SPIN's Promela to Java Compiler, with help from Stratego

Master's Thesis

Edwin Vielvoije

SPIN's Promela to Java Compiler, with help from Stratego

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Edwin Vielvoije
born in Rotterdam, the Netherlands

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

SPIN's Promela to Java Compiler, with help from Stratego

Author: Edwin Vielvoije
Student id: 1150731
Email: edwinus@gmail.com

Abstract

In model checking a formal model of a software system is constructed. That model is verified against a set of properties expressed in some logic. Once a model has been created and verified, it is still necessary to write the application itself completely by hand. No tools have yet been developed that can automatically create a system or application using a model written in Promela. .... **until now!**

The Promela2Java Compiler to be described in this paper is a unique tool transforming a Promela model into an executable Java application. The Promela2Java Compiler has been constructed using the Stratego/XT tool set. Developers can use SPIN to check their designs for certain properties and use the Promela2Java compiler to successfully create executable Java code from their designs.

Thesis Committee:

Chair:	prof. dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	ir. C. Pronk, Faculty EEMCS, TU Delft
Committee Member:	D. M. Groenewegen MSc, Faculty EEMCS, TU Delft
Committee Member:	dr. ir. T.C. Ruys, Dep. Computer Science, University of Twente

Preface

This document is the result of a project in which I learned that I will never stop learning. A lot of people were able to teach some the things that I needed to learn for this project. I would like to thank everybody that helped me. Thank you all!!!

As the person that helped me the most I would like to thank my supervisor Ir. C. Pronk. I want to thank him for the patience he had with me and pushing me to keep on working.

Another big help was Danny Groenewegen MSc, I would like to thank him for all the times he advised me on the problems I had dealing with Stratego, I would have been in big trouble if he did not help me. I also would like to thank Eelco Visser, Lennart Kats and Zef Hemel who all also answered some of my questions about Stratego.

I would like to thank Drs. D. E. Butterman-Dorey who pushed me into writing a lot and early on in the project. Of course I would like to thank my family for being there for me and making sure I also took time to relax during the course of this project.

Edwin Vielvoije
Rotterdam, the Netherlands
August 21, 2008

Contents

List of Figures

Chapter 1

Introduction

Model Checking is a technology to be used in the design phase of a system. In model checking a formal model of a software system is constructed. That model is then used to verify the design. SPIN is such an model verifier. The models that SPIN verifies are written in the language Promela [3, 19].

Although Model checking can be very useful, once a model has been created it is still necessary to write the application itself completely by hand. No tools to automatically create a system or application from a model written in Promela are known. The Promela language contains many semantic features that can be represented by parts of the Java language [27]. Java [24] is therefore a good target language for a tool to create an executable system or application from a model written Promela.

Such a tool will be used by developers who have created a design for a software system or application and who would like to know if the system or application conforms to the design requirements. These developers would then create a Promela model of their design and use SPIN to check if their design adheres to their formal specifications.

Developers can use the Promela2Java compiler that will be described in this paper to generate Java code from their Promela model. The Java code generated can then immediately be used as their newly created software system or application. The code generated can also be adapted and used in a larger system containing code that Promela is not able to represent, such as a Graphical User Interface.

The Promela2Java compiler will not do any error checking on the Promela source, because a developer would use SPIN before transforming their model with the compiler. The user is assumed to input a Promela model for the Promela2Java compiler that has been properly checked by the SPIN system.

1.1 Research

I have done some research for this project and can be found in my previous paper [30]. In this paper I showed whether Model Driven Architecture [20, 23] (MDA) could be used to make a transformation from Promela to Java possible. In order to find out if MDA could be used I needed to know what languages and tools could be used for the different parts

that are created in MDA. If those languages and tools could help me in creating such a transformation, then MDA could be used for this project. I also wanted to know what already has been done in other research about transforming the Promela language into some other language and how this would translate to certain Java properties. I needed to find a certain solution for the parts of which there was no known transformation as well.

Model Driven Architecture

An MDA design consists of a Platform Independent Model or PIM, that describes a system or application without dependencies on any platform. A platform can be anything from a programming language to hardware to an operating system. A PIM would then be transformed into a Platform Specific Model or PSM, which still describes the system or application but now the design is specific to a certain platform on which the system or application will eventually be realized. A PSM would then be transformed into actual source code to create the actual system or application. The PIM and PSM can be written in different or the same languages and all transformations between the different models are done automatically via (MDA) tools. An overview can be found in Figure 1.1.

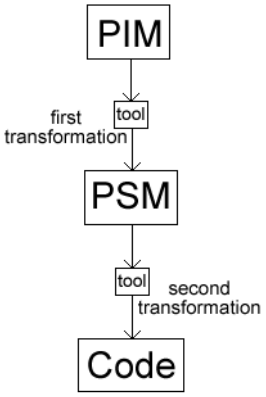

Figure 1.1: The MDA models and their relation

If MDA would be used for this project then a Platform Independent Model (PIM) should be created that describes the Promela language independent from any platform, such as any programming language, operating system or hardware. This PIM should then be transformed automatically via (MDA) tools into a Platform Specific Model (PSM), which in this case would be specific to the Java language. The derived PSM should then be transformed into source code, which in this case would be Java code [24]. This Java code should then be used as a framework of which each Promela model would depend on after its transformation to Java.

UML

My research [30] indicated that UML [6, 7, 12] probably is the most widely known and most commonly used modeling language. UML consists of a number of different diagrams.

The most import models are the class, sequence, state-machine and use case diagrams.

The SPIN program creates a state-machine from a Promela model to verify the requirements of the system that is modeled [17, 19]. A state-machine diagram can be created easily of a Promela model. The state-machine will cover every possible way of executing the model, just as SPIN does. However, this will end up in a state-machine diagram with a lot of states. The bigger the Promela model is, the larger the amount of states will be in the diagram. This diagram shows how the model behaves, but does not show the components the model consists of.

One could try creating a class diagram corresponding to a Promela model by listing the processes and channels as classes in the diagram and the variables as attributes in the classes. There should then always be an extra class to contain all the global variables and the previously described channels. The processes can access that class to gain information about the global variables or change them and send or receive from the declared channels. This diagram can be used to show what parts the model consists of, but does not show anything at all about the behavior.

A sequence diagram of a Promela model is a bit like a simulated run in SPIN. The verification will take all paths whilst the simulation only takes one path [17, 19]. In this way a sequence diagram cannot show all the possible behavior of a model or sometimes not even all the components of the model. A sequence diagrams can be created for every possible path in the model. All of those sequence diagrams together show all the behavior and all the components that are used in the model. However, as with the state-machine diagram, the number of paths is large and will be even larger when a model grows bigger.

A use case diagram can be created for a Promela model. The processes will then be the actors and the use cases will then describe an action the process will undertake. The use cases follow each other in the same way the actions follow each other in the Promela model. The use case diagram shows the behavior of the separate processes but not how those processes interact. These diagrams do not show when which process executes which actions.

Action Semantics

UML is not a good choice to use as a language for the PIM because UML is not able to show the non-deterministic and interleaving nature of the Promela language. Action Semantics [25] did show promise to be used instead.

Action semantics can be used to describe a programming language. In [25] a complete description can be found on Action Semantics. There is no graphical representation for any part of the Action Semantics, it only consists of formulas. When Action Semantics are used to describe a language, then three parts need to be created. Those three parts are an abstract syntax, semantic functions and semantic entities. These three parts together can be used to describe a programming language. These descriptions could be used as a PIM in this project.

The abstract syntax provides as described in [25] "*an appropriate interface between the concrete syntax and the semantics*". An abstract syntax can most of the time be obtained by looking at parse tree structures and leave out those details which have no semantic signifi-

cance. Semantic functions are used to link the abstract syntax to the semantic entities that represent the particular behavior. The semantic entities are used to represent that behavior in a implementation-independent manner. A semantic entity can be one of three kinds: an action, data or a yielder. Actions are computational entities that represent the information that is being processed. Data entities are mathematical entities that represent pieces of information and whilst the actions are dynamic, the data is static. A yielder represents data, whose value dependents on the information that exist at the time of execution.

Non-deterministic choices can be dealt with in Action Semantics. There are actions like `_or_` which chooses to do either one or the other action. By contrast there are also actions such as `_and then_` which is used to show sequential execution. The interleaving of a Promela model can be accomplished with agents in Action Semantics. An agent represents a process and communicates with other agents by sending messages .

There are however, no (MDA) tools available that make a transformation from a PIM written in Action Semantics into any kind of PSM. This is because Action Semantics is not widely used. Which indicated that if Action Semantics would be used then the transformation between PIM and PSM should be done completely by hand. Besides this problem, there is also the problem that when Action Semantics will be used then the creation of the PIM or the PSM does no help at all in the creation of the Promela2Java compiler. This is because the PIM and PSM will describe the Promela language but will not describe the transformations.

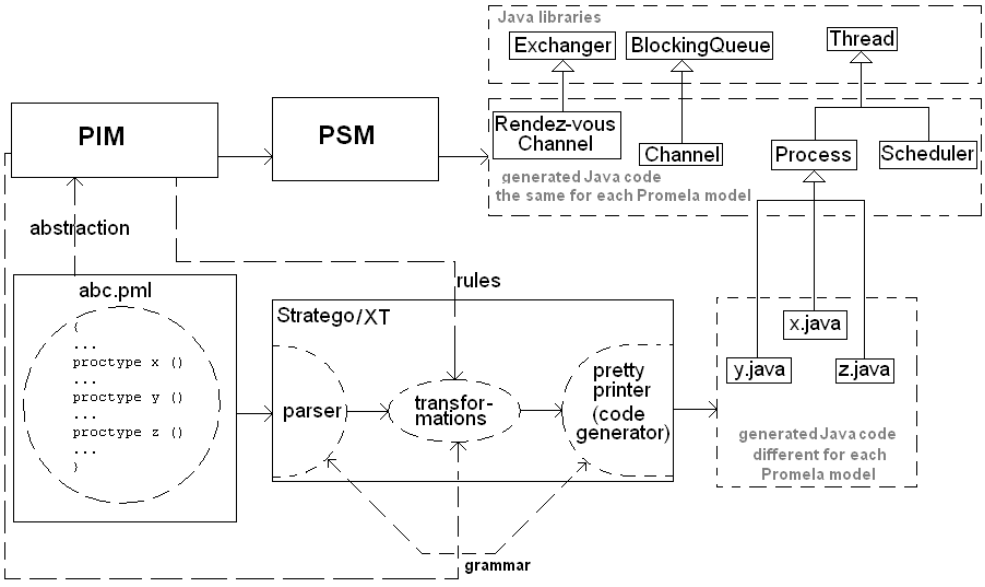

Figure 1.2: Project overview sketch

Promela2Java Compiler

When you read a book or article about compiler construction, such as [15], then you know that a compiler mainly consist out of three parts. The front-end, the semantic representation and the back-end. The front-end of the compiler analyzes the program in the language that needs to be compiled, in this case Promela models. The back-end creates a program in the target language, which is in this case Java. The semantic representation glues both parts together. The output of the front-end is an abstract syntax tree of the input program. The semantic representation part of the compiler takes that abstract syntax tree and constructs intermediate code. Finally, the back-end takes the intermediate code and generates the code in the target language.

I showed in [30] as well that the transformation from a Promela model to Java code could be done by creating a compiler with the Stratego/XT toolset [9, 31]. With this toolset a parser, a transformation and code generator (or pretty-printer) can be created from a grammar and transformation rules written by a user. These three parts are practically equal to the parts of the compiler. The created parser represents the front-end of the compiler, whilst the transformation rules represent the semantic representation part of a compiler and the back-end is represented by the code generator. A separate package for the toolset is even able to understand parts of the Java language, which could be very helpful because the target language of the compiler that needs to be created is the Java language. More on how Stratego works can be found further on in this chapter.

Promela to Java Transformations

Information that showed up during my research [30] were some Promela to Java transformations that could be used to create a compiler. All these transformations can be found in the Table 1.1, the left column shows the Promela part and the right column shows what those parts should transform to in Java or what kind of construction could help facilitate the Promela part in Java.

Promela	**Java**
Processes	Threads
Variables	Variables
Data types	built-in data types and Classes
`d_step` and `atomic`	Lock interface and the Metalocking algorithm
Channels and Concurrency	`BlockingQueue` and `java.util.concurrent`
Non-determinism	Scheduler chooses a path and speculative execution
Channel operations	Methods in subclasses of `BlockingQueue`
Labels and `goto`	Loop and switch statement
`unless` statement	Executable checks and `if` statements
Embedded C code	No translation

Table 1.1: Promela to Java translations

For the Processes, data types and variables should it be easy to see how those could translate to the Java languages [27]. Two ways to deal with the `atomic` and `d_step` are the new Lock interface of Java 1.5 and the Metalocking algorithm [1, 4]. The Java Threads already deals with interleaving, this means that to make sure that no other process can interfere with the statements inside an `atomic` or a `d_step`, only those variables that are used within those statement need to be locked. This also means that before every statement is evaluated the variables that are going to be used need to be checked whether or not those are locked by using the `synchronized` statements of Java. A better possibility is to let a scheduler lock the other processes. This means when an `atomic` or `d_step` wants to execute, then the scheduler must be notified, which in turn will lock the other processes. This means that before every statement is evaluated the scheduler will be contacted to check whether or not the statement is allowed to execute.

In Java 1.5 there is the new `java.util.concurrency` package [27] that contains the `BlockingQueue` to deal with concurrency and the channels of Promela. To deal with non-determinism a scheduler can be created [22] to choose one of the possible paths that can be taken. To know which paths can be taken some sort of speculative execution [32, 33] needs to be enabled as well.

The `goto` statement does need a Java counterpart. The solution looks complicated because the labels can be anywhere in a model. The thing is to split the statements of the process into different parts separated by the labels. When a `goto` statement occurs, then the flow of control will jump to the correct part of the processes statements. The `goto` statement can be translated to Java, by using a loop and a `switch` statement. The switch will be inside the loop and the switch will break whenever a `goto` occurs. The loop makes sure the switch is called again and the label of the previous `goto` will make sure that a jump is made to the correct switch case.

The embedded C code will not be incorporated into the Promela2Java compiler because this project does have some time constraints. The entire C language cannot have any translation to Java within the given time limit along with everything else.

