discussed in the prior chapters of this thesis.

- The use of macro constructions could be improved in EpiSpin. It is not possible to include support for all structures, because macro calls can be placed everywhere. Still, more constructions could be included, like the one where a macro definition defines an existing type. Also, it could be investigated if not more macro calls could be transformed in their macro definitions so they are included during the analysis of the AST, as is only done with macros representing constants now.

  In general, more research could be done on integrating macros with context-free grammars. For Spoofax, it could be investigated if it is possible to implement a pre-parsing phase, in which the source code is expanded by the C preprocessor. By keeping track of the position information, the macro containing models could be edited while the macro-free version is used to build the AST.

- For the simulation and verification option files a small label could be added that shows the verifier or simulator invocation string. Now this string is shown in the console, before the output is printed. This string disappears easily when the results are printed, due to the limited space in the console. Although it could be seen by scrolling up again, it would be much easier if this invocation string is shown in a fixed position.

- Creating graphs with the SCA may be better integrated with EpiSpin. Now it is necessary to manually compile the *dot* files into a graph and to view the graph in a separate program. Eclipse plug-ins exist for viewing an image file in Eclipse or for displaying graphs using the *dot* code, but trying them in combination with EpiSpin did not lead to satisfying results. Semi-complex graphs (using colors or different shapes) were not displayed correctly in these plug-ins.

- The last EpiSpin related recommendation for future work is to make EpiSpin compatible for Windows. It currently only works on Unix-based systems, and is tested on Linux and Mac OSX, but does not have support for Windows yet. Since SPIN could be used on Windows, EpiSpin should as well. Furthermore, the SPIN and *gcc* commands are hard coded in the EpiSpin source code now and this could be improved by providing option fields in which a path to the executables could be specified.

- For Spoofax, an area of future work is to improve the definition for the outline view. It is not possible now to include keywords in the outline view, where this could be wishful in some cases, like for the `init` keyword. Furthermore, the selection of the candidate sorts for content completion could be improved, since some structures might appear in the completion list while they are not expected at the cursors' position.

7.5 Evaluation

The hardest part of designing EpiSpin was how to figure out how macros would be implemented. Both deciding which constructions would be implemented and figuring out how to

do that exactly was a time consuming part of my work. If time permitted, more constructions could have been added, like using a macro definition as a type, or allowing conditional proctype declarations like

```
#if condition
proctype A(int i)
#else
proctype A(int i,j)
#end
{ body }
```

Working with Spoofax was difficult in the beginning of this project as well. Learning the concepts of desugaring and renaming were all new to me and took some time to get completely familiar with, even while I did a course already in which I used Spoofax. The constructions I needed for EpiSpin were much more complicated and investigating some existing projects helped me a lot.

The last difficulty of this project relates to my time schedule. I started to work on EpiSpin in December 2010, and the first of April was the submission deadline for the SPIN workshop. Since we really wanted to publish this work over there I had to make some design choices which were not fully justified at the moment of writing the paper. This had no influence on writing this thesis since I completed the research after the submission deadline and changed some minor design choices which turned out to be less efficient than first expected. So this thesis reflects the design choices that are made correctly.

Bibliography

[1] Tcl Developer Xchange. http://www.tcl.tk/about/features.html.

[2] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer, first edition, 2008.

[3] Mordechai Ben-Ari. Tool presentation: Teaching concurrency and model checking. In Corina Pasareanu, editor, *Model Checking Software*, volume 5578 of *Lecture Notes in Computer Science*, pages 6–11. Springer Berlin / Heidelberg, 2009.

[4] M. G. J.van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[5] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52 – 70, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[6] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundam. Inf.*, 69:123–178, July 2005.

[7] Marc de Jonge and Theo Ruys. The SpinJa model checker. In Jaco van de Pol and Michael Weber, editors, *Model Checking Software*, volume 6349 of *Lecture Notes in Computer Science*, pages 124–128. Springer Berlin / Heidelberg, 2010.

[8] Eclipsepedia. Jface. http://wiki.eclipse.org/JFace.

[9] The Eclipse Foundation. Eclipse - The Eclipse foundation open source community website. http://www.eclipse.org.

[10] Graphviz. The DOT language - graph visualization software. http://www.graphviz.org/content/dot-language.

[11] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Not.*, 24:43–75, November 1989.

[12] Mark Hennessy and James F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 104–113, New York, NY, USA, 2005. ACM.

[13] G. J. Holzmann. Promela language reference. `http://www.spinroot.com/spin/Man/promela.html`.

[14] G. J. Holzmann. Spin - formal verification. `http://spinroot.com/spin/Man/GettingStarted.html`.

[15] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[16] G.J. Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279 –295, may 1997.

[17] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, October 2009.

[18] S.C. Johnson. *YACC - Yet Another Compiler-Compiler*. Murray Hill, Bell Laboratories, 1975.

[19] Pim Kars. The application of Promela and Spin in the BOS project. 1996.

[20] Lennart C. L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In Gary T. Leavens, editor, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, volume 44 of *ACM SIGPLAN Notices*, pages 445–464, New York, NY, USA, October 2009. ACM Press.

[21] Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. Integrated language definition testing: Enabling test-driven language development. In Kathleen Fisher, editor, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011)*, Portland, Oregon, USA, 2011. ACM.

[22] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, pages 444–463, 2010.

[23] Tim Kovše, Botjan Vlaovi, Aleksander Vree, and Zmago Brezonik. Eclipse plug-in for Spin and st2msc tools-tool presentation. In Corina Pasareanu, editor, *Model Checking Software*, volume 5578 of *Lecture Notes in Computer Science*, pages 143–147. Springer Berlin / Heidelberg, 2009.

[24] Ralf Lämmel. Grammar Testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.

[25] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. `http://dinosaur.compilertools.net/lex/index.html`.

[26] Alberto Lluch. Promela database. `http://www.albertolluch.com/research/promelamodels`.

[27] Oracle Corporation. Netbeans IDE. http://netbeans.org/community/releases/70/.

[28] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, Inc., 1994. (ISBN 0-201-63337-X).

[29] Yoann Padioleau. Parsing C/C++ code without pre-processing. In Oege de Moor and Michael Schwartzbach, editors, *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 109–125. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-00722-4_9.

[30] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[31] Radek Pelánek. Beem: benchmarks for explicit model checkers. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 263–267, Berlin, Heidelberg, 2007. Springer-Verlag.

[32] J. Racine. The Cygwin tools: a GNU toolkit for Windows. *Journal of Applied Econometrics*, 15(3):331–341, 2000.

[33] G. Rothmaier, T. Kneiphoff, and H. Krumm. Using SPIN and Eclipse for optimized high-level modeling and analysis of computer network attack models. In *Model Checking Software*, volume 3639 of *LNCS*, pages 236–250. Springer, 2005.

[34] Gerrit Rothmaier. *Integrated Formal Modeling and Automated Analysis of Computer Network Attacks*. PhD thesis, Universität Dortmund, 2005.

[35] P. Taverne and C. Pronk. RAFFS: Model Checking a Robust Abstract Flash File Store. In *Formal Methods and Software Engineering; 11th Intn'l Conf. on Formal Engineering Models, ICFEM2009*, volume 5885 of *LNCS*, pages 226–245, 2009.

[36] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[37] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *SIGPLAN Not.*, 34:13–26, September 1998.

Appendix A

Promela grammar

This appendix shows the *Promela* grammar as found on the SPIN website [13].

```
spec    : module [ module ] *

module  : proctype      /* proctype declaration */
        | init          /* init process       */
        | never         /* never claim        */
        | trace         /* event trace        */
        | utype         /* user defined types */
        | mtype         /* mtype declaration  */
        | decl_lst      /* global vars, chans */

proctype: [ active ] PROCTYPE name '(' [ decl_lst ]')'
          [ priority ] [ enabler ] '{' sequence '}'

init    : INIT [ priority ] '{' sequence '}'

never   : NEVER '{' sequence '}'

trace   : TRACE '{' sequence '}'

utype   : TYPEDEF name '{' decl_lst '}'

mtype   : MTYPE [ '=' ] '{' name [ ',' name ] * '}'

decl_lst: one_decl [ ';' one_decl ] *

one_decl: [ visible ] typename  ivar [',' ivar ] *

typename: BIT | BOOL | BYTE | SHORT | INT | MTYPE | CHAN
        | uname /* user defined type names (see utype) */

active  : ACTIVE [ '[' const ']' ]      /* instantiation */
```

```
priority: PRIORITY const        /* simulation priority */

enabler : PROVIDED '(' expr ')' /* execution constraint */

visible : HIDDEN | SHOW

sequence: step [ ';' step ] *

step    : stmnt [ UNLESS stmnt ]
        | decl_lst
        | XR varref [',' varref ] *
        | XS varref [',' varref ] *

ivar    : name [ '[' const ']' ] [ '=' any_expr | '=' ch_init ]

ch_init : '[' const ']' OF '{' typename [ ',' typename ] * '}'

varref  : name [ '[' any_expr ']' ] [ '.' varref ]

send    : varref '!' send_args          /* normal fifo send */
        | varref '!' '!' send_args      /* sorted send */

receive : varref '?' recv_args          /* normal receive */
        | varref '?' '?' recv_args      /* random receive */
        | varref '?' '<' recv_args '>'  /*poll with side-effect*/
        | varref '?' '?' '<' recv_args '>'      /* ditto */

poll    : varref '?' '[' recv_args ']'  /* poll without side-
                                                effect */
        | varref '?' '?' '[' recv_args ']'      /* ditto */

send_args: arg_lst | any_expr '(' arg_lst ')'

arg_lst  : any_expr [ ',' any_expr ] *

recv_args: recv_arg [ ',' recv_arg ] * | recv_arg '('recv_args')'

recv_arg : varref | EVAL '(' varref ')' | [ '-' ] const

assign  : varref '=' any_expr   /* standard assignment */
        | varref '+' '+'        /* increment */
        | varref '-' '-'        /* decrement */

stmnt   : IF options FI         /* selection */
        | DO options OD         /* iteration */
        | FOR '(' range ')' '{' sequence '}'     /* iteration */
        | ATOMIC '{' sequence '}'       /* atomic sequence */
        | D_STEP '{' sequence '}'      /* deterministic atomic */
        | SELECT '(' range ')'      /* non-deterministic value
                                                selection */
        | '{' sequence '}'      /* normal sequence */
```

```
        | send
        | receive
        | assign
        | ELSE                  /* used inside options */
        | BREAK                 /* used inside iterations */
        | GOTO name
        | name ':' stmnt        /* labeled statement */
        | PRINT '(' string [ ',' arg_lst ] ')'
        | ASSERT expr
        | expr                  /* condition */
        | c_code '{' ... '}'    /* embedded C code */
        | c_expr '{' ... '}'
        | c_decl '{' ... '}'
        | c_track '{' ... '}'
        | c_state '{' ... '}'

range   : varref ':' expr '..' expr
        | varref IN varref

options : ':' ':' sequence [ ':' ':' sequence ] *

andor   : '&' '&' | '|' '|'

binarop : '+' | '-' | '*' | '/' | '%' | '&' | '^' | '|'
        | '>' | '<' | '>' '=' | '<' '=' | '=' '=' | '!' '='
        | '<' '<' | '>' '>' | andor

unarop  : '~' | '-' | '!'

any_expr: '(' any_expr ')'
        | any_expr binarop any_expr
        | unarop any_expr
        | '(' any_expr '-' '>' any_expr ':' any_expr ')'
        | LEN '(' varref ')'    /* nr of messages in chan */
        | poll
        | varref
        | const
        | TIMEOUT
        | NP_                   /* non-progress system state */
        | ENABLED '(' any_expr ')'          /* refers to a pid */
        | PC_VALUE '(' any_expr ')'         /* refers to a pid */
        | name '[' any_expr ']' '@' name    /* refers to a pid */
        | RUN name '(' [ arg_lst ] ')' [ priority ]

expr    : any_expr
        | '(' expr ')'
        | expr andor expr
        | chanpoll '(' varref ')'       /* may not be negated */

chanpoll: FULL | EMPTY | NFULL | NEMPTY
```

```
string  : '"' [ any_ascii_char ] * '"'

uname   : name

name    : alpha [ alpha | number ] *

const   : TRUE | FALSE | SKIP | number [ number ] *

alpha   : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' |
          'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' |
          's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' |
          'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' |
          'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
          'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '_'

number  : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
          '9'
```