Conclusions

One of the conclusions of my research [30] was that MDA is not going to help with the creation of the Promela2Java compiler. Figure 1.2 shows an overview of how this project would have been set up. However, the PIM and the PSM do not help in the creation of the grammar and the transformation rules when Action Semantics are chosen as a language for the PIM. This means that a different language needed to be found for the PIM, but so far there has not been found any language that could show the properties of the Promela language and help with the creation of the grammar and transformation rules that are needed for a compiler. After the research I have been trying to transform a few small parts of the Promela language into Java using the Stratego/XT toolset to find some commonalities that could be exploited to find another language for the PIM. But nothing had turned up and work continued without using MDA but using only the documentation of Promela, SPIN [17, 19] and Stratego [9, 31].

Another conclusion from my research [30] results in Table 1.1. This table shows all the Promela properties that can be transformed into certain Java properties. Most of these transformations were derived from other papers [1, 4, 22, 27, 32, 33] and a little bit from my self (the `goto statement`). These transformations will be used in the Promela2Java compiler to ensure the generated Java application has the same properties as the corresponding Promela model.

1.2 Promela

SPIN is a model checker that uses the language Promela (Process Meta Language) as the language to write models in [18]. These models can be used in the designing or testing phase of a software system. A model created during the design phase will be a model of what a system is going to be, while a model created during the test process will be a model of the implemented system. SPIN will check whether the model will adhere to the requirements of that system. A model written in the Promela language consists of three types of objects: processes (proctypes), channels and variables [3].

Like with a lot of other programming languages variables can be declared. Values can be assigned to variables after those have been declared. Variables van be declared either globally or local to a certain process. Constants can be defined using the C-style macro `#define`. It is even possible to create symbolic constants, which can be defined as an `mtype`. The creation of structures is available in Promela and can be used in the same way as `typedef` in the programming language C. Embedded C code is also allowed inside a Promela model, which is usually marked by a prefix such as `c_code`. The use of C code is discouraged by SPIN because, SPIN is unable to verify those parts of the Promela model.

The processes in Promela describe the behavior of the modeled system. A process in Promela is known as a `proctype`. Statements reside within the processes, these define the behavior of that particular process. There can be multiple instances of the same process at runtime or in the case of SPIN at simulation or verification time, depending on how the process is modeled in the language. The initial process is a special process, generally used to create a true initial state of the system and usually starts other processes, however a normal process can also be started without the initial process, but there can only be one initial process.

A Promela statement can be either blocking or executable. When a statement is blocking, the process containing that statement will not continue executing until that statement is no longer blocking. Some statements, such as assignments and declarations are always executable, executability of the other statements depends on their run-time evaluation.

Channels together with variables describe the environment that processes work in. Channels are used to facilitate communication between the different processes and can be declared either globally or locally. A channel can contain different messages, a messages can consist of multiple parts of different or the same kind of type. A channel can only contain one type of message and that type needs to be defined when the channel is declared [3].

Generally, a send to a channel will be blocking when the channel buffer is full and will be executable again when there is room for the message at the end of the buffer. The receive

statement will be blocking when the channel is empty but also when there are constraints on parts of the message. A constraint will always enforce that the message or part of the message that is about to be received matches a certain value. Some of the other channel operations are random receive and sorted send.

A grammar definition of the complete Promela language and a more complete description of Promela and SPIN for which Promela has been developed can be found in [17, 19]. For a short introduction into Promela and for a little bit more information on the different possible operations in the Promela language, please refer to my previous paper [30].

1.3 Stratego/XT

The toolset Stratego/XT [31] has been designed for program transformations using rewrite rules. The toolset is called XT and the language used is known as Stratego; the toolset will from now on be described as Stratego. Using this tool set a compiler developer can create transformations from a textual document in one language to a document in another or the same language.

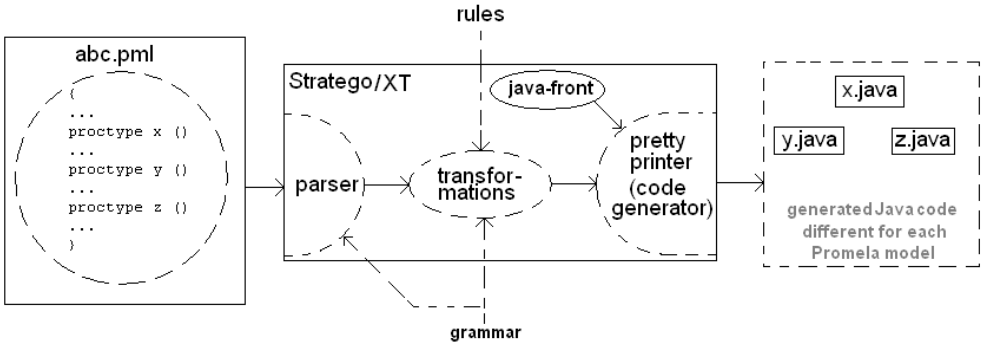

Figure 1.3: Overview of the different parts of a compiler written with the help of Stratego.

To create such transformations, first a grammar of the language to be transformed needs to be created. This grammar is written by the developer in the Syntax Definition Formalism (SDF) [16].

A parser tool from the Stratego toolset uses this grammar to generate the Abstract Syntax Tree corresponding to the source program in that language. The Abstract Syntax Tree is represented in Stratego as Annotated Terms or ATerms. Transformation rules can be applied to those terms by using another Stratego tool. These transformation rules are written by the developer of the compiler in the Stratego language.

What remains is a pretty-printer to generate code from the transformed terms. Code can be generated in the original language, by using the same grammar as before to create the pretty printer, with another tool from Stratego. Code can be generated into another language by using a different grammar to create a pretty-printer or another already existing pretty-printer.

The Stratego toolset also contains a Java-front package [31] delivering a few new tools. One tool parses Java code into ATerms and another Java pretty printer tool reads these ATerms and generates Java code [24] from them. The ATerms used by these tools are specific to the Java language. The Java pretty-printer can be used in the back-end of the Promela2Java compiler by using these terms in the transformation rules of the compiler. Figure 1.3 shows how the parts work together. This package allows embedded Java code within the transformation rules. The package even does syntax checks on the embedded Java code in those transformation rules. This makes it easier to create transformation rules, but also should make them easier to understand for anyone less familiar with Stratego.

More about how a grammar or transformation rules are created can be found in Chapters 2 and 3 or for a more complete description of the toolset please refer to [9, 31].

1.4 Problem statement

At the time of writing there is nothing *yet* that translates a Promela model to a Java application. Stratego could be used in order to create such a compiler. A grammar should be created to generate a parser. The ATerms that are generated from such a parser should be transformed using transformation rules that need to be created as well. These transformation rules will need to change the ATerms generated by the parser into ATerms that are used to represent Java code. Then the pretty-printer that is incorporated by the Java-front package of Stratego can generate Java code from the transformed ATerms.

All of this should be packed together to become a Promela2Java compiler. This compiler should be extensively tested as well and any encountered problems should be fixed. The original planning mentioned that everything should be finished at the beginning of June, however, due to to some delays the deadline had been shifted to the end of August. In order to keep that schedule, I have created the planning that can be found in Table 1.2.

Date		**Event**
20 January	2008	Start of Master Thesis
04 March	2008	Parser finished
17 June	2008	Transformation rules finished
11 August	2008	Testing of the compiler finished
12 August	2008	Promela2Java compiler finished
21 August	2008	Master Thesis finished
23 September	2008	Presentation of Master Thesis held at colloquium

Table 1.2: Planning

Creating the grammar in Stratego for the Promela language to create a parser will not be to difficult, because the original grammar can be found in [17, 19]. The difficult part will be to find all the parts that are not described in that original grammar such as an `inline` and a few other things. Another difficulty will be to make sure the grammar will make sure that a Promela model can only be parsed in one way and that there are no ambiguities.

The creation of the transformation rules will be difficult because I have never worked with Stratego before. However, embedded Java code can be used, which should make it easier for me to use and easier for anyone else to read.

I have had a course, named compiler construction, in which I had to change some big parts of a compiler, so I do have some experience with compiler constructing. That course has thought me that I will not have a completely error free compiler after the creation of the Promela2Java compiler and that I should thoroughly test the compiler and then there might still be some errors. The testing is done by creating all kinds of Promela models, which will include models that are suppose to work correctly. This testing phase will not be too difficult, but will be time consuming if done thoroughly.

1.5 Structure

This document will describe my findings on creating a compiler from Promela to Java. I have used Stratego to create this compiler. A parser was needed to create the compiler. How this parser was created and how an SDF grammar is created in Stratego can be found in Chapter 2. The abstract syntax tree that is a result from such a parser will need to be transformed to an abstract syntax tree that represents Java code. How these transformation rules were created can be found in Chapter 3.

The testing phase of the Promela2Java compiler will be described in Chapter 4, as well as any problems that were discovered during testing. Some extra information on the compiler can be found in Chapter 5. This will describe any parts of the Promela language that are not translated by the Promela2Java compiler and why those parts are not included as well as the instructions to get the compiler to work correctly will be described. An example of a Promela protocol transformed into Java code is described in Chapter 6.

Any work from other authors that is related to this project will be mentioned in Chapter 7. This document will finish with an evaluation of the Promela2Java compiler and of this project and the conclusions in Chapter 8.

Chapter 2

Parser

A parser that is created with Stratego, will take in a file written in the input language in this case Promela. The parser will then analyze the input file and recognizes the various tokens and transforms the file into an abstract syntax tree [9]. This chapter shows how a parser is created with Stratego and shows a small example. This chapter describes how the parser for this project was created as well.

The abstract syntax tree is formatted in Stratego as Annotated Terms or better known as ATerms. An ATerm consist of a constructor that is applied to zero or more terms. Strings and Integers are terms as well and usually represent a leaf in the abstract syntax tree.

As an example we could show how a simple expression such as `x - (3 * y)` will be transformed to abstract syntax tree as Annotated Term. When you look at the expression, you could easily imagine the abstract syntax tree in a graphical structure such as in Figure 2.1 [31]. The ATerm version of this abstract syntax tree will look something like:

```
Minus(Var("x"), Times(Int("3"), Var("y")))
```

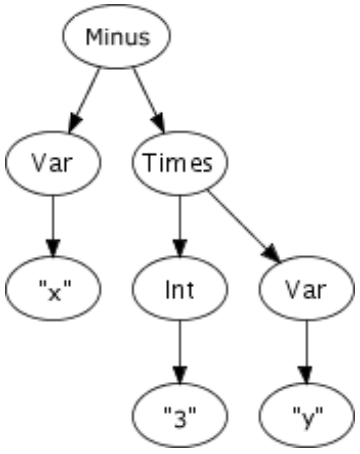

Figure 2.1: The graphical structure of the abstract syntax tree of a simple expression [31]

The parser needs to have some information in order to create these terms. This information is necessary to recognize the tokens from the input file. The information that is needed is a grammar written in the Syntax Definition Formalism [16, 9], which is also known as SDF. The grammar that is needed to recognize the terms for Promela will look a lot like the grammar of Promela itself [17, 19]. This is because you describe the Promela language for Stratego in the SDF grammar and the grammar of Promela itself exist for the purpose of describing the Promela language to whomever reads that documentation. The parser can be automatically derived from such an SDF grammar by using tools from the Stratego/XT toolset.

2.1 Example SDF Grammar

The SDF grammar will consist of certain rules. These rules have on the left hand side, one or a combination of more then one terms and on the right hand side, the term that the parser should recognize the combination as.

The example `x - (3 * y)` that was used above can be used here as well to show how such a simple expression can be parsed into the ATerm

```
Minus(Var("x"), Times(Int("3"), Var("y")))
```

This could be done by creating a simple grammar [1] such as Figure 2.2.

```
module SimpleExpression

exports
  context-free start-symbols Exp
  sorts Id IntConst Exp

  context-free syntax
    Name         -> Exp {cons("Var")}
    Integer      -> Exp {cons("Int")}
    "(" Exp ")" -> Exp {bracket}

    Exp "*" Exp -> Exp {left, cons("Times")}
    Exp "/" Exp -> Exp {left, cons("Divide")}
    Exp "%" Exp -> Exp {left, cons("Modulo")}

    Exp "+" Exp -> Exp {left, cons("Plus")}
    Exp "-" Exp -> Exp {left, cons("Minus")}

lexical syntax
  [\ \t\n]   -> LAYOUT
  [a-zA-Z]+ -> Name
  [0-9]+     -> Integer

context-free priorities
  {left:
    Exp "*" Exp -> Exp
    Exp "/" Exp -> Exp
    Exp "%" Exp -> Exp
  }
> {left:
    Exp "+" Exp -> Exp
    Exp "-" Exp -> Exp
  }
```

Figure 2.2: The grammar for a simple expression

This grammar recognizes an Exp or Expression. An Expression can be either a Name, an Integer or an Expression followed by an operator and another Expression. A Name term is recognized as one or more (as indicated by the + in the grammar) characters in the range of a-z or A-Z, which means any word of at least one letter. The Integer works in

[1] A modified example from [31]

a similar fashion, except with one or more characters from the range 0-9. When a Name is recognized then an Expression will be recognized and given the constructor Var, which indicates that is a variable. The same is true for the Integer but that kind of Expression is given the constructor Int.

When one of the operators is found between two Expressions then the appropriate Expression is recognized and given the correct constructor. The extra `left` that is found just before the constructor indicates that the operator groups from left to right, which means that `Exp1 + Exp2 + Exp3` will always group as `Plus(Plus(Exp1, Exp2), Exp3)` and not as `Plus(Exp1, Plus(Exp2, Exp3))`. The priorities of these operators are dealt with at the end of the grammar by grouping the rules for Multiplication, Division and Modulo and using the greater then symbol (>) to show that those expressions have a higher priority then the Addition and Subtraction.

The grammar can be separated into modules if the number of terms that need to be recognized is large. When a term that is declared in a different module is needed then a simple import statement could be used at the top of the module to import all the declared terms from that module. The grammar that was described in this section can recognize the following code and categorizes that piece of code as as shown below:

`x - (3 * y)` → `Minus(Var("x"), Times(Int("3"), Var("y")))`

2.2 Promela SDF Grammar

The SDF grammar that was created to parse Promela models, has been created using the grammar of Promela as a starting point. The grammar of Promela can be found in [17, 19]. The SDF grammar has been split up into several files. Every file can link to another file by using import statements. The files of the SDF grammar can be found in Appendix A.1. The top most term that can be recognized by the parser is the `Program` term, because a Promela model must consist of at least one or more modules.

There are a few different modules declared in the Promela grammar, so these must all be recognized as well by the parser. Each module will be recognized by one or more rules. As an example we could look at the `Proctype` module. The Promela grammar for a proctype[2] looks like the rules that are shown in Figure 2.3.

```
proctype    : [ active ] PROCTYPE name '(' decl_lst ')'
              [ priority ] [ enabler ] '{'sequence'}'
active      : ACTIVE [ '[' const ']' ]
name        : alpha [ alpha | const | '_' ]
priority    : PRIORITY const
enabler     : PROVIDED '(' expr ')'
```

Figure 2.3: The original grammar rules for a Proctype in Promela

A proctype consists of the optional word "`active`" followed by square brackets and a constant, which are optional as well, however those square brackets and constant must

[2] Without the rules for `decl_lst, sequence, expr` and `const` to keep everything more understandable

always follow the word "active". This is followed by the word "proctype" which is followed by the name of the proctype. The name of the proctype is described as a letter followed by zero or more letters, numbers or underscores. The name is followed by a pair of brackets, with in between an optional DeclarationList. This all is followed by an optional word "priority" and a constant, which in turn is followed by an optional word "provided" and an Expression followed between some brackets. At the end of the proctype declaration in the Promela grammar there are a pair of curly braces with in between them a Sequence.

The Promela grammar was used as a starting point for the SDF grammar. So the SDF rules for the proctype are a little bit different from the Promela grammar and will look like Figure 2.4.

```
  Proctype                    -> Module   {cons("Module")}

  Active "proctype" Id "(" DeclarationList* ")" "{" Sequence "}" -> Proctype     {cons("Proctype")}
  "proctype" Id "(" DeclarationList* ")" "{" Sequence "}"        -> Proctype     {cons("Proctype")}
  "proctype" Id "(" DeclarationList* ")" PriorEnabler "{" Sequence "}"
                                                                 -> Proctype     {cons("Proctype")}
  Active "[" Constant "]" "proctype" Id "(" DeclarationList*")" PriorEnabler "{" Sequence "}"
                                                                 -> Proctype     {cons("Proctype")}
  Active "[" Constant "]" "proctype" Id "(" DeclarationList* ")" "{" Sequence "}"
                                                                 -> Proctype     {cons("Proctype")}
  Active "proctype" Id "(" DeclarationList* ")" PriorEnabler "{" Sequence "}"
                                                                 -> Proctype     {cons("Proctype")}

  "active"                                      -> Active       {cons("Active")}
  "priority" Constant "provided" "(" Exp ")"    -> PriorEnabler {cons("PriorEnabler")}
  "priority" Constant                           -> PriorEnabler {cons("PriorEnabler")}
  "provided" "(" Exp ")"                        -> PriorEnabler {cons("PriorEnabler")}
  [a-zA-Z] [a-zA-Z0-9\_]*                       -> Id

lexical syntax
  "proctype"                -> Id {reject}
  "active"                  -> Id {reject}
  "priority"                -> Id {reject}
  "provided"                -> Id {reject}
```

Figure 2.4: The SDF grammar rules for a Proctype in Promela

For every optional part in the Promela grammar (such as the word "active" for proctype) , there are two versions of a rule in the SDF rules, one where the optional part is included and one where that part is not included. At the end there are a few lexical restrictions mentioned which represent that keywords are not recognized as Id. There can thus be no variables or proctypes with the same name as those keywords. For every keyword in the Promela language there exists such a lexical restriction in the SDF grammar.

The entire Promela grammar is transformed into an SDF grammar in a similar way. There are a few parts different though, such as whenever a term should be followed by a separating token (',' or ';') then the first term is held separately from the rest of the following terms. For example the rule for a Sequence: `sequence : step [ ';' step ]*`
In the SDF grammar this will be represented as:

```
Step -> Sequence {cons("Sequence")}
Step SepStep+ -> Sequence {cons("Sequence")}
";" Step -> SepStep {cons("SepStep")}
```

This will be the same for all terms that are followed by a list of the same terms, such as the declaration list and a few others. An exception to this part is the Varref term. In the original grammar this term is represented as:

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

The rules for the Varref in the SDF grammar is separated into two distinct Terms to avoid ambiguity. The first part of the Varref (before the first '.') is the part that can either be global or local. The second part of the Varref belongs to the variable and is not declared globally or locally. That is why there should be a distinct separation in the different parts of the Varref. In the SDF grammar this will look like:

```
Id                                  -> Varref {cons("Varref")}
Id "." XtraVarref                   -> Varref {cons("Varref")}
Id "[" AnyExpr "]"                  -> Varref {cons("Varref")}
Id "[" AnyExpr "]" "." XtraVarref   -> Varref {cons("Varref")}

Id                                  -> XtraVarref {cons("XtraVarref")}
Id "." XtraVarref                   -> XtraVarref {cons("XtraVarref")}
Id "[" AnyExpr "]"                  -> XtraVarref {cons("XtraVarref")}
Id "[" AnyExpr "]" "." XtraVarref   -> XtraVarref {cons("XtraVarref")}
```

The original grammar of Promela that can be found in [17, 19] is actually not complete. There are few parts missing from that grammar that are described in the manual pages of the Promela language, which can also be found in [17, 19]. Those parts however are represented in the SDF grammar. The predefined variables ('_', '_last', '_nr_pr' and '_pid') are not mentioned in the original grammar. These are defined as special Varref terms in the SDF grammar. Another missing part is the unsigned data type. An "inline" can be defined in a Promela language, however, this is also not mentioned in the original grammar.

The original grammar shows that a Declaration can be preceded by a Visibility term which can be either "hidden" or "show", however the manual pages show that "local" is another possibility for it. The manual pages show that besides a "proctype" declaration it is also possible to have a "D_proctype" declaration that will execute a non-deterministic sequence. The remote reference to a label in a process (which can only exist in a never trace) is available in the original grammar, however the remote reference to a local variable of a process is not available. All these parts have been added to the SDF grammar and the manual pages have been used to make sure those parts were added correctly.

The Promela language allows the use of macros in its models, but SPIN does not deal with them itself. SPIN calls the C-preprocessor to deal with the macros. In order to get the same result as SPIN, I propose to use the C-preprocessor before parsing a model as well. This way, the result will be the same as what SPIN uses.

There are few parts in SPIN that are only used for verification purposes, such as the trace and the never claim. Because the parser is to be used in a compiler and not in a verifier then that means that such parts are actually unwanted. This has been represented by recognizing those parts in the SDF grammar as unwanted terms. In the SDF grammar the trace and never claim will look like this:

```
UnwantedModule -> Module {cons("Module")}
Never -> UnwantedModule {cons("UnwantedModule")}
Trace -> UnwantedModule {cons("UnwantedModule")}
```

Other unwanted parts are the parts that have embedded C code. The embedded C code will not be part of the Promela2Java compiler *for now* because of the time constraints on this project. Those parts will look similar to the unwanted verification parts. When there is a time to add C code to Java transformation on this, then the unwanted recognition can easily be undone by changing `UnwantedModule` (in this case) to `Module` at the rule that is no longer unwanted. This way the parts that are no longer unwanted will be recognized as that but those parts that still are unwanted will still be recognized as unwanted.

Table 3.2 Operator Precedence, High to Low

Operators	Associativity	Comment
() [] .	left to right	parentheses, array brackets
! ~ ++ --	right to left	negation, complement, increment, decrement
* / %	left to right	multiplication, division, modulo
+ -	left to right	addition, subtraction
<< >>	left to right	left and right shift
< <= > >=	left to right	relational operators
== !=	left to right	equal, unequal
&	left to right	bitwise and
^	left to right	bitwise exclusive or
|	left to right	bitwise or
&&	left to right	logical and
||	left to right	logical or
-> :	right to left	conditional expression operators
=	right to left	assignment (lowest precedence)

Figure 2.5: The order of priorities of expressions from [19]

An order of priority of the operators in expressions is described in [19] as can be seen in figure 2.5. A context-free priorities part is added to the SDF grammar to make sure that the order of the expressions is the same for the SDF grammar as in the original grammar. This part groups the rules with the operators that have the same priority and puts a “greater then” symbol (>) between two groups of which the left side has the higher priority. A small part of this can be found in Figure 2.6.

```
context-free priorities
   {left:
  AnyExpr "*" AnyExpr      -> AnyExpr
  AnyExpr "/" AnyExpr      -> AnyExpr
  AnyExpr "%" AnyExpr      -> AnyExpr
  }
 > {left:
  AnyExpr "+" AnyExpr      -> AnyExpr
  AnyExpr "-" AnyExpr      -> AnyExpr
  }
```

Figure 2.6: The order of priorities of expressions in the SDF grammar

2.3 Difficulties

The Stratego toolset makes it easy to create a parser for the Promela language. The hardest part was to deal with ambiguity. There are a few parts in the original grammar that caused a bit of confusion when the counterpart was implemented as SDF grammar and then parsed. For instance, brackets are allowed around an `Expression`, which itself can be recognized as an `Any_expression`. However, an `Any_expression` can have brackets around itself as well, which leads to an ambiguity when a `Any_expression` with brackets around itself is parsed. The parser cannot understand if the brackets belong to the `Expression` or to the `Any_expression`. This was solved by removing the brackets around `Expression` as a possibility and adding optional brackets around the parts that can only be an `Expression` and not as an `Any_expression`.

The manuals that are available for Stratego [9, 31] were very helpful. These contained many examples to learn from, such as the priority handling and the way deal with comments. Any embedded C code in a Promela model is marked by keywords in Promela and should be marked as unwanted in the SDF grammar. The way to deal with comments in the examples looked like an ideal way to deal with the embedded C code.

Chapter 3

Transformation Rules

The previous chapter described how the parser can transform a Promela model into ATerms. These Aterms need to be transformed into Java code. The transformation rules do just that and will be described in this chapter. I have created a number of Java files that will be the same for every Promela model that will be transformed. This is done in order to get a full working Java application from a Promela model. These Java files can be found in Appendix A.3 and will be described in this chapter as well.

3.1 Java Files

In my research [30] I showed which Java classes could be generated from a Promela model in order to get a working Java application. As shown in Figure 3.1 the Java code produced by the Promela2Java compiler consists of three parts:

- Standard Java library packages,
- A fixed part containing the scheduler and related classes (created once), and,
- A variable part depending upon the Promela model that is to be translated (generated every time).

3.1.1 Classes Created Once

The papers [22, 32, 33] showed that a good idea is to use a scheduler in order to deal with atomicity and non-determinism. Such a scheduler would only need to be created once, because a scheduler will always need to deal with these properties in the same way. For that reason I have created the class `Scheduler`. The `Scheduler` and all classes that have been created can be found in Appendix A.3.

The `Scheduler` will extend from the Java class `Thread`. This class will contain all instances of the various `Proctype` classes during run-time. The `Scheduler` will give all `Proctypes` a unique number, which is equivalent to the *_PID* variable in the *proctype* of a Promela model. The class `Scheduler` has a special function that can halt the execution of

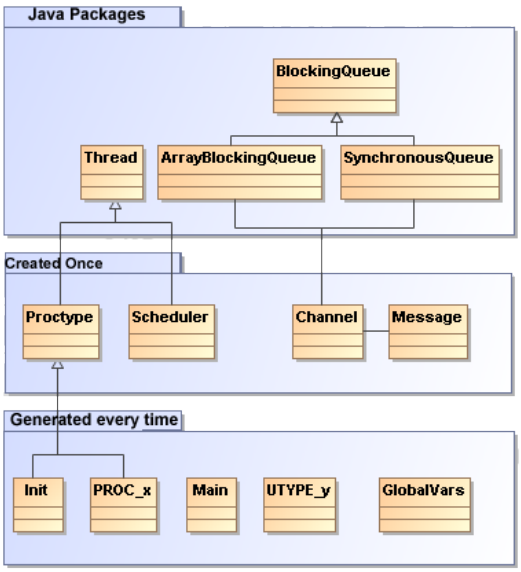

Figure 3.1: Overview of all classes

all other `Proctypes` in order to let one of the `Proctypes` be the only one that is executing at that time (atomicity). The `Scheduler` will also have a function that gets an array of boolean variables and will choose randomly between one of the values that is true. The number of the place of the chosen value in the array will then be returned to the `Proctype` that called the function. This `Proctype` uses the returned value to execute the correct path.

The *proctypes* of a Promela model will transform into a Java class that inherits from the class `Proctype`. This class contains methods to pause and resume execution enabling the `Scheduler` to handle atomicity. The `Proctype` itself inherits from the Java class `Thread`, just like the `Scheduler`, in order to execute separately.

The class `Channel` will represent the *channels* in Promela and will contain all the methods available in Promela as *channel* operations. This will mean that the methods will also block the execution of a `Proctype` in the same way as in a Promela model. The `Channel` will use the Java class `ArrayBlockingQueue`, because this class will make sure that a send or receive will be blocking if the buffer is full or empty. The `SynchronousQueue` is used to represent a *rendez-vous channel*, because this class will block a send until another receive is available and the reverse is true as well, just like a *rendez-vous channel* in Promela. C. Pronk shows in [27] that the blocking will be done with the Meta-locking algorithm described in [1]. The `Channel` class will be constructed with a number that tells how big its buffer will be and another number that indicates the size of the *messages* that can be send. A buffer of size zero will be a special *rendez-vous channel*.

The class `Channel` will use the class `Message` to fill its buffer with properly typed messages. This class `Message` will hold a `Vector` with all the parts of the actual *message*. This class also has an equality function to see whether the parts of the two messages are equal. In Promela it is possible to not let all parts match, but only a few parts of the message,

this is considered in the equality function.

3.1.2 Classes Generated Every Time

The class `Main` will be the class containing the `main` method. This method will start the `Scheduler` and the *proctypes* that are supposed to be active from the start (including the initial one). The global variables will reside in the class `GlobalVars`. The class `Main` and the class `GlobalVars` will be different for every new Promela model that is transformed into Java code. The `Main` class will make sure that all active `Proctypes` use the same instance of the `GlobalVars` and `Scheduler` class.

The data types of the Promela language will be represented by their equivalent Java data types, except for those not having an equivalent type. This is the case for the *mtype, pid, bit, unsigned* and every user defined type using the *typedef* construction. The *mtype, pid, bit* and *unsigned* data types will be represented as `integers`. A separate class will be generated for each user defined type in the Promela model that is being transformed.

The *mtype definition* in Promela will be transformed into separate integers inside the class `GlobalVars`. Each integer must have an unique value and when an *mtype* is printed then the symbolic name should be given in order to get the same kind of properties as an *mtype* in Promela. When an instance of the class `GlobalVars` is created then the constructor of the class will start filling the *mtype* integers with their unique values. A function is added that returns the symbolic name that is used for printing. The integers inside the class will have the prefix "`MTYPE_`", which has been done to ensure that there are no name clashes with Java keywords.