Appendix B

EpiSpin grammar

This appendix contains the SDF rules that form the EpiSpin grammar. On the left-hand side the terminals and non-terminals are shown that form the rule. On the right-hand side the non-terminal that is matched is indicated in bold, together with the constructor name that is used in the abstract syntax tree. The notations are:

- Terminals are shown in double quotes
- Non-terminals (also called sorts in SDF) are shown as words starting with a capital
- Lexical syntax is indicated by fully capitalized words
- Optional sorts are indicated with a '?'
- Non empty lists are indicated with '+'
- Possibly empty lists are indicated with '*'
- A sort and a terminal between curly brackets followed by a list indicator represents a list of that sorts separated by that terminal

Lexical syntax

```
[a-zA-Z\_][a-zA-Z0-9\_]*          -> ID
[0-9]+                            -> CONST
CHAR                              -> CONST
"true"                            -> CONST
"false"                           -> CONST
"skip"                            -> CONST
","                               -> COMMA
[\#]                              -> DIRECTIVE

"'" CharChar "'"                  -> CHAR
"\\n"                             -> CharChar
"\\t"                             -> CharChar
" "                               -> CharChar
~[\'\n]                           -> CharChar
```

```
"\"" StringChar* "\""                -> STRING
~[\"\n]                              -> StringChar
"\\\""                               -> StringChar
BackSlashChar                        -> StringChar
"\\"                                 -> BackSlashChar

[\n]                                 -> ENDOL
[\*]                                 -> CommentChar

[\ \t\n\r]                           -> LAYOUT
"/*" (~[\*] | CommentChar)* "*/"     -> LAYOUT
"//" ~[\n\r]* ([\n\r] | EOF)         -> LAYOUT

DIRECTIVE ~[\n\r]* ([\n\r] | EOF)    -> LAYOUT

"c_code" LAYOUT* "{" InnerCode* "}"  -> CCODE
"c_expr" LAYOUT* "{" InnerCode* "}"  -> CCODE
"c_decl" LAYOUT* "{" InnerCode* "}"  -> CCODE
"c_track" LAYOUT* "{" InnerCode* "}" -> CCODE
"c_state" LAYOUT* "{" InnerCode* "}" -> CCODE

~[\{\}]                              -> InnerCode
Opening                              -> InnerCode
Closing                              -> InnerCode
CCODE                                -> InnerCode
[\{]                                 -> Opening
[\}]                                 -> Closing
```

Lexical restrictions

```
CommentChar       -/- [\/]
CONST             -/- [0-9]
ID                -/- [a-zA-Z0-9\_]
MACRO             -/- ~[\n]

EOF               -/- ~[]
BackSlashChar     -/- [\"]
```