In Promela the values of the *mtype* are given to them in the reverse order as those were defined. However a second set of definitions for the *mtype* will result in a similar reverse order but none will have values below the first set of definitions. An example can be found in Figure 3.2. The values of the *mtype* ordering will start counting from one and thus will the class `GlobalVars` start by counting from one when the integers representing the *mtype* values are filled.

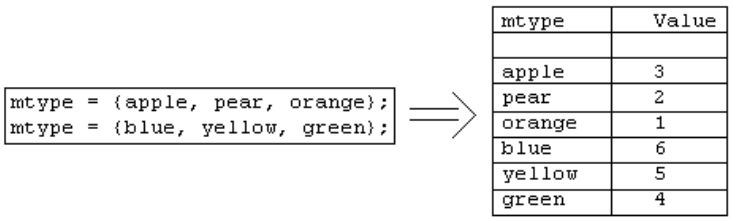

Figure 3.2: An example of the ordering of mtype values

When looking at the properties of a Promela *mtype* in SPIN [17, 19], then it is easy to see that these properties are very similar to the properties of the Java class `enum`. However, there are a few problems when this solution is chosen. For instance there is the possibility in SPIN to have a variable that is of type *mtype* but does not have the same value as any of the *mtype* values. For example, when all the *mtypes* are defined as `mtype = {apple, pear,`

orange}, then a variable definition `mtype var = 99;` is allowed. An `enum` variable is not allowed to have a value that is larger then the number of possibilities. This larger value can in SPIN even be used in expressions and when the *mtype* is printed then not the symbolic name is printed, (because there is no corresponding name to that value) but just the value of the *mtype* variable is printed. The function I have described above also has this feature. These problems are the reason why I have chosen to use integers instead of an `enum` class.

A *user defined type* is created by using the *typedef* statement in Promela, which will be transformed into a separate Java class. The member of such a *user defined type* will be represented as local variable of its class. The name of that class will always begin with "`UTYPE_`" and followed by the name of the *user defined type*. This prefix is used in order to prevent any name clashes with Java keywords. For instance, Promela allows to call your *user defined type* by the name "class", but it is impossible to use the name "class" as a name for a Java class.

All the *proctypes* and the *initial* process in a Promela model will be transformed into separate classes. A *proctype* in a Promela model will result in a Java class that inherits from the Java class `Proctype` that was described before. The name of these classes will have as a prefix "`PROC_`", again to prevent name clashes. If a `Proctype` can be called with arguments then there will be two constructors generated. One that will only set the `Scheduler` and the global variables and one that will do the same plus setting local variables that will be generated as well. These classes will overwrite the `run` method and specialize that method with the body of the *proctype* of the Promela model.

3.2 Strategies and Rules

More information on how to create transformations is necessary in order to generate the Java classes that were described above. The transformation rules are used by the Stratego toolset to create the part of the Promela2Java compiler that makes the actual transformations [9, 31]. The transformation rules can consist of one or multiple modules. Each module exists in a file and are all linked together using import statements. These modules can contain either strategies, rules or both.

Strategies are used to apply certain rules in a certain order, which rules and what order depends on the chosen strategy. A number of predefined strategies are available in the Stratego toolset. An example of a strategy is the predefined strategy `outermost(s)` which walks down a ATerm tree applying the strategy `s`, starting with the outermost ATerm and going inwards. An example for strategy `s` could be the strategy `try(r)` in which `r` is an existing rule. This rule `r` will then be applied to every ATerm that the outermost strategy looks at.

A rule will always be preceded by its name and will consist of at least two parts. The first part is the ATerm that is recognized, the second part of the rule is the ATerm that the recognized ATerm will be transformed into. The two parts are separated by an arrow (`->`). Variables are allowed to be used in the left side of the rule that can be reused on the right side of the rule as well. An ATerm starts with an upper case letter and a variable with a lower case letter. An example of a rule can be found in Figure 3.3.

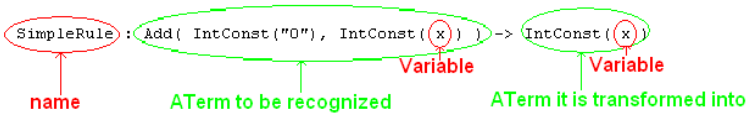

Figure 3.3: A simple example of a transformation rule

This rule transforms an `Add` ATerm of which the first element of that ATerm is the zero integer constant and the second element can be any other integer constant. The rule transforms the entire addition into the second element. As an example the ATerm

```
Add( IntConst(0), Add( IntConst(0), IntConst(34) ) )
```

will be transformed into the ATerm `IntConst(34)`. This is because the part of the ATerm that matches to the first part of the rule is `Add( IntConst(0), IntConst(34) )` which results in the ATerm `IntConst(34)`. That part has now been transformed and has been put back in the original ATerm and results in `Add( IntConst(0), IntConst(34) )`. But this part can be matched to the first part of the rule as well and will thus be transformed into `IntConst(34)`.

A `where` clause is allowed in a transformation rule to let the rule behave in a more specific way [9, 31]. Such a `where` clause will contain one or more strategies that all either can succeed or fail. A rule with a `where` clause will only be applied when all the strategies in the `where` clause succeed. The outcome of a certain strategy can be bound to a variable. As an example we can look at Figure 3.4.

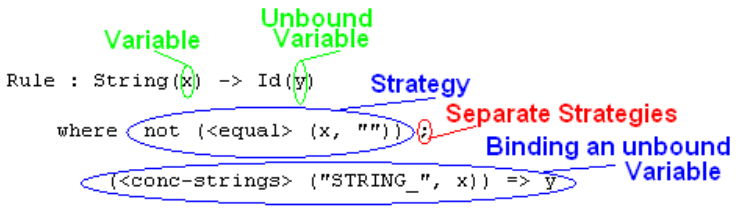

Figure 3.4: Another simple example of a transformation rule

The first strategy in that `where` clause makes sure the entire rule will only be applied when the predefined strategy equal fails. This means that the rule from Figure 3.4 will only be applied when the ATerm that will be matched to variable `x` is not the empty String. The different strategies inside a `where` clause will be separated by a ';'. The second strategy binds the outcome of the `conc-strings` strategy to the unbound variable `y`. The `conc-strings` strategy is a predefined strategy that concatenates two strings into one.

The rule will thus change every occurrence of the ATerm `String(x)`, where `x` can be anything, and transforms the term into the ATerm `Id(y)`, where `y` will be the word "`STRING_`" concatenated with `x`, unless `x` is equal to the empty string, then the term will not be transformed at all. Strategies and rules can be used to transform the ATerms of a parsed

Promela model into different ATerms, but a code generator is necessary in order to get Java code.

3.2.1 Java-front for Stratego

Stratego has a Java-front library [31] to make transformations on Java code [24]. This Java-front comes among other things with two tools that can help a lot during this project. The first tool can take in a Java file and creates ATerms from that file, in other words that tool is a parser for Java files. The other tool takes in ATerms and creates Java files from the ATerms, which is thus the code generator or as it is called in Stratego a pretty-printer. This pretty-printer will be used directly as the back-end in the Promela2Java compiler that will be created for this project. This Java pretty-printer can only take in ATerms that represent parts of the Java language in order to get correct Java code. The Java parser can be used to get to know what parts of the Java language corresponds to which ATerms by creating small Java files and parsing them.

In the previous sections I described how transformation rules are created. The transformation rules can simply have the right hand side as terms that are used to represent the Java code. Figure 3.5 shows an example how a Program ATerm would result into a simple Java class `Main` with an empty body.

```
Rule : Program(x) ->
         ClassDec( ClassDecHead( [],
                                 Id("Main"),
                                 None,
                                 None,
                                 None
                               ),
                   ClassBody([])
                 )
```

Figure 3.5: A simple example of a transformation rule into a Java class

Actual Java code can be used inside a transformation rule. This should be preceded by the name of the part of the Java language that is represented. An example of how this is done for a simple class can be seen in Figure 3.6. A class can be represented by embedded Java code in a transformation rule and should therefore be preceded by its name which is just like the previous figure the `compilation-unit`. This way of using actual Java code can be used for a number of different parts of the Java language, such as a block statement, a statement or an expression.

```
Rule : Program(y) -> compilation-unit |[ class ~x:name{} ]|
          where (<conc-strings> ("CLASS_", y)) => name
```

Figure 3.6: Another simple example of a transformation rule into a Java class

Unbound variables can be used to change parts inside the Java code, such as the name of the class in Figure 3.6. In this case the transformation rule will bind the ATerm, that will be matched, to variable y and concatenate it with the string "`CLASS_`" as the name for the Java

class. The unbound variable name in the example is preceded by an "∼x:" , this is to let the transformation rule know that the token that follows will be an ATerm (or variable). The letter x indicates that the following part will be a string, for other parts of the Java language there are other letters or letter combinations that can be used, such as an e for an expression or an i for decimal integer.

3.3 Promela to Java Strategies and rules

The transformation rules that are created for this project need to create Java ATerms from the ATerms that were generated with the parser as was described in the previous chapter. The generated Java files will be the ones that were described earlier in this chapter as the ones that need to be generated every time. The files with the transformation rules can be found in Appendix A.2.

3.3.1 Preparing Strategies

This section will describe those parts that are needed to prepare (the ATerms generated from parsing) the Promela code for the actual transformation into Java code. These preparing strategies will transform the ATerms into different ATerms that have nothing to do with any ATerms that represent Java code.

Inline

To make sure the Promela model will correctly be transformed is to deal with the Promela *construction inline* first. Such a *construction* consists of a *inline definition* and one or more *inline calls* to that *inline definition*. A Promela model can have multiple of these constructions. Every *inline call* contains the name of a *inline definition* and a sequence of variables. Every *inline definition* contains its name, zero or more parameters and a sequence of statements in which the parameters are incorporated. For every *definition* these elements are different.

SPIN [17, 19] deals with an *inline construction* by replacing an *inline call* with the sequence of statements that reside in the *inline definition* in which the parameters of the *inline definition* have been replaced by the defined variables of the *inline call*. A rule that transforms an *inline call* to a sequence of statements is necessary in order to accomplish the same in Stratego. However, a rule is always the same and the statements that replace the *inline call* are different for every *inline definition*. This can be solved by using dynamic rules [8].

Dynamic rules are created inside normal rules by adding a strategy to the where clause that is called `rules`. These dynamic rules can then use parts of the rule that has created them. In the case of an *inline construction* there should be a normal rule that matches an *inline definition* with its name and and sequence of statements and transforms that definition into a checked *inline definition*, so that this definition will not be matched to the same rule again. The where clause of that rule will create a dynamic rule that matches a *inline call*

with the same name and transforms the call into the sequence of statements that was in the *inline definition*.

If there are any parameters defined in the *inline definition* then the sequence of statements need to transformed as well before inserting the statements at the place where the *inline call* was. When all the dynamic rules have been created and thus when the strategy succeeded then the dynamic rules still need to be applied by using another strategy by calling all the newly created dynamic rules. A simplified version of how this is done can be found in Figure 3.7. The renaming strategy creates a new sequence in which the old arguments are transformed into the new ones.

```
strategies
    inlineHandling = try(Handle + ChangeCalla + ChangeCallb); all(inlineHandling)

rules
    Handle : InlineModule(name, seq) ->  CheckedInlineModule(name, seq)
             where
                rules( ChangeCalla : InlineCallStatement(name) -> seq    )

    Handle : InlineModule(name, oldargs, seq)-> CheckedInlineModule(name, oldargs, seq)
             where
                 rules( ChangeCallb : InlineCallStatement(name, newargs) -> newseq
                             where (<renaming> Replacement(oldargs,newargs,seq))=> newseq
                 )
```

Figure 3.7: A simplified version of Inline dynamic rules

Working with the dynamic rules has given me a better look of Stratego and made me look at other uses for the dynamic rules. And I did found another use for them, type checking.

Type Checking

Every variable in a Promela model has a certain type. There are nine predefined types in Promela, namely: `int`, `short`, `bit`, `byte`, `pid`, `unsigned`, `bool`, `mtype` and `chan` (for channels). Defining your own type structures is allowed in a Promela model, by using the C style macro `typedef`. These type structures will from now on be addressed as user defined types. An array of variables can also be created. In Promela it is possible to assign a value of certain type to a variable of a different type, this is because the value will automatically be converted to the correct type. This is the case for all types, except for the channels, the arrays and the user defined types.

The type of a variable is known when it is declared. However, there is no indication of the type of a variable when one is referenced. This is because the only thing that is necessary to reference a variable is its name. Type checking is used to make sure that a variable reference is transformed into Java in the correct way. For instance, a `bool` variable can be used in the assignment of an integer. The transformation should make sure that the Java code does not assign the integer value to that variable but the value of equality of the integer to zero. This example can be seen in Figure 3.8.

The transformation rule needs to know the type of the variable that is referenced in order to assign the correct value to the variable. This can be done by using dynamic rules that are

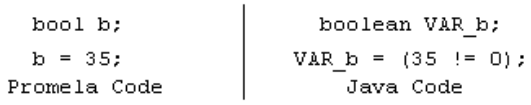

Figure 3.8: A simple transformation for a variable reference

similar to those rules that deal with the *inline constructions*. A dynamic rule will be created in the transformation rule that matches a variable definition and takes from that definition its name, its type and its scope. The name will be used to match all variable references for that variable and change them into checked variable references that contain the same information as the original variable reference, but also two extra pieces of information: the type and the scope of the original variable declaration.

The scope of the checked variable reference will be string that can have one of three values: "*local*", "*global*" or "*mtype*". The scope is necessary to find out how to reference the variable in Java. A global variable will reside in an instance of the class `GlobalVars` as was previously described. The way to reference to such a variable in Java will look like `global.VAR_name`. The local variables will reside inside a `Proctype` and thus do not need any prefix and can be referenced by its name alone, like `VAR_name`. The "*mtype*" scope indicates that not a variable is referenced but actually a specific mtype is referenced. The mtypes are represented in Java as integers as previously described that reside in the class `GlobalVars` as well. The mtype reference will look like `global.MTYPE_name` in Java.

The complete type checking strategy in the files that can be found in Appendix A.2. Everything that has been described so far has not resulted in any Java code. Preparing the Promela code for the actual transformation into Java code is all that has been done. These transformations will be described in the next few sections.

3.3.2 Promela to Java Rules: P2J

This section will describe how the ATerms, that resulted from the original Promela code and already transformed by the preparing strategies (which from now on will be referred to as the original ATerms), will be transformed into Java code. A Promela model needs to be transformed into a Java class called `Main` and a class called `GlobalVars` and for every user defined type and process (*proctype* or *init*) that is inside the Promela model there should be a separate class. This means that the original ATerms should be separated into four parts and will later on be joined together.

The Main Class

One of the parts that the original ATerms will be transformed into, will result in a Java class `Main`. The rules that are associated with the class `Main` will change the original ATerms into a `CompilationUnit`, which is the ATerm representing a Java class in Stratego. This class will initialize an instance of the Class `Scheduler` and an instance of the class `GlobalVars` and there is also a part that will make an instance of all active `Proctype` classes, if any are inside the model.

The generated code will look similar to the code in Figure 3.9. The loop that can be found inside the figure is there to facilitate the option when multiple instance of the `Proctype` need to be active from the start. In Promela this is done by adding a number between square brackets after the word *active*. In Java this will be done by using a `for` loop around the instance creation of the `Proctype`.

```
public class Main
{
  public static void main(String[] args)
  {
    Scheduler scheduler = new Scheduler();
    GlobalVars global = new GlobalVars();
    {
      for(int a = 0; a < 1; a++)
      {
        scheduler.addProctype(new PROC_process(scheduler, global));
      }
    }
  }
}
```

Figure 3.9: An example of a class Main.

The `Proctype` is an extension of a `Thread` and can execute on its own and is started inside the `Scheduler`. The order of execution will be chosen by Java, because that is how `Threads` work in Java. The instance of a `Proctype` will always have the earlier created instance of a `Scheduler` and the earlier created instance of the `GlobalVars` class. This is to ensure every process will use the same global variables and scheduler. The description of the `Scheduler` can be found earlier in this chapter, but the class `GlobalVars` will be described next.

Global Variables Class

The class that will contain all the global variables will be called `GlobalVars`. This class will contain all variables that have been globally declared in the Promela model, but will also contain the *mtype* variables and a function to get their symbolic names. The class is constructed by using rules that filter out the *proctype* ATerms, the *user defined type* ATerms and the *inline definition* ATerms in such a way that only global declarations and *mtype* declarations remain. The global declarations and *mtypes* will then be transformed into `FieldDeclarations`, which is an ATerm used by Stratego to describe a variable of a local class. The *mtypes* will be transformed into integers as has been described earlier in this chapter.

A similar filtering rule is applied on the ATerms to get the global declarations and *mtypes*, in order to initialize the values that needs initializing, such as arrays. This initialization will be done inside the constructor of the class to ensure the correct values of the variables. Then a separate function is created that returns a String of the name of an *mtype*. An example of a class `GlobalVars` that has been generated from a Promela model that has a few *mtype* declarations and a few global declarations can be found in Figure 3.10.

```
class GlobalVars
{
  int MTYPE_orange = -1;
  String STR_MTYPE_orange = "orange";

  int MTYPE_apple = -1;
  String STR_MTYPE_apple = "apple";

  int VAR_a = 0;

  int VAR_b[] = new int[5];

  public GlobalVars ()
  {
    int nrofmtype = 1;
    {
      MTYPE_orange = nrofmtype;
      nrofmtype++;
      MTYPE_apple = nrofmtype;
      nrofmtype++;
    }
    { }
  }

  public String getStringMtype(int mtype)
  {
    {
      if(mtype == MTYPE_orange)
        return STR_MTYPE_orange;
      if(mtype == MTYPE_apple)
        return STR_MTYPE_apple;
    }
    return "" + mtype;
  }
}
```

Figure 3.10: An example of a class GlobalVars.

Proctype Classes and a Class Init

A *proctype* and an *init* are pretty similar in a Promela model, both are processes and only differ in the fact that an *init* will always have one (and only one) instance at the start of a simulation run of a Promela model. The *init* can never be instantiated more then once during a simulation run. This means that the generated class will be the same as the one for a *proctype*, but with a different name. From now on when I describe a *proctype* I will also mean the *init*.

The generated class will extend from the earlier described class `Proctype`. The generated class will also have a constructor that sets the `Scheduler` and the global variables. Another constructor is added if a *proctype* in the Promela model has parameters, these parameters will be added as local variables similar to the declarations inside the body of a *proctype* and the constructor will fill those variables with the given values to the constructor.

The generated class will also have a `run` function that will execute the statements that

can be found inside body of the *proctype* in a Promela model. Those statements will be surrounded by a `while` and a `switch` to ensure the correct control flow when a *goto* statement occurs. This has been done in the same way as has been described in my research paper [30]. The `switch` has several cases, the statements are separated in such a way that each case will contain all statements before a *label* statement occurs and the statements that are located behind *a label* statement will be in the next cases of the switch.

The declarations that are made in the body of a *proctype* will result in local variables, that are similar as the global declarations for the class `GlobalVars`. However, if there is an initial value for the variable, then that value is not set in the constructor, but the declaration itself will be replaced by an assignment to ensure the correct value at the correct time.

Every statement that can occur in a *proctype* of a Promela model will have a different Java transformation. A small example of a generated class of a *proctype* can be found in Figure 3.11. The original Promela code that belongs to this example is: `proctype(int n){int a = 0; int b = 1;}`.

```
class PROC_process extends Proctype
{
  public PROC_process(Scheduler scheduler,
                      GlobalVars global, int VAR_n)
  {
    super(scheduler, global);
    this.VAR_n = VAR_n;
  }
  int VAR_n = 0;
  public PROC_process (Scheduler scheduler,
                       GlobalVars global)
  {
    super(scheduler, global);
    { }
  }
  Vector path = new Vector();
  int index = 0;
  boolean going = true;
  int choice = 0;
  final int first = 0;
  boolean labelset = false;
  int VAR_a = 0;
  int VAR_b = 0;
  public void run()
  {
    while(going)
    {
      labelset = false;
      {
        switch(choice)
        {
          case first:
            {
              canExecute();
              synchronized(scheduler)
              {
                VAR_a = 0;
              }
              canExecute();
              synchronized(scheduler)
              {
                VAR_b = 1;
              }
            }
        }
      }
    }
  }
}
```

Figure 3.11: Generated Java code for: `proctype(int n){int a = 0; int b = 1;}`

Statements

All the statements inside the body of *proctype* in a Promela will transform into a Java statement inside the `run` function of the corresponding Java class that is generated from the *proctype*. Declarations in the body of a *proctype* are transformed into a class variable and an assignment statement, as can be seen in Figure 3.11. The `while` and `switch` that surrounds all other statements of the `run` function can also be found in that figure. These constructions are there to ensure the correct workings of the *label* and *goto* statements.

User defined types

A user defined type is a type that is constructed in a Promela model by using the *typedef* statement. The Java transformation will result in a separate class for each use of a *typedef* statement. The *typedef* statements contains one or more variable declarations. These declarations will be added to a new class in the same way as the global variables are added to the `GlobalVars` class. The declarations will be initialized in the constructor of the class in the same way as the global declarations were initialized in the class `GlobalVars`. Because these classes are similar, these are in fact, generated by some of the same rules to get the same result. An example of a class generated from a user defined type can be found in Figure 3.12.

```
class UTYPE_uu
{
  int VAR_bb = 0;
  int VAR_cc = 0;
  public UTYPE_uu ()
  {
    { }
  }
}
```

Figure 3.12: An example of a class generated from a user defined type.

3.4 Difficulties

The hardest part for me was the fact that Stratego was all new to me, I had never worked with Stratego before. But I did have some help from the people that worked a lot longer with Stratego than I had. One of the first big problems I found, in the case of the transformation rules, was with the type checking. I had no idea how to pass on information from the place where a variable was declared to the place where the variable was used. After some help, I discovered the dynamic rules [8], which were very useful. After I had used those dynamic rules for type checking I found that I could use them for dealing with the *inline* constructions as well.

Other difficult part was the size of the amount of work. I underestimated the amount of time that would take to implement all transformation rules for all the different Promela statements. For instance the amount of work that came into trying to transform a combination of a *unless* statement and an *atomic* statement, which resulted in a separate file of

approximately *1370* lines of code. These transformation rules do, however, look very similar to those rules that were used for the transformation of the *unless* statement and the *atomic* statement separately, but that does show how much is needed when one kind of statement is combined with another kind of statement.

I know that it is human to make a mistake and I probably made a few in the process of creating all the transformation rules, the Java code or even in the SDF grammar. Therefore, there will always be the need for testing, as will be described in the next chapter.

Chapter 4

Testing

This chapter will describe what kind of tests I have been doing. I will also describe what has resulted from those tests.

I should have started testing much earlier then when I did start. What I should have done was create test cases before I even started creating any code for the parser at all, just as is described in [5]. These test cases could then have been executed at the end of each day to see if I was not making (much more) mistakes.

I did however parse Promela models that I have found on the world wide web even before I created any test cases of my own. But these tests are not enough, I also made special test cases to make the compiler as close enough to error-free as possible.

4.1 Promela from the World Wide Web

I can simulate the intended use of the compiler, by parsing and transforming all kinds of complete Promela models. Many people can look at a language and use very different ways to expose parts of a compiler or simulator in such a way the creator of such a compiler or simulator never thought was possible. That is why it is very important to write as many tests as possible by just as many different people. The best way to do that is to not let them write actual test cases, but let them use the language for their own problems and see what kind of models will be created.

The tests that I use, to see if my compiler works correctly, came mostly from the world wide web and [26]. This is to ensure as many different people that write tests as possible. some of the models that I used were from my supervisor and from a fellow-student who is working with Promela for his thesis as well. I was amazed that there was still so much possible that I had not considered when I looked at the Promela language. For instance, Promela code may contain two ‘;’ characters right after each other or use arrows ‘->’ instead. I did not have that possibility accounted for in my parser at first, but that possibility is incorporated in the parser right after I came across this problem in some of the tests.

Another problem was with the fact that the channel functions “empty”, “nempty”, “full” and “nfull” are allowed to be used within an “&&” expression or an “||” expression. These are the only two binary expressions in which these channel functions are allowed, but I did

not have that implemented in my parser. Now that I have located this problem, I could easily add this to the parser. However, the impact on the transformation rules was bit larger than expected and had to be dealt with as well.

4.2 Test Cases

There are three parts of the compiler that need to be tested separately: the parser, the transformation rules and the Java files. The parser can be tested by creating test cases in the form of Promela models and use them as input for the parser. These test cases will fail if the parser reports any problems or if any ambiguous parts occur within the parsed results.

The transformations can be tested by taking test cases in the form of Promela models and compile these models. The same test cases could be used as the ones used for testing the parser. Compiling the models will mean first parsing them an then transforming them, the parsing step can be skipped by taking the results of the parser test cases and immediately transform them. This can, of course, only be done if the results were correct. The transformation test cases will fail if no Java code is generated or if the transformations report any problem.

The Java code that is generated from the Promela models will need to be tested as well. The results from the transformation test cases are the Java code that needs to be tested. The Java files that remain the same for every Promela model (`Scheduler`, `Proctype`, `Channel` and `Message`) need to be tested as well. For these Java classes there can be some separate test cases to ensure those classes work correctly as well. These test cases are, however, not included in this document, because these would take up too much space.

The Promela2Java compiler relies on the description of the Promela language in [17, 19]. This description of the Promela language is not complete, there are a few small things missing, I cannot anticipate on all those missing descriptions. The Promela models that were found on the world wide web did help a bit in this regard and showed me the first missing bit in the Promela language description. There are two ways to write comments in Promela, by `/*comment*/` or by `//comment end-of-line`. The latter of the two was not described in [17, 19] but is allowed in Promela and has been included in the parser as well.

4.3 Promela Test Cases

The test cases that are used for the parser and the transformations are Promela models. These Promela models are correct Promela models because the assumption is made that the Promela models will be used by SPIN (and are therefore error-free) before those models are used with this compiler. For now I have used [17, 19] and the grammar of Promela to see what kind of models can be created with the Promela language. These models will then be used by SPIN before those models are used as test cases.

Here I will describe the test cases I have created that are used for testing the parser and the transformation rules. The results will be described when all the test cases have been described. The tests have been devised by taking a look at the grammar and try to use all possible test cases for a part of that grammar.

Proctype Test Case

The `proctypes` are tested by a Promela model having a number of different processes that all contain one `print` statement as their body. All `proctypes` differ in activity and multiplicity at the start of the simulation of the model. Another difference is that a `proctype` can have a list of parameters and a `priority` or `provided` part. This results in twenty-four different `proctypes` and all are started in an `initial` with `run` statements.

User Defined Type Test Case

The user defined types are tested by adding a few to a Promela model and making them different in their declaration list which is used by the `typedef` statement. A user defined type used inside another user defined type is also one of the possible test cases. The `mtype` declarations are tested as well, by using both ways of declaring `mtypes` and by changing the values of the `mtype` variables.

Declaration Test Case

Every declaration has an optional visibility, a type, a name, optional square brackets with a constant and an optional initializing expression. The expression is every time a constant number in the test case but all possible combinations of optional and required parts are used in the test cases for the declarations. Whenever a combination is not allowed in the Promela language, for instance, an array of `unsigned` variables, then that combination is not included in the test cases.

Atomic and d_step Test Case

The `atomic` statement and the `d_step` statement are tested by making two `proctypes`. One `proctype` will contain the `atomic` or `d_step` and tries to print the value of a variable two times, whilst another `proctype` will try to change the value of that variable. When the Promela model is either simulated or executed, then the same value should be printed twice. An `atomic` can also be blocked half way, so this is tested as well by another `atomic` that again prints the value of the variable twice, only half way there the `atomic` sequence will be blocked and restarted after the variable has changed before printing the second time.

If and Do Test Case

The `do` and the `if` statement are tested by having all different kinds of those statements. Test cases with one or more options, some missing '`;`' characters at the end. Some test cases will have `else` statements in the option followed by other statements and some have an `else` statement followed by nothing. The `break` statement will occur in a few of the `do` statement test cases as well.

Expression Test Cases

All the expressions that are possible have been tested by assigning an expression to a variable. This variable is then printed together with the expected value. This resulted in a few test cases for binary and unary expressions, but tests were also created for all other expressions. The most important tests that involve expressions are the ones that test the priority. The priorities of the expressions in Promela are given in [19] as can be seen in Figure 4.1. The test cases will see if the generated Java code abides to the same priorities as the original Promela code.