Context-free syntax

```
Module+  -> Start {cons("Modules")}

Active? "proctype" ID "(" ")" Priority? Enabler? "{"Sequence "}"
                                -> Module {cons("Proctype")}
Active? "proctype" ID "(" One-decl (Sep One-decl)* ")" Priority?
       Enabler? "{"Sequence "}" -> Module {cons("ProctypeParam")}
Active? "D_proctype" ID "(" ")" Priority? Enabler? "{"Sequence"}"
                                -> Module {cons("Proctype")}
Active? "D_proctype" ID "(" One-decl (Sep One-decl)* ")"Priority?
       Enabler? "{"Sequence "}" -> Module {cons("ProctypeParam")}
```

```
"init" Priority? "{" Sequence "}"    -> Module{cons("Init")}
"never" ID? "{" Sequence "}"         -> Module{cons("Never")}
"trace" "{" Sequence "}"             -> Module{cons("Trace")}
"notrace" "{" Sequence "}"           -> Module{cons("Notrace")}
"typedef" ID "{" One-decl (Sep One-decl)* Sep? "}"
                                     -> Module{cons("Typedef")}
"mtype" "{" {ID ","}+ COMMA? "}"     -> Module{cons("Mtype")}
"mtype" "=" "{" {ID ","}+ COMMA?"}"  -> Module{cons("MtypeEq")}
"ltl" ID? "{" LTLFullExpr Sep? "}"   -> Module{cons("Ltl")}
Sep                                  -> Module{cons("Seperator")}
One-decl                             -> Module{cons("GlobalDecl")}
"inline" ID "(" {Varref "," }* ")" "{" Sequence "}"
                                     -> Module {cons("Inline")}

"active"                        -> Active {cons("Active")}
"active" '[' CONST ']'          -> Active {cons("ActiveMult")}
"active" '[' ID ']'             -> Active {cons("ActiveMultMacro")}

"priority" CONST                -> Priority {cons("Priority")}
"priority" ID                   -> Priority {cons("PriorityMacro")}

"provided" "(" FullExpr")"      -> Enabler {cons("Enabler")}

Visible? Type {Ivar ","}+               -> One-decl {cons("Decl")}
Visible? TypeUnsigned {IvarUnsigned ","}+ ->
                                        One-decl {cons("Decl")}

"hidden"                            -> Visible {cons("Hidden")}
"show"                              -> Visible {cons("Show")}
"local"                             -> Visible {cons("Local")}

ID  VarInit?              ->Ivar {cons("Var")}
ID "[" CONST "]" VarInit?->Ivar {cons("IvarArrayStandard")}
ID "[" ID "]" VarInit?    ->Ivar {cons("IvarArrayStandardMacro")}
ID ":" CONST VarInit?     ->IvarUnsigned{cons("VarUnsigned")}
ID ":" ID VarInit?        ->IvarUnsigned{cons("VarUnsignedMacro")}

"=" AnyExpr                     -> VarInit {cons("VarStandard")}
"=" "[" CONST "]" "of" "{" {Typename ","}+ "}"
                                -> VarInit {cons("ChanInit")}
"=" "[" ID "]" "of" "{" {Typename ","}+ "}"
                                -> VarInit {cons("ChanInitMacro")}

(Step)+ Stepfs                -> Sequence {cons("Steps")}
Stepfs                        -> Sequence {cons("Steps")}

Stmnt   Sep*                  -> Stepfs {cons("Stmnt")}
Stmntz  Sep*                  -> Stepfs {cons("Stmntz")}
One-decl Sep*                 -> Stepfs {cons("Declaration")}
Stmnt "unless" Stmnt  Sep*    -> Stepfs {cons("Unless")}
Stmntz "unless" Stmnt Sep*    -> Stepfs {cons("Unless")}
```

```
Stmnt "unless" Stmntz Sep*    -> Stepfs {cons("Unless")}
Stmntz "unless" Stmntz  Sep*  -> Stepfs {cons("Unless")}
"xr" {Varref ","}+    Sep*    -> Stepfs {cons("XR")}
"xs" {Varref ","}+    Sep*    -> Stepfs {cons("XS")}

Stmnt Sep+                    -> Step {cons("Stmnt")}
Stmntz  Sep*                  -> Step {cons("Stmntz")}
One-decl Sep+                 -> Step {cons("Declaration")}
Stmnt "unless" Stmnt  Sep+    -> Step {cons("Unless")}
Stmntz "unless" Stmnt Sep+    -> Step {cons("Unless")}
Stmnt "unless" Stmntz Sep*    -> Step {cons("Unless")}
Stmntz "unless" Stmntz  Sep*  -> Step {cons("Unless")}
"xr" {Varref ","}+    Sep+    -> Step {cons("XR")}
"xs" {Varref ","}+    Sep+    -> Step {cons("XS")}

ID                          -> Varref {cons("Name")}
ID '[' AnyExpr ']'          -> Varref {cons("ArrayElement")}
Uref '.' ID '[' AnyExpr ']'-> Varref{cons("ArrayElementObject")}
Uref '.' ID                 -> Varref {cons("ObjectElement")}

ID                          -> Uref {cons("UTName")}
ID '[' AnyExpr ']'          -> Uref {cons("UTArrayElement")}
Uref '.' ID '[' AnyExpr ']'-> Uref {cons("UTArrayElementObject")}
Uref '.' ID                 -> Uref {cons("UTObjectElement")}

ID "(" {AnyExpr ","}* ")"   -> FunctionCall{cons("FunctionCall")}

Varref "!" SendArgs         -> Send {cons("StandardSend")}
Varref "!" "!" SendArgs     -> Send {cons("SortedSend")}

Varref "?" RecvArgs         -> Receive {cons("StandardReceive")}
Varref "??" RecvArgs        -> Receive {cons("RandomReceive")}
Varref "?" "<" RecvArgs ">"-> Receive {cons("SpecialReceive")}
Varref "??" "<" RecvArgs ">"
                        -> Receive {cons("SpecialRandomReceive")}

Varref "?" "[" RecvArgs"]" -> Poll {cons("StandardPoll")}
Varref "??" "[" RecvArgs"]"-> Poll {cons("RandomPoll")}

{AnyExpr","}+               -> SendArgs {cons("StandardSendArgs")}
AnyExpr "(" {AnyExpr ","}+ ")"
                            -> SendArgs {cons("SpecialSendArgs")}
{RecvArg ","}+              -> RecvArgs {cons("StandardRecvArgs")}
RecvArg? "(" RecvArgs ")"   -> RecvArgs {cons("SpecialRecvArgs")}

Varref                  -> RecvArg {cons("VarrefRecv"), avoid}
"eval" "(" Varref ")" -> RecvArg {cons("Eval")}
"-" CONST             -> RecvArg {cons("NegativeConstRecv")}
"-" ID                -> RecvArg {cons("NegativeConstRecvMacro")}
"_"                   -> RecvArg {cons("RecvNC")}
CONST                 -> RecvArg {cons("ConstRecv")}
```

```
ID                      -> RecvArg {cons("ConstRecvMacro")}

Varref "=" FullExpr     -> Assign {cons("Assignment")}
Varref "++"             -> Assign {cons("Increment")}