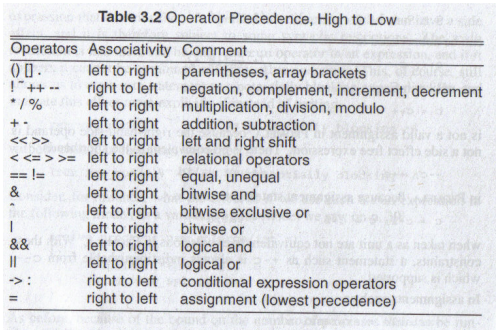

Table 3.2 Operator Precedence, High to Low

Operators	Associativity	Comment
() [] .	left to right	parentheses, array brackets
! ~ ++ --	right to left	negation, complement, increment, decrement
* / %	left to right	multiplication, division, modulo
+ -	left to right	addition, subtraction
<< >>	left to right	left and right shift
< <= > >=	left to right	relational operators
== !=	left to right	equal, unequal
&	left to right	bitwise and
^	left to right	bitwise exclusive or
|	left to right	bitwise or
&&	left to right	logical and
||	left to right	logical or
-> :	right to left	conditional expression operators
=	right to left	assignment (lowest precedence)

Figure 4.1: The order of priorities of expressions from [19]

Sending and Receiving Test Cases

Send and receive statements have been tested by sending a few message to a channel and then receiving those messages one by one. The idea is that what has been sent must be equal to what has been received and that will be shown via the print statement.

The receive statement will only be executed if all parts of the message that is to be received match with those parts that are expected. The parts that are expected are represented in a receive statement as a parameter list. This list can contain parts of which there is no expectation, which can be represented by a name of a variable or an underscore. A name of variable means that the value of that part of the message that is received will be assigned to the variable with that name, whilst the underscore means that the corresponding part in the message can be ignored.

The first test will test normal send and receive statements. Three messages are sent and then received with a receive statement, that has no part that needs to match. Both the normal send and receive statement on a buffered channel have a first-in first-out property which means that the order in which the message are received should be equal to the order in which the messages were sent. The same test is repeated only now there are parts in the receive statement that contain underscores or parts that should match up with the message that should be received.

A sorted send statement will make sure that the message that is send will be placed in the channel immediately before the message that is greater than the message that is sent. The sorted send statement is tested by sorted sending a few different messages in a specific

order. The receive statement will be a normal receive statement and the result should be that messages are received in a different but correct order.

A random receive statement can take any message in the channel that matches with the message that is expected. The random receive statement is tested by sending a few different messages in a normal way and then random receiving the messages and printing them. If these are executed a number of times then the order of the printed messages should become different over time.

Receiving a message without removing the message from the channel is also possible in Promela. This is tested by sending a message to a channel and by receiving the message without removing, more then once. This indicates that the message is indeed not removed. The same can be done for random receiving a message without removing the message from the channel. All the test cases that have used a channel have all been executed twice: once with a normal buffered channel and once with a rendez-vous channel.

Parts that were Left Out Test Case

The parts of the Promela language that are left out of the compiler as is described in Chapter 5 are also used in a couple of test cases. This is to ensure that those parts can occur in a Promela model but do not give any faulty result and will simply be ignored. These test cases contain embedded C code, the `xr` and `xs` statements, the `assert` statement, the `trace` and `notrace` modules and the `never` claim.

4.3.1 Parser Results

Problems that I found during this test phase were mostly ones that are extensions of the problems that I found when I ran test programs from the world wide web. The double '`;`' characters right after each other gave some other problems as well. The "&&" expression and "||" expression were also giving problems, especially if those statements were combined with different expressions.

4.3.2 Transformation Results

The first thing that stood out when I ran some of the tests, was the fact that the transformations on larger files take very long. This is because most of the transformation rules are all ordered in one kind of rule set. This kind of rule set was applied with the use of the `outermost` strategy, which is a built in strategy that walks over an abstract syntax tree and applies a rule from the rule set. This walk over the entire syntax tree is repeated until no more rules can be applied to any of the terms in the abstract syntax tree.

During a transformation parts occur in the abstract syntax tree that will remain the same until the end of the transformations, but the strategy does not know that yet and still walks over those parts in the abstract syntax tree to see if some of the rules can still be applied. This is somewhat inefficient and the reason for the compiler for being slow.

The problems that were found in the parser also resulted in problems for the transformation rules. This means that all transformation rules, that uses the problematic parts of the parser, needed to be fixed as well.

4.4 Transformed Promela to Java Test Cases

The test cases that will be described in this section will be test cases for the generated Java code. The generated code is from the Promela model test cases that were described earlier in this chapter. Each test case will refer to that model and the results of the test case will be described here.

Proctype Test Case

The Promela model involved all kinds of *proctypes* that were slightly different in the way those are created, but all had a print statement within their bodies to see if those *proctypes* are running. An *init* was there to make sure all *proctypes* were running. The test case for this model is not too hard, a simple execution of the `Main` class of the generated code to see if all print statements were executed correctly, which would mean that all classes generated from *proctypes* were executed. This was indeed the case and no problems were found.

User Defined Type Test Case

The Promela code that corresponds to the user defined type test case was a collection of various user defined types. Some of the types were nested and there were also a few *mtype* declarations. The test case should test some instances of the user defined type and change all the members and check if those were correctly changed in the corresponding generated classes. The *mtypes* should be checked to see if those have the correct value. The function that returns the name of the *mtype* should be checked as well.

The only problem that was found had to do with the initializing values of some of the user defined types. In the generated code the values were all set to zero, whilst this was not always the case. This could easily be solved by changing the rules that set the field declarations to zero.

Declaration Test Case

The Promela model that corresponds with this test case contains a lot of declarations. Each declaration differs a little bit from the others, a different type, a different visibility or the declaration can be an array declaration. The result is a massive file with approximately two-hundred different declarations, which are all global. The same number of declarations were also tested within a *proctype*, making the declarations local.

The best way to test this is to check if the value of a declaration matches the initializing value in the Promela model and then change that value into a bunch of different values and see if the value is cast to the correct value. However, testing this should take a long time, so for now I will simply inspect the generated class by hand to see if there are any problems. If there is enough time left then a correct test case can be written to test this.

A few problems caught my attention at once. The array declarations that were being filled were called with a wrong name the prefix "`VAR_`" was added twice instead of once. Another problem was similar to the one found with the user defined test case, the variables

that were initialized with a value did not have that in the generated code. This last problem was solved simultaneously with the problem of the user defined types, because both problems resulted from the same set of rules.

The other problem with the double prefix was found by following the rules to the place where a global array declaration with an initializing value was transformed. There are two kinds of declarations possible during a transformation, a checked or a not checked declaration. A checked declaration has among other thing its name changed with a prefix. When an array was filled with a value then a checked declaration was changed into a non-checked declaration, which means the prefix would be added later again. This was simply solved by not changing the declaration.

If and do Test Case

The Promela model that corresponds to this test case has several *do* and *if* statement, with either one or more then one optional paths. This test case should check if the correct path was taken. The easiest way to do this is to add `assert(true)` statements with the paths that should be taken and the paths that should not be taken should have the `assert(false)` statement.

The problem here was that a `while` loop was wrapped around the call to the `Scheduler` to choose a path. That loop waited until the chosen path did not equal zero anymore. The actual problem was that whenever the `Scheduler` chooses the first path then the value zero would be returned, which will not break the loop and thus the first path would have never been taken. The set of rules that would set this variable to zero were changed to minus one to solve this problem. Minus one is also the value that the `Scheduler` returns when there was no path executable.

Another problem was found, it was possible for processes to interleave between the evaluation of the guards and the execution of the guard in the Java code for the Promela *do* or *if* statement. This has now been solved by making the process atomic before the evaluation starts and the process will no longer be atomic until after the guard has been executed.

Atomic and d_step Test Case

The Promela model of the *atomic* and *d_step* test case consists of two *proctypes*, one tries to print the value of a variable two times whilst being atomic and the other *proctype* tries to change that value. The two print statements should print the same value, because the *proctype* was atomic at that time. An *atomic* statement can be blocking, so this was tested as well by adding a blocking statement in between the print statements, which makes sure that the two print statements do not print the same value anymore.

The test case will be a look at the output to see if everything worked correctly. There was a small problem, but that was because of the problem with the *do* and *if* statements that were used, other than that there were no problems.

Sending and Receiving Test Cases

The sending and receiving test cases were combined because these statements were tested by first sending a few messages and then trying to receive those messages. Every possible manner in which a message can be send (fifo and sorted) or received (fifo, random and both without removing) was tested. A few problems were found during the execution of these tests.

The first most obvious mistake I found was a typing error. A more interesting fault occurred when a message was about to be received, but a part of the message should have been matched to the value of a boolean variable. The fault occurred because the boolean value was not converted into an integer and thus would not match the integer that was inside the message that was supposed to be received. This was solved by changing the transformation rules that belong to this statement. The name of the variable was changed into an expression that has to have the type used for channel sending. An example of what went wrong and how that has been solved can be found in Figure 4.2.

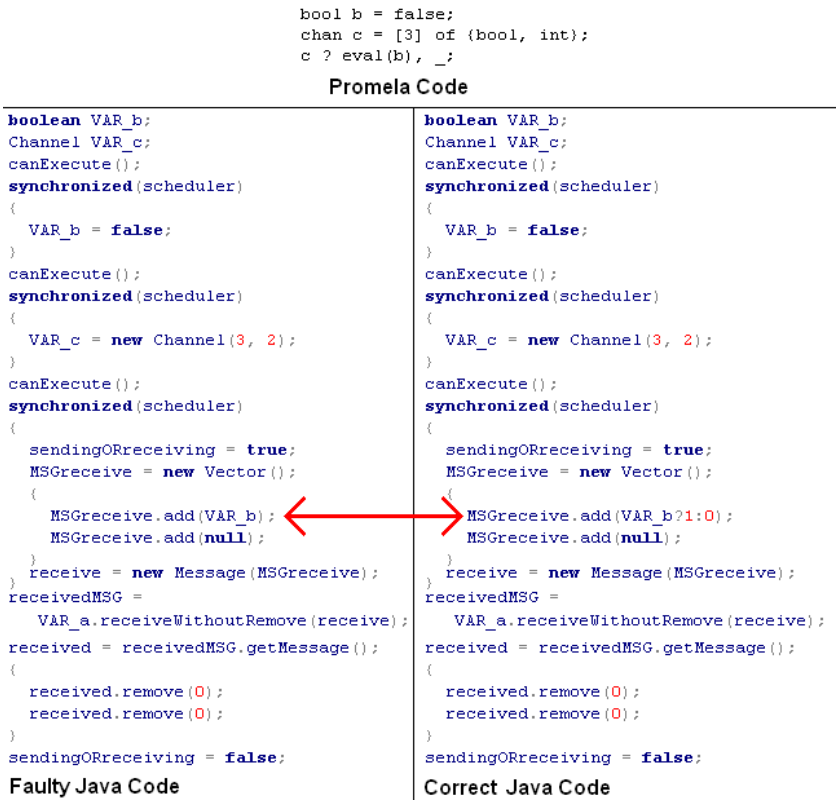

Figure 4.2: Finding the error in sending and receiving statements.

Another problem occurred with the receiving without removing the message from the channel. The problem was that the message was received and not removed, but the message that stayed behind had become empty. This has happened because the function had indeed

not removed the message from the channel but a reference to the message in the channel was returned instead of a duplicate of the message. This was simply resolved by changing the functions in the class `Channel` that had this problem by returning a clone of the message.

Expression Test Cases

The expression test cases were tests that involved all possible expressions. All went well except for the problem that was described earlier with the "&&" and "||" expressions and the channel operators *empty(chan), nempty(chan), full(chan)* and *nfull(chan)*. The problems occurred after some changes in the parser for these expressions. The problem was that a channel operation would have been evaluated twice which resulted in duplicate variable names and expressions that block an *if statement* when that should not be blocking at all.

Knowing now what the problem was I could easily spot where the problem occurred in the transformation rules. I could then change the rules easily to get no more duplicate evaluations for those kind of expressions. There were no problems found when the priorities of the expressions were tested.

Parts that were Left Out Test Case

The parts of the Promela language that are not implemented in the compiler can occur in a Promela model that needs to be compiled. The idea is that those parts will simply be ignored, so the corresponding Promela model will have a bunch of statements that will simply not occur in the generated code. The result should be a practically empty file, which can be checked by inspecting the class. The result was indeed a practically empty class which means that the missing parts are correctly ignored inside the compiler.

4.5 Conclusion

I do now, more then ever, understand the importance of testing. I did not have enough time to test the entire compiler thoroughly. Test cases should have been created in an early stage of the project. Then these tests could run every time something had changed and found out if the changes would not fail the test cases.

A lot of test cases resulted in a problem when executed for the first time. This emphasizes the fact even more that testing is one of the most important processes in the development of a system or application. Luckily, I could solve all problems that I found, but that does not mean that the compiler is completely error-free. There is still the possibility that there are errors in my test cases or that my test cases do not completely cover the entire Promela language.

It would have been nice to have automated test programs to help test the Promela models and the Stratego results with programs that look and act the same as JUnit [29] or Ant [13]. This would help a lot, but these kinds of programs do not exist just yet. I did however use JUnit to test most of the Java files, which was a great help.

Chapter 5

Compiler

This chapter will describe the parts that have intentionally been left out of the compiler and how the Promela2Java compiler can be used to create Java code from a Promela model.

5.1 Intentionally Left Out Parts

There are some parts of the Promela language that have not been incorporated in the compiler that was built. Parts of the Promela language that are only needed for verification were intentionally left out, other parts were left out due to lack of time.

Inside a Promela model embedded C code can be used, however, from a validation view point the use of such code is highly discouraged because those code fragments cannot be verified by SPIN. Inside C code fragments any C statement is allowed. Consequently, the entire C language would need to be parsed and transformed into Java in order to incorporate the embedded C code of the Promela language into the compiler. This was considered out of scope for the current project.

A Promela model can be verified by the SPIN program, this is usually the main goal when a Promela model is created. The Promela2Java compiler assumes that the Promela model was verified by SPIN before an executable application from the model will be created. The executable application is more of a simulation then a verification. Therefore, all the parts of the Promela language that only exist for verification purposes are not included in the compiler. These parts include the `never` clause and the `xr` and `xs` statements for the exclusive behavior of channels.

The visibility of variable declarations are also not included in the compiler, because these are not used in a simulation run. A variable declaration visibility can either be `hidden`, `show` or `local`. A `hidden` variable means that the variable is excluded from the global system state in the verification process of SPIN [17, 19]. A variable that is declared using `local` means that the variable is used as a local variable but declared globally in order to be used for verification purposes. A declaration that uses `show` as its visibility is being tracked in a message sequence chart displays in the Xspin tool [17, 19]. The properties `hidden`, `show` or `local` are therefore not useful in the Promela2Java compiler. The

visibility of a declaration is remembered by the parser, but will simply be ignored by the transformation rules.

All the parts that are not included in the compiler will be parsed, but those parts will be categorized by the parser as "unwanted". For instance, the `never` clause is normally a module of a Promela model (same scope as `proctypes` and `typedef`), but because the `never` claim is "unwanted", the parser categorizes the module as an "`UnwantedModule`". This is the case for all the "unwanted" parts, whenever there are statements that are left out of the compiler, then those statements become "`UnwantedStatements`", an expression becomes an "`UnwantedExpression`", etcetera. Except for the visibility of statements, which are simply categorized as "`Visible"`".

A full list of parts that are not included in the compiler can be found in Table 5.1. The first part is the actual Promela code and the second part is the category the parser puts these pieces of code in. Whenever in the future the C code will be implemented then the parser can simply change the rule `CCodeStatement -> UnwantedStatement` into the rule `CCodeStatement -> Statement`. This means however that wherever the Statement is dealt within the transformation rules there should be some new rules created that would deal with C code.

Unneeded Promela	**Parsed Category**
`never`	`UnwantedModule`
`trace`	`UnwantedModule`
`notrace`	`UnwantedModule`
`c_code` module	`UnwantedModule`
`c_decl`	`UnwantedModule`
`c_state`	`UnwantedModule`
`c_track`	`UnwantedModule`
`xr`	`UnwantedStep`
`xs`	`UnwantedStep`
`assert`	`UnwantedStatement`
`c_code` statement	`UnwantedStatement`
`_last`	`UnwantedVarref`
`_nr_pr`	`UnwantedVarref`
`c_expr`	`UnwantedAnyExp`
`proctypename @ labelname` (remote reference)	`UnwantedAnyExp`
`np_`	`UnwantedAnyExp`
`enabled`	`UnwantedAnyExp`
`pc_value`	`UnwantedAnyExp`
`hidden, show` and `local`	`Visible`

Table 5.1: Parts not incorporated in the compiler.

For future incorporation of embedded C code one can have a look at paper [11] in which Erik D. Demaine describes ways to translate pointers in C to references in Java. This could

help with accessing the variables that can be declared in embedded C code in the generated Java code

Sheng Liang describes in [21] how, with some help of the Java Native Interface (JNI), functions which are implemented in another language, such as C can be called. However, Sheng Liang suggests that the use of JNI is a very difficult.

5.2 Installation Instructions

This section will describe how to use the compiler and how to get the compiler working. The files of the compiler are split into three parts: the parser files, the transformation files and the Java files. These files can be found in Appendix A.1, A.2 and A.3

The compiler uses a couple of other parts to work, so these need to be installed first and any system requirements on those parts are the system requirements for the entire compiler.

SPIN and Java

There are two parts that are essential to get the compiler working. Of course it is not necessary to have SPIN working, but SPIN is needed to validate your Promela model, because the compiler will not do that for you. The compiler uses a C preprocessor that is also used by SPIN as well, so by knowing that SPIN works correctly, means that the C preprocessor will work correctly as well. SPIN can be found on [17] and installation instruction can be found there or in [19] as well.

The end product of the compiler is Java code, so it is only natural to assume that Java is installed before using the compiler. The Java SDK that was used during this project is the Java SDK 1.5, but any Java SDK that is of a later version should work as well. The Java SDK and the installation instructions for them can be found on [24].

Stratego/XT

Parts of the Stratego/XT toolset are used in the compiler to parse a Promela file and other parts are used to create the transformations. That is the reason why some parts of Stratego need to be installed as well before the compiler can be created. The parts that are needed for the compiler are:

- aterm-2.5.1
- sdf2-bundle-2.4.1
- strategoxt-0.17
- Java-front 0.9

These parts and the instructions on how to install them, can be found on [31].

5.2.1 Compiler: Parser

When all other parts are installed correctly, then there should be a few things that need to be done to create the parser of the compiler. First, the grammar needs to be combined into one file by using the command:

Command A `pack-sdf -i Main.sdf -o PromelaPars.def`

`Main.sdf` is the main file in which links to all other files of the grammar. The above command results into the file `PromelaPars.def`. The parser is created by using the following command:

Command B `sdf2table -i PromelaPars.def -o PromelaPars.tbl`

The result is not actually a parser but a parse table(`PromelaPars.tbl`), that is used by Stratego to parse a file.

5.2.2 Compiler: Transformations

The following commands must be done first, in order create the transformations:

Command C `sdf2rtg -i PromelaPars.def | rtg2sig -o transformer/gener.str`

This will create a file `gener.str` inside the directory transformer that contains all signatures of all used ATerms in the parser. All the files that contain transformation rules will point to this file. The following command will create the actual transformations:

Command D `strc -i generation.str -I /usr/share/java-front/ -I /usr/share /java-front-syntax -la stratego-lib`

The `generation.str` is used as the main file and links to all other files that contain the transformation rules. The -I option should be used to import or include the two `java-front` directories. The locations of those directories might be different on any other computer. The generated file will be an executable called `generation`. The output on the screen will keep informing you when which part is completed successfully. The transformations are now complete and can be used to create Java code.

5.2.3 Using the Compiler

Compiling a Promela model into Java code can be done in three steps after all previous installation instructions have been followed. The actual parsing will be done by using the following commands:

Command E `cpp -P -C input.pml | sglri -p PromelaPars.tbl | pp-aterm -o input.trm`

The input file is the Promela model. The C preprocessor deals with all the "`#defines`" and then the output of that is parsed. The `pp-aterm` is there to create a nice layout for the results. These were actually three commands and can be separated by using the options `-o outputfile` and `-i inputfile` to create separate output and input files, if no `-o` option used, then the output will be printed on to the screen.

The transformation rules can be applied by executing the generation executable, by using the following command:

Command F `./generation -i input.trm | pp-aterm -o input.trm2`

The input file is the file that is used here is the output from the parser. The `pp-aterm` is used here again to get a nice layout for the generated file that contains the transformed abstract syntax tree.

The Java code can be generated by using the following command:

Command G `pp-java -i input.trm2 -o Main.java`

The input here is the transformed abstract syntax tree from the previous step. The tool pp-java, from the java-front package of Stratego, is used to generate the actual Java code. Compiling a Promela model can now be done in one big command pipe line, instead of first parsing and then transforming, by using the following commands:

Command H
```
cpp -P -C input.pml | sglri -p PromelaPars.tbl |
    ./generation | pp-java -o Main.java
```

The input file is now the Promela model and the result will be a file called Main.Java that will contain the generated Java code and can be used together with the other Java files in the corresponding directory. An overview of how all files relate to each other can be found in Figure 5.1

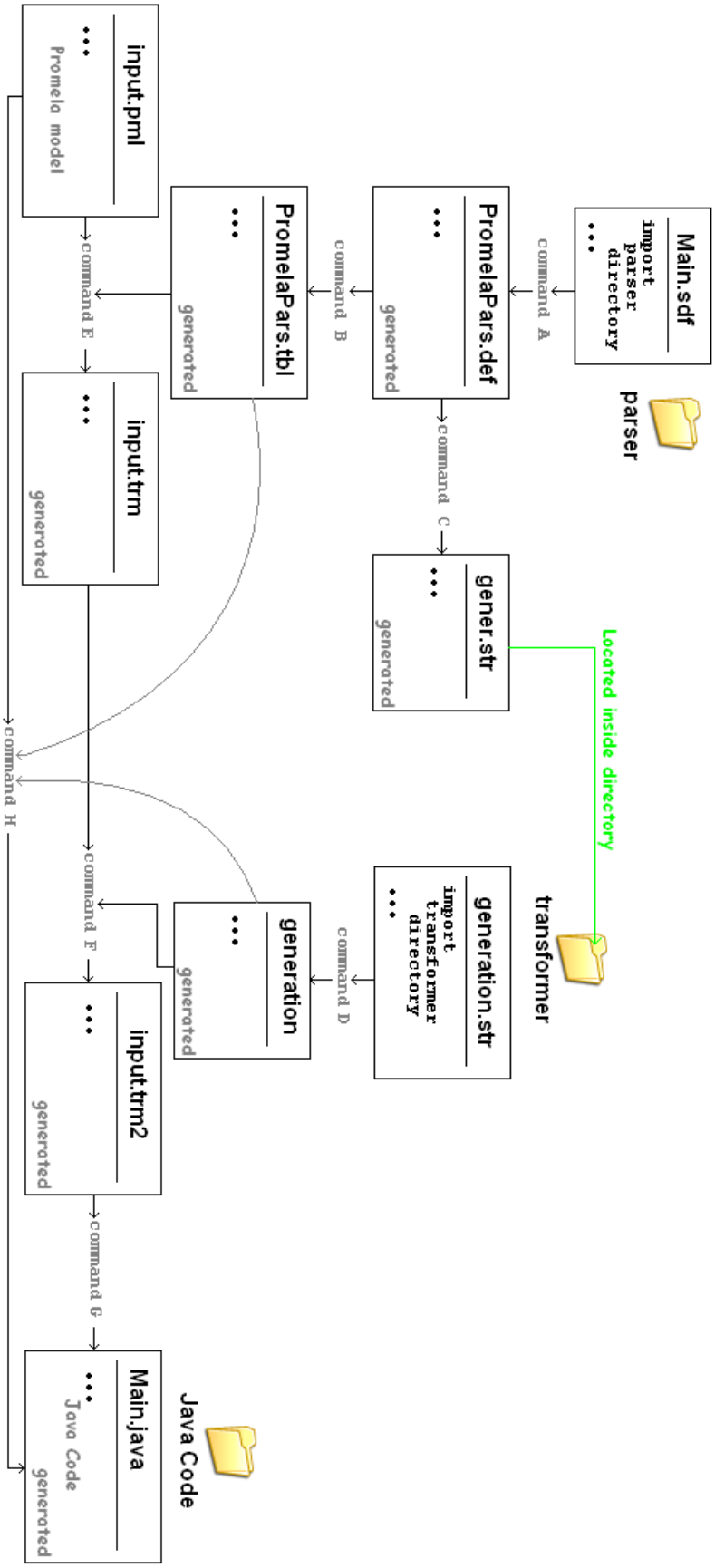

Figure 5.1: Overview of all files during installation and compilation.

Chapter 6

Example

This chapter will show how a protocol written in Promela is transformed into Java. The protocol chosen for this example is the 'leader-election using filters'-protocol. This protocol from [26] was derived from [2]. The filters that James H. Anderson et al. [2] use are shown in [10].

First, the protocol will be described, together with the Promela implementation of this model. Subsequently, the Java code generated with the Promela2Java compiler will be described followed by a comparison.

6.1 The Leader Protocol

The leader-protocol using filters selects a leader between a number of processes by using a filter. James H. Anderson et al. [2] describe that a filter as described in [10] ascertains that if m processes enter the filter, at most $\lceil m/2 \rceil$ processes and at least one process must leave the filter successfully.

The filter is described in [2] as a set of execution steps that every process will try to execute at the same moment in time. The execution steps will involve manipulating two global variables. Figure 6.1(a) shows how these processes could be connected. It does not matter how the processes are connected as long as all processes are connected to the shared variables. The two shared variables are an integer called *turn* and a boolean variable called *b*, which is initially false. All processes have a unique number called *p* as well.

Every process will copy the value of *p* to the variable *turn*. The process will then wait until the value of *b* is false and then set *b* to true allowing other processes to block at this point. The value of the variable *turn* is then checked to see if the value is equal to the unique number of the process *p*. If these values are equal the process has exited the filter successfully. If these values are not equal, the process will reset the value of *b* allowing the other processes to continue and it will block its own execution; the filter has not been exited successfully.

To elect one leader, only one process shall leave the filter successfully. This is solved in [2] by executing the filter a number of times until only one process remains. This has

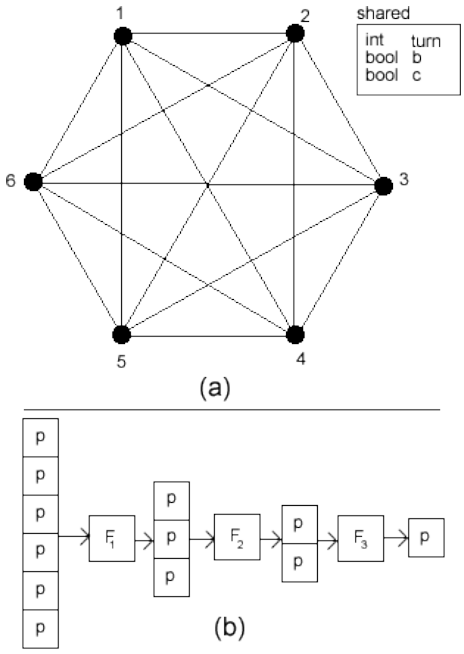

Figure 6.1: (a) Six processes in a network (b) Progress of the filters.

been effectuated by adding an extra global boolean variable called c and turning the global variables (including c) into arrays.

The execution steps will now operate upon the array values. All processes will start at the same moment manipulating the variables at the front of the arrays; the processes having exited the filter successfully will start their execution steps again, but now at the next place in the arrays.

Every filter will let at most $\lceil m/2 \rceil$ processes exit successfully; after $\lceil \log_2 m \rceil$ filters only one process has exited successfully. Checking whether that process is the only process having exited the filter successfully requires an additional boolean variable and one extra filter. Figure 6.1(b) shows how a filter phase eliminates approximately half of the processes before entering the the next filter phase until only one process remains.

6.2 The Protocol in Promela

The Promela model of this protocol has been found in [26] and the Promela model has been automatically derived from a model written in DVE, which is a low-level modeling language. The Promela model contains three global variables: an integer array *turn* and two boolean arrays b and c. There are six processes in the Promela model and they all will execute the same code.

The Promela model uses bytes instead of the integers and the booleans. The only addition to the model is a print statement informing the user which process has won the election.