Varref "--"             -> Assign {cons("Decrement")}
"_" "=" FullExpr        -> Assign {cons("AssignmentNC")}

"if" Options "fi"             -> Stmnt {cons("If")}
"do" Options "od"             -> Stmnt {cons("Do")}
"for" "(" Range ")" "{" Sequence "}"-> Stmntz {cons("For")}
"atomic" "{" Sequence "}"     -> Stmntz {cons("Atomic")}
"d_step" "{" Sequence "}"     -> Stmntz {cons("Dstep")}
"select" "(" Range ")"        -> Stmnt {cons("Range")}
"{" Sequence "}"              -> Stmntz {cons("Sequence")}
FunctionCall                  -> Stmntz {cons("FunctionCall")}
Send                          -> Stmnt {cons("Send")}
Receive                       -> Stmnt {cons("Receive")}
Assign                        -> Stmnt {cons("Assign")}
"else"                        -> Stmntz {cons("Else")}
"break"                       -> Stmnt {cons("Break")}
"goto" ID                     -> Stmnt {cons("Goto")}
ID ":" Stmnt                  -> Stmnt {cons("Label")}
ID ":" Stmntz                 -> Stmntz {cons("Label")}
"printf" "(" STRING? ")"      -> Stmnt {cons("Printf")}
"printf" "(" STRING "," {AnyExpr ","}+ ")"
                              -> Stmnt {cons("PrintfVars")}
"printm" "(" FullExpr ")"     -> Stmnt {cons("Printm")}
"assert" FullExpr             -> Stmnt {cons("Assert")}
FullExpr                      -> Stmnt {cons("FullExpr")}
CCODE                         -> Stmnt {cons("Ccode")}

Varref ":" AnyExpr ".." AnyExpr -> Range {cons("RangeStandard")}
Varref "in" Varref              -> Range {cons("RangeArray")}

("::" Sequence)+              -> Options {cons("Option")}

Expr                          -> FullExpr {cons("Expr")}
AnyExpr                       -> FullExpr {cons("AnyExpr")}

ChanpollOp "(" Varref ")"     -> Expr {cons("Chanpoll")}
"(" Expr ")"                  -> Expr {cons("ParExpr")}
Expr "&&" Expr                -> Expr {left, cons("AND")}
Expr "&&" AnyExpr             -> Expr {left, cons("AND")}
AnyExpr "&&" Expr             -> Expr {left, cons("AND")}
Expr "||" Expr                -> Expr {left, cons("OR")}
Expr "||" AnyExpr             -> Expr {left, cons("OR")}
AnyExpr "||" Expr             -> Expr {left, cons("OR")}

"full"                        -> ChanpollOp {cons("ChanFull")}
"nfull"                       -> ChanpollOp {cons("ChanNfull")}
"empty"                       -> ChanpollOp {cons("ChanEmpty")}
```

```
"nempty"                    -> ChanpollOp {cons("ChanNempty")}

"(" AnyExpr ")"             -> AnyExpr {cons("Parentheses")}
AnyExpr "*" AnyExpr         -> AnyExpr {left, cons("Times")}
AnyExpr "/" AnyExpr         -> AnyExpr {left, cons("Div")}
AnyExpr "%" AnyExpr         -> AnyExpr {left, cons("Mod")}
AnyExpr "+" AnyExpr         -> AnyExpr {left, cons("Plus")}
AnyExpr "-" AnyExpr         -> AnyExpr {left, cons("Minus")}

AnyExpr "&" AnyExpr         -> AnyExpr {left, cons("And")}
AnyExpr "^" AnyExpr         -> AnyExpr {left, cons("Power")}
AnyExpr "|" AnyExpr         -> AnyExpr {left, cons("Or")}

AnyExpr ">" AnyExpr         -> AnyExpr {left, cons("GreaterThen")}
AnyExpr "<" AnyExpr         -> AnyExpr {left, cons("LessThen")}
AnyExpr ">=" AnyExpr        -> AnyExpr {left, cons("GreaterThenEq")}
AnyExpr "<=" AnyExpr        -> AnyExpr {left, cons("LessThenEq")}
AnyExpr "==" AnyExpr        -> AnyExpr {non-assoc, cons("Equal")}
AnyExpr "!=" AnyExpr        -> AnyExpr {non-assoc, cons("NotEqual")}
AnyExpr "<<" AnyExpr        -> AnyExpr {left, cons("LeftShift")}
AnyExpr ">>" AnyExpr        -> AnyExpr {left, cons("RightShift")}
AnyExpr "&&" AnyExpr        -> AnyExpr {left, cons("LazyAnd")}
AnyExpr "||" AnyExpr        -> AnyExpr {left, cons("LazyOr")}

"-" AnyExpr                 -> AnyExpr {cons("Negative")}
"~" AnyExpr                 -> AnyExpr {cons("BitNot")}
"!" AnyExpr                 -> AnyExpr {cons("Not")}

"(" AnyExpr Sep AnyExpr ":" AnyExpr ")"
                              -> AnyExpr {cons("CondExpr")}
"len" "(" Varref ")"          -> AnyExpr {cons("Length")}
Poll                          -> AnyExpr {cons("Poll")}
Varref                        -> AnyExpr {cons("Varref")}
CONST                         -> AnyExpr {cons("Const")}
"timeout"                     -> AnyExpr {cons("Timeout")}
"np_"                         -> AnyExpr {cons("NP")}
"enabled" "(" AnyExpr ")"     -> AnyExpr {cons("Enabled")}
"pc_value" "(" AnyExpr ")"    -> AnyExpr {cons("Pcvalue")}
ID "[" AnyExpr "]" "@" ID     -> AnyExpr {cons("RemoteLabel")}
ID "[" AnyExpr "]" ":" ID     -> AnyExpr {cons("RemoteVariable")}
ID  "@" ID                    -> AnyExpr {cons("RemoteLabel")}
ID  ":" ID                 -> AnyExpr {cons("RemoteVariable"), avoid}
"run" ID "(" {AnyExpr ","}* ")" Priority?-> AnyExpr {cons("Run")}
"run" ID                      -> AnyExpr {cons("Run")}
"_pid"                        -> AnyExpr {cons("ReadOnlyPid")}
"_nr_pr"                      -> AnyExpr {cons("ReadOnlyNrPr")}
"_last"                       -> AnyExpr {cons("ReadOnlyLast")}

Typename                      -> Type{cons("SimpleType")}
TypenameUnsigned              -> TypeUnsigned{cons("SimpleType")}
```

```
"bit"                       -> Typename{cons("Bit")}
"bool"                      -> Typename{cons("Bool")}
"byte"                      -> Typename{cons("Byte")}
"short"                     -> Typename{cons("Short")}
"int"                       -> Typename{cons("Int")}
"mtype"                     -> Typename{cons("Mtype")}
"chan"                      -> Typename{cons("Chan")}
"pid"                       -> Typename{cons("Pid")}
"unsigned"                  -> TypenameUnsigned{cons("Unsigned")}
ID                          -> Typename{cons("Uname")}

";"                         -> Sep{cons("Sep")}
"->"                        -> Sep{cons("Sep")}
```

Appendix C

Macro grammar

Lexical syntax

```
[a-zA-Z\_][a-zA-Z0-9\_]*      -> IDMAC
[a-zA-Z\_][a-zA-Z0-9\_]*      -> IDIFDEF
[a-zA-Z\_][a-zA-Z0-9\_]*      -> MACRO

~[\\ \n \#]+                  -> ANY
[\\]                          -> ENDML
~[\<\>]+                      -> INCLUDENAME
~[\#]*                        -> ANYSPIN
```

Lexical restrictions

```
IDMAC                         -/- ~[\(]
IDIFDEF                       -/- ~[\n\ ]
MACRO                         -/- ~[\n]

ANY                           -/- ~[\\ \n ]
ENDML                         -/- ~[\n\ ]
INCLUDENAME                   -/- ~[\>]
ANYSPIN                       -/- ~[\#]
```

Context-free syntax

```
"#define" ID ANY (ENDML ANY)* -> Module{cons("MacroDef"),avoid}
"#define" MACRO               -> Module{cons("MacroDef"), prefer}
"#define" IDMAC"(" {ID ","}* ")" ANY (ENDML ANY)*
                        -> Module{cons("MacroFunctionDef"),prefer}

"#include" STRING             -> Module{cons("MacroInclude")}
"#include" "<"INCLUDENAME ">" -> Module{cons("MacroInclude")}
```

```
"#if" "_EPISPIN_" ANYSPIN "#endif"
                                -> Module{cons("MacroAllesmag")}
"#if" ANY Module+ ("#elif" ANY Module+)* MacroElseModule?
                         "#endif"-> Module{cons("MacroIf")}
"#else" Module+       -> MacroElseModule{cons("MacroElseModule")}
"#ifdef" IDIFDEF Module+ MacroElseModule? "#endif"
                                -> Module{cons("MacroIfDef")}
"#ifndef" IDIFDEF Module+ MacroElseModule? "#endif"
                                -> Module{cons("MacroIfNDef")}

"#define" ID ANY (ENDML ANY)* -> Stmntz{cons("MacroDef"),avoid}
"#define" MACRO               -> Stmntz{cons("MacroDef"), prefer}
"#define" IDMAC"(" {ID ","}* ")" ANY (ENDML ANY)*
                        -> Stmntz{cons("MacroFunctionDef"),prefer}

"#include" STRING             -> Stmntz{cons("MacroInclude")}
"#include" "<"INCLUDENAME ">" -> Stmntz{cons("MacroInclude")}

"#if" "_EPISPIN_" ANYSPIN "#endif"
                                -> Stmntz{cons("MacroAllesmag")}
"#if" ANY Sequence ("#elif" ANY Sequence)* MacroElseStmnt?
                        "#endif" -> Stmntz{cons("MacroIf")}
"#else" Sequence         -> MacroElseStmnt{cons("MacroElseStmnt")}
"#ifdef" IDIFDEF Sequence MacroElseStmnt? "#endif"
                                -> Stmntz{cons("MacroIfDef")}
"#ifndef" IDIFDEF Sequence MacroElseStmnt? "#endif"
                                -> Stmntz{cons("MacroIfNDef")}
```

Appendix D

LTL grammar

```
"(" LTLAnyExpr ")"           -> LTLAnyExpr {cons("Parentheses")}
LTLAnyExpr "*" LTLAnyExpr    -> LTLAnyExpr {left, cons("Times")}
LTLAnyExpr "/" LTLAnyExpr    -> LTLAnyExpr {left, cons("Div")}
LTLAnyExpr "%" LTLAnyExpr    -> LTLAnyExpr {left, cons("Mod")}
LTLAnyExpr "+" LTLAnyExpr    -> LTLAnyExpr {left, cons("Plus")}
LTLAnyExpr "-" LTLAnyExpr    -> LTLAnyExpr {left, cons("Minus")}

LTLAnyExpr "&" LTLAnyExpr    -> LTLAnyExpr {left, cons("And")}
LTLAnyExpr "^" LTLAnyExpr    -> LTLAnyExpr {left, cons("Power")}
LTLAnyExpr "|" LTLAnyExpr    -> LTLAnyExpr {left, cons("Or")}

LTLAnyExpr ">" LTLAnyExpr -> LTLAnyExpr {left,
cons("GreaterThen")}
LTLAnyExpr "<" LTLAnyExpr -> LTLAnyExpr{left, cons("LessThen")}
LTLAnyExpr ">=" LTLAnyExpr-> LTLAnyExpr{left,
cons("GreaterThenEq")}
LTLAnyExpr "<=" LTLAnyExpr-> LTLAnyExpr{left, cons("LessThenEq")}
LTLAnyExpr "==" LTLAnyExpr-> LTLAnyExpr{non-assoc, cons("Equal")}
LTLAnyExpr "!=" LTLAnyExpr-> LTLAnyExpr{non-assoc,
cons("NotEqual")}
LTLAnyExpr "<<" LTLAnyExpr-> LTLAnyExpr{left, cons("LeftShift")}
LTLAnyExpr ">>" LTLAnyExpr-> LTLAnyExpr{left, cons("RightShift")}
LTLAnyExpr "&&" LTLAnyExpr-> LTLAnyExpr{left, cons("LazyAnd")}
LTLAnyExpr "||" LTLAnyExpr-> LTLAnyExpr{left, cons("LazyOr")}

"-" LTLAnyExpr               -> LTLAnyExpr {cons("Negative")}
"~" LTLAnyExpr               -> LTLAnyExpr {cons("BitNot")}
"!" LTLAnyExpr               -> LTLAnyExpr {cons("Not")}

"("LTLAnyExpr Sep LTLAnyExpr ":" LTLAnyExpr ")"
                             -> LTLAnyExpr {cons("CondExpr")}
"len" "(" Varref ")"         -> LTLAnyExpr {cons("Length")}
Poll                         -> LTLAnyExpr {cons("Poll")}
Varref                       -> LTLAnyExpr {cons("Varref")}
CONST                        -> LTLAnyExpr {cons("Const")}
```

```
"timeout"                       -> LTLAnyExpr {cons("Timeout")}
"np_"                           -> LTLAnyExpr {cons("NP")}
"enabled" "(" LTLAnyExpr ")"    -> LTLAnyExpr {cons("Enabled")}
"pc_value" "(" LTLAnyExpr ")"   -> LTLAnyExpr {cons("Pcvalue")}
ID "[" AnyExpr "]" "@" ID       -> LTLAnyExpr {cons("RemoteLabel")}
ID "[" AnyExpr "]" ":" ID   -> LTLAnyExpr {cons("RemoteVariable")}
ID  "@" ID                      -> LTLAnyExpr {cons("RemoteLabel")}
ID  ":" ID           -> LTLAnyExpr {cons("RemoteVariable"), avoid}

"_pid"                        -> LTLAnyExpr {cons("ReadOnlyPid")}
"_nr_pr"                      -> LTLAnyExpr {cons("ReadOnlyNrPr")}
"_last"                       -> LTLAnyExpr {cons("ReadOnlyLast")}
"np_"                         -> LTLAnyExpr {cons("NP")}

LTLAnyExpr Until LTLAnyExpr -> LTLAnyExpr{left, cons("LTLUntil")}
LTLAnyExpr Release LTLAnyExpr
                            -> LTLAnyExpr{left, cons("LTLRelease")}
LTLAnyExpr WeakUntil LTLAnyExpr
                            -> LTLAnyExpr{left, cons("LTLWeak")}
LTLAnyExpr "->" LTLAnyExpr  -> LTLAnyExpr{left, cons("LTLImpl")}
LTLAnyExpr "implies" LTLAnyExpr
                            -> LTLAnyExpr{left, cons("LTLImplies")}
LTLAnyExpr Equiv LTLAnyExpr -> LTLAnyExpr{left, cons("LTLEquiv")}
Next LTLAnyExpr               -> LTLAnyExpr{ cons("LTLNext")}
Always LTLAnyExpr             -> LTLAnyExpr{ cons("LTLAlways")}
Eventually LTLAnyExpr         -> LTLAnyExpr{ cons("LTLEventually")}

"[]"                          -> Always{cons("Always")}
"always"                      -> Always{cons("Always")}
"eventually"                  -> Eventually{cons("Eventually")}
"<>"                          -> Eventually{cons("Eventually")}
"U"                           -> Until{cons("Until")}
"until"                       -> Until{cons("Until")}
"stronguntil"                 -> Until{cons("Until")}
"weakuntil"                   -> WeakUntil{cons("WeakUntil")}
"W"                           -> WeakUntil{cons("WeakUntil")}
"X"                           -> Next{cons("Next")}
"next"                        -> Next{cons("Next")}
"<->"                         -> Equiv{cons("Equiv")}
"equivalent"                  -> Equiv{cons("Equiv")}
"V"                           -> Release{cons("Release")}
"release"                     -> Release{cons("Release")}
```

Appendix E

Implemented Errors

E.1 Language independent errors

- Multiple never claims are defined and only one is used in a verification run. Choose which one with `./pan -N name`
- Multiple (no)trace claims are defined (2)
- Label < *name* > is undefined
- Variable < *name* > is undefined (4)
- Variable < *name* > can not be used like this (4)
- Process < *name* > is undefined
- Inline call < *name* > is undefined
- Inline < *name* > is called with the wrong amount of parameters
- Inline < *name* > is defined multiple times
- Variable < *name* > is not a process
- Proctype < *name* > is called with the wrong amount of parameters
- Array < *name* > can not be passed as an argument
- This label is declared before
- This variable is declared before (4)
- This never claim is redefined
- This Ltl claim is redefined
- Macro definition defined before

- This proctype name is declared before
- Variable $<$ *name* $>$ is not a channel

E.2 Errors imposing additional constraints on the syntax

- Array in parameter list
- Misplaced break statement
- A remote reference may only be used in never claims
- pc_value may only be used in never claims
- np_ may only be used in never claims
- enabled may only be used in never claims
- Cannot jump to the guard of an escape since it is not a unique state
- Label $<$ *lab* $>$ placed incorrectly. Use "Label: { stmnt unless stmnt }" instead
- Label$<$ *lab* $>$ placed incorrectly. When a statement starts with a label, the label should be placed in front of the do-statement
- Label $<$ *lab* $>$ placed incorrectly. When a statement starts with a label, the label should be placed in front of the if-statement
- Label $<$ *lab* $>$ placed incorrectly. When a sequence starts with a label, the label should be placed in front of the sequence
- Initializer in parameter list
- Duplicate else (2)
- Dubious else with channel operation (2)
- Cannot hide channels
- More than one run operator in expression
- Atomic or d_step in never claim
- Never claim contains i/o statements
- Run in d_step sequence
- Unless statement in never claim
- A bit variable can not be hidden
- A channel initializer can not be used for a non-channel variable

E.3 Construction specific errors

- There is no runnable process
- This macro call is undefined or does not expand to a constant (6)
- _EPISPIN_ undefined
- Hidden array in parameter
- Variable $< varref >$ is not a channel (6)
- No array index specified (2)
- Identifier $< idt >$ is not an array (4)
- No array index specified (2)
- Label $< lab >$ is not defined in process $< name >$
- Can not assign a value to a channel
- Syntax error: Run should be followed by parenthesis
- A sequence needs at least one statement (3)
- A reference to the array element is invalid in this context. Instead of, e.g., `x[rs] qu[drie]`, use `chan nm_3 = qu[drie]; x[rs] nm_3;`
  and use `nm_3` in sends/recvs instead of `qu[drie]`
- Exclusive receive defined multiple times for channel $< name >$ (2)
- Exclusive send defined multiple times for channel $< name >$ (2)
- This type is not defined
- No field $< field >$ defined in structure $< uref >$ (4)
- Start value exceeds end value
- Do not index an array with itself

Appendix F

Building EpiSpin

This appendix describes which files and folders are necessary to build EpiSpin with Spoofax. The source code should be downloaded from `http://epispin.ewi.tudelft.nl/`. Here a zip file can be found that contains all source files that are used for building EpiSpin. Here a list is given with the most important files and folders that are necessary to build the EpiSpin project. Names that are shown in a bold font are folder names and names that are shown in an italic font are file names.

- **editor** - This folder includes the editor service description files. For the semantic editor services these files state which strategies should be used to start an editor service. For the syntactic editor services like the outline view and code folding these files contain a list of constructors that are included for these editor services. For all files there is a generated version and a customizable version. The *builders* file state which menu items are added to the 'Transform' menu. Finally, there is a main file which contains some general properties about the project, including the extensions that should be parsed by EpiSpin. The files are:
  - *Promela-Builders.esv*
  - *Promela-Colorer.esv*
  - *Promela-Completions.esv*
  - *Promela-Folding.esv*
  - *Promela-Outliner.esv*
  - *Promela-References.esv*
  - *Promela-Syntax.esv*
  - *Promela.main.esv*
- **editor/java** - In this folder all hand-written Java files are stored in four different folders:
  - **Spin/strategies** : The custom strategies written in Java for opening the model in *iSpin* or preprocess it with the C preprocessor.
  - **spingui.actions** : Java code for including the SPIN syntax checker in EpiSpin.

  - **spingui.editors** : All Java code related to the simulation and verification of models.
  - **spingui.wizards** : Java code for the wizard for creating new *Promela* files with option files.

- **lib** - This folder contains two pre-defined Stratego files: *editor-common.generated.str* and *refactor-common.generated.str*. Both contain helping strategies.
- **syntax** - The syntax folder includes all SDF-files from which Spoofax generates a parser. The only reason for using three files for the syntax definition is to separate different aspects of the grammar and to keep the files clear.
  - *Common.sdf* - Lexical syntax
  - *LTL.sdf* - Context-free syntax of LTL expressions
  - *Promela.sdf* - Context-free syntax of the rest of the *Promela* language
- **test** - This folder contains three folders with the test suites that are used to test EpiSpin:
  - **unitTest**
  - **syntacticSet**
  - **semanticSet**
- **trans** - In this folder all hand written Stratego code can be found:
  - *check.str* : error checking strategies
  - *complete.str* : completion strategies
  - *desugar.str* : desugar strategies
  - *generate.str* : strategies to generate DOT code
  - *pp.str* : pretty-print rules that return a textual representation of a term
  - *promela.str* : general strategy for starting analysis and strategies that are called when an item from the 'Transform' drop down menu is clicked
  - *rename.str* : all strategies used during renaming
  - *resolve.str* : reference resolving strategies
  - *type.str* : helping strategies to check the type of an identifier
  - *utype.str* : all strategies related to user-defined types
- The next files contain build and run scripts and general information about the EpiSpin project.
  - *build.generated.xml*
  - *build.main.xml*
  - *build.properties*
  - *plugin.xml*
  - *strategoxt.jar* : The standard Stratego libraries

Building EpiSpin

Follow the next steps to include the EpiSpin project in Eclipse.

1. Download the zip file from `http://epispin.ewi.tudelft.nl/` and unzip it.

2. Start Eclipse and download the latest **unstable release** of Spoofax according to `http://strategoxt.org/Spoofax/Tour#Installation`.

3. In Eclipse, go to *File* $->$ *Import* and select *Existing Projects into Workspace* from the *General* folder and click *Next*.

4. In the next window, browse to the unpacked folder in the *Select root directory* field and click Finish. EpiSpin will now be imported in Eclipse and an initial built is started.

Now changes can be made to the source code. The general steps for including new constructions is as follows. First the new construction is added to the grammar. New keywords are indicated in *Common.sdf* and the grammar rule is added to *Promela.sdf*. This rule should be annotated with a constructor name to identify it during analysis. A rename rule that renames any direct subterms of the new constructor and applies the rename strategy top-down should be defined in *rename.str*. The next step is to decide if this construction should be visible in the outline view and if it is foldable. If so, the constructor name should be added to the corresponding files. When semantic error messages need to be added a strategy called `constraint-error` should be added. This strategy should transform the new constructor to a tuple (*position*, $[*Error message*]) when the conditions in the `where` clause succeed. In the *resolve.str* and *complete.str* files the strategies for reference resolving and content completion could be added in *check.str*. When the Static Communication Analyzer should traverse the new construction to look for channel usage for example, a simple strategy should be added in the *generate.str* file. This strategy only needs to apply the `to-dot` rule to all subterms of this new constructor. Since the `to-dot` strategies for the existing language structures are already implemented, the `to-dot` rule can be applied recursively.

When you want to use the new construction in the editor, the modified files should be saved and the project should be rebuilt (using *Project* $->$ *Build project* or the shortcut Ctrl+Alt+B). The new editor is now dynamically loaded in the same Eclipse instance and *Promela* models using the new construction are parsed correctly.

More information

- To learn more about how editor services are implemented with Spoofax, take the Spoofax tour at `http://strategoxt.org/Spoofax/Tour`.

- For more information about SDF, read the SDF Reference Manual at `http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.html` or [36] for more information on scannerless generalized-LR parsing.

- To master the Stratego language, read the manual on `http://hydra.nixos.org/build/1564906/download/1/manual/chunk-chapter/index.html` or use the API documentation at `http://releases.strategoxt.org/docs/api/libstratego-lib/stable/docs` to use predefined strategies