The implementation of a process inside the Promela model of the leader election with filter protocol can be found in Figure 6.2.

```
active proctype P_4() {
          byte curr=0;
     p1: if
               :: turn[curr] = 4; goto p2;
          fi;
     p2: if
               :: b[curr]==0; goto p3;
          fi;
     p3: if
               :: b[curr] = 1; goto p4;
          fi;
     p4: if
               :: turn[curr]!=4; goto p5;
               :: turn[curr]==4; goto p8;
     fi;
     p5: if
               :: c[curr] = 1; goto p6;
          fi;
     p6: if
               :: b[curr] = 0; goto p7;
          fi;
     p8: if
               :: curr>0 && c[curr-1]==0; goto p9;
               :: d_step {curr==0 || c[curr-1]==1;
                         curr = curr+1;}  goto p1;
          fi;
     p9: if
               :: goto elected;
          fi;
     p7: false;
elected: printf("Elected p = 4");
}
```

Figure 6.2: Promela code for a process in the leader election with filter protocol.

6.3 The Protocol in Java

The Java code has been generated using the Promela2Java compiler. The code consists of a class `Main`, a class `GlobalVars` and one class for each of the six processes. The class `Main` adds an instance of each of the processes to the scheduler, which in turn will start the execution of all processes.

The Promela code of the leader protocol consists for the most part of *if statements*. An example of a transformation from an *if statement* in Promela to the corresponding Java code can be found in Figure 6.3. Every *if statement* in Promela has been translated into a `while loop` evaluating all guards, until at least one guard becomes true. This is followed by a Java `if statement` that selects the chosen guard and executes the corresponding sequence, which includes executing the first guard that has been evaluated before. It should be noted here that the distinction between evaluating a guard and executing a guard is needed because guards in Promela are not necessarily side-effect free.

```
p2: if
        :: b[curr]==0; goto p3;
    fi;
```

(a)

```
case LABEL_p2: {                                    //p2:
      int chosenPathi1 = -1;                              //if
      while(chosenPathi1 == -1){
          scheduler.want2Bspecial(_PID);
          path = new Vector();
          canExecute();
          synchronized(scheduler){
              boolean zero = false;
              zero = (int)global.
                 VAR_b[((int)VAR_curr)] == 0;
              path.add(zero);       //:: b[curr] == 0;
          }
          chosenPathi1 =
            scheduler.choosePath(path);
          if(chosenPathi1 == -1)
              blocked = true;
          else
              if(blocked)
                    blocked = false;
      }
      int indexi1 = -1;
      indexi1++;
      if(indexi1 == chosenPathi1){
          canExecute();
          boolean zero = false;
          while(!zero){
              canExecute();
              synchronized(scheduler){
                zero = (int)global.
                   VAR_b[((int)VAR_curr)] == 0;
                if(!zero)        //:: b[curr] == 0;
                  blocked = true;
                else
                  blocked = false;
              }
          }
          scheduler.stop2Bspecial(_PID);
          canExecute();
          choice = LABEL_p3;      //goto p3;
          labelset = true;
          break;                  //fi;
      }
}
```

(b)

Figure 6.3: A Promela if statement (a) and its corresponding transformed Java code (b)

Apart from the proper translation of the Promela code, one can notice the calls to the scheduler `canExecute()` and `synchronized(scheduler)` needed for proper interleaving. In addition, an increase in code size is noticeable; this is the subject of the next section.

6.4 Promela Model Versus Java Code

The first thing noticed from looking at the code is the size of the code. The size of a leader process in the Promela model is approximately 30 lines of code, whereas the Java code contains approximately 540 lines for each process. This has a few reasons, one of which is the layout of the code. The code generator uses a layout putting every opening and closing curly brace on a separate line and variable declarations are separated by white lines. Another rea-

son for the increase in the statement count is the following: As seen in the previous section, a Promela `if statement` with a single guard consisting of only three lines of Promela code is transformed into approximately thirty lines of Java code. As the transformed protocol contains eight `if statements` per process the blow-up is considerable.

The execution of Java code behaves identical to the simulation in SPIN of the Promela model: both print the number of the process having been elected to the screen. SPIN will stop the processes after a given amount of time and claim that a timeout has occurred, because all processes that not having been elected have blocked.

It turns out that successive executions of the Java code constantly elect the same process as the new leader, whilst various Promela simulations will not always elect the same process as the new leader. When the process being consistently elected in the Java execution is halted before the process executes the filter execution steps, sometimes the next process or the one after that will be elected. But, surprisingly, the result is never one of the last processes.

The reason this happened is because the first Thread has finished its execution before any other Thread has been able to start executing. This has been solved by letting all processes wait until a variable in the scheduler has become true and by letting that variable become true at the end of the main method in the Main class that has to add all proctypes to the scheduler. The solution has been added to the Promela2Java compiler and the generated Java code also elects a different process each time.

6.5 Conclusions

The Promela2Java compiler has successfully transformed an example Promela model into executable Java code. The generated Java code is executable but can also be used in a larger application of which leader election is only a part.

Chapter 7

Related Work

This chapter will portray other efforts to translate Promela into executable code.

7.1 Promela to Java

T.C. Ruys has already put some effort into comparing parts of the Promela language to parts of the Java language in [27]. The result thereof can be used to construct an automatic translation from Promela to Java. The slides describe the following mappings for the basic constructs of the Promela language:

- Process or Proctype → Thread
- Promela Datatypes → Corresponding Java Datatypes
- Promela Variables → Java Variables

The fifth version of the Java language (Java 1.5 [24]) contains some features facilitating the automatic transformation of the more complicated parts of the Promela language (in particular the atomicity, synchronization and channel constructs). These constructs are:

- Atomicity → Lock Interface or Condition Variables
- Global Variables → Atomic Variables
- Channels → BlockingQueue
- Rendez-vous Channels → Exchanger

The Lock Interface allows threads to synchronize their actions, for instance when dealing with `d_step` or `atomic` statements. The `java.util.concurrent` package in Java 1.5 contains the classes `BlockingQueue` and `Exchanger`. Because those classes will block a Thread object in the same way a channel blocks a process in Promela, they will be used to implement Promela channels .

The Lock Interface is one of the possibilities to deal with the synchronization of the processes or threads. Another possibility is the Metalocking algorithm described in [1].

This algorithm ensures mutual exclusive access for Threads in a highly optimized way. The article [4] proves that the algorithm indeed provides mutual exclusion and freedom from deadlock. Samik Basu and Scott A. Smolka [4] used the model checker XMC [28] and proved those properties by modeling the entire algorithm.

Information discussed in the papers [1, 4, 27, 28] was used to create some parts of the compiler. The proctypes of a Promela model will be represented by Threads and the `BlockingQueue` deals with the channels of a Promela model.

7.2 Promela to Byte-code

In [32, 33] Michael Weber describes a Virtual Machine for model checkers. This virtual machine was designed as a target language for several model checker languages, however, the author used Promela as a starting point. Constructs in Promela are mapped upon the instructions of the Virtual Machine. Doing so, the result is not a translation but again a model checker. Some parts of that system and some tests on this system have been reused in this project.

7.3 Promela to C

Siegfried Löffler has made an early attempt in [22] to generate an executable implementation from a Promela model. SPIN itself will transform a Promela model into a C-program called *pan.c*. This C-program is then used for simulation and verification purposes. Siegfried Löffler reuses part of this C-program and has created an extension of SPIN. A scheduler deals with non-determinism and atomic sequences.

Siegfried Löffler admits that there are some features of Promela missing in his implementation. The sorted send (`!!`) and random receive (`??`) operations of Promela have not been implemented. The program lacks an implementation of global synchronous channels as well. The `d_step` statement is not implemented in the program either. At the time these were new features of SPIN for which the compiler was not intended. Using a scheduler to deal with the non-determinism and concurrency was implemented by Siegfried Löffler similarly to the idea of Michael Weber in [32, 33].

Chapter 8

Evaluation and Conclusions

I will discuss all the tools that were used during this project. I will also evaluate the the Promela2Java compiler and what I do and do not like about it. This chapter will also describe the process I went through for this project: the parts that went well and less well, the parts that I would do differently looking back and parts that still can be extended to the Promela2Java compiler in the future.

8.1 Technical Evaluation

8.1.1 Tools

SPIN

SPIN [17, 19] is a model checker and is able to check a model for certain properties. A model is written in the Promela language and can be used by SPIN in a verification or a simulation. A model can be created of an existing system or of a design of a system. After a design is modeled and checked by SPIN, the system still needs to be created by hand. That is were this project comes into view.

SPIN itself is a program that will give any output back in textual form. There is the possibility to use Xspin [17, 19] to get a more graphical output when a model is being checked, but both tools look a bit outdated and could probably use an update. SPIN was very helpful in this project and in fact should be used before the Promela2Java compiler is used. SPIN will check if the Promela model, that is about to be transformed, is a correct Promela model.

The problems I encountered with SPIN are more related to the documentation of the Promela language [17, 19] than to the actual use of SPIN. There are still a few minor parts missing in the documentation that should have been described. I do understand that over the years there were a few additions to the language and probably just not documented or were not documented clearly enough. I do however believe that I have found most of the parts that were not described clearly in the documentation of SPIN and the Promela language.

I always had to use a simple text editor whenever I had to write a Promela model or when I was looking at a Promela model that was created by someone else. I would have

been pleased if I have had a text editor that could highlight the keywords of the Promela language or show erroneous constructions in a Promela model when the model is shown in the editor. An environment for Promela that behaves just like Eclipse, would have been nice.

Eclipse and JUnit

Eclipse [14] is an Integrated Development Environment, or better known as an IDE. Eclipse was used to help create Java code. The environment shows if whatever your typing at the time is erroneous or correct Java code, content assist can be used as well to automatically complete Java constructions. This all just makes writing Java code so much easier.

Eclipse was also used in combination with JUnit [29] to create test cases for the Java code that was created for this project. Executing a JUnit test case from Eclipse is possible and results are immediately received, because JUnit is integrated as a plug-in in Eclipse. The integration of JUnit in Eclipse makes creating and executing test cases so much easier, which is essential in any kind of development.

Stratego/XT

The Stratego/XT toolset [9, 31] was used to create the parser and the transformations, the java-front package was used and the pretty-printer in that package is used as a code generator. The parser is generated by writing the grammar of Promela in the SDF format. This was easy to use, because the SDF grammar turned out to be very similar to the Promela grammar as was described in the documentation [17, 19].

The transformation rules were written in a simple text editor and those rules were then used by a tool in the Stratego toolset to generate the transformations. The transformations can be executed by reading as input the output of the parser, the output of the transformations could then be used by the pretty-printer (code generator) to generate actual Java code.

The parser was easily created in comparison to the creation of the transformation rules because only the different parts that are allowed in the Promela language were necessary to describe. The transformation rules were a bit harder to create because all the possible combinations of all the different parts of the Promela language needed to be described. For each combination there should also be a corresponding piece of Java code and for each piece of Java code there should be a correct ATerm used or embedded Java code should be used.

The transformation rules were getting large fast, almost 14000 lines of code and although the rules can be separated in different files or modules, getting lost in the code is still easy. The creation of the executable transformations took more and more time when the number of transformation rules grew.

What would be nice is if an editor or environment existed that could be used to better highlight the constructions in the languages used in the SDF grammar and the transformation rules. The ATerms that are used in the transformation rules that should be highlighted are dependent on the language that is being transformed and all those ATerms have a sig-

nature description, maybe those signatures can be used to create some sort of dynamic environment that changes whenever a new signature is added.

8.1.2 Promela2Java Compiler

Use of Compiler

The compiler being split up into three different parts (the parser, the transformations and the code generator) is great in my opinion. This is nice from a developers point of view, because every part can be separately executed and tested. However, from a users point of view, executing all three parts could become a bit repetitive if a lot of Promela models need to be transformed. This can be easily overcome by using the Linux pipe-line or using a `makefile`.

The creation of the transformations executable takes a lot more time then expected. Transforming a Promela model into Java code also takes a lot longer then expected. This is because most of the transformation rules are all ordered in one kind of rule set. This kind of rule set was applied with the use of the `outermost` strategy, which is a built in strategy that walks over an abstract syntax tree and applies a rule from the rule set. This walk over the entire syntax tree is repeated until no more rules can be applied to any of the terms in the abstract syntax tree.

During a transformation, parts occur in the abstract syntax tree that will remain the same until the end of the transformations, but the strategy does not know that yet and still walks over those parts in the abstract syntax tree to see if some of the rules can still be applied. This is somewhat inefficient and probably the main reason for the compiler being slow and for the creation of the transformations executable being slow.

The problems users of the Promela2Java compiler can have are expectation problems. When a random Promela model is being compiled then a user might expect a proper application. This however will probably not always be the case.

The reason for this is the use of Promela models in verifications. It is for instance possible to model an existing problem in the Promela language. The SPIN verification process can then be used to find a solution to the problem, by finding one path through the model that conforms to the properties of the solution. This one path holds the solution, however SPIN looks at all paths to find that solution. If such a Promela model is compiled with the Promela2Java compiler, then the generated Java code will not contain any verification at all. This means that an execution of the Promela model will not necessarily execute the path that holds the solution to the problem, but a different path that does not hold the solution might be executed.

The reason for this is that the Promela language has the feature that proctypes can interleave each other and a selection (do or if) will choose randomly between all true guards. A verification process can look at all possibilities and finds the one path with the correct properties. However, because this feature is available there is the possibility that Promela code is created for a model of system that depends on this feature.

A solution for this problem could be that an execution option for the generated Java code can be created that would be guided by an execution trail. A trail can be created with

SPIN in the verification process of the Promela model and can be used in a SPIN simulation as a guided simulation to where a property did or did not adhere to a certain property. If this option is enabled then it would help people that want to follow a path to a certain point in the execution, but because the option can also be disabled would mean that other models that depend on the availability of multiple paths have nothing to fear.

Example

The Promela2Java compiler can take in any kind of Promela model and will generate an executable application, this is great. Another great thing is that the compiler could be used immediately as can be seen the example in Chapter 6. The Promela2Java compiler has successfully transformed the leader election protocol into executable Java code. The generated Java code is immediately executable. The generated code can even be used in a larger application in which a leader needs to be elected.

8.2 Project Evaluation

8.2.1 Well Done and less Well Done

I can split this project into four phases: the parser phase, the transformation rules phase, the the test phase and the writing phase. For each of theses phases I will describe what went well and what did not went as well as it should.

Parser Phase

During the creation of the parser I did not have much problems. One of the hardest parts was finding out what parts were allowed in the Promela language that were not easily derived from the grammar of Promela in [17, 19]. Another part which was a little more difficult, was dealing with ambiguity, because there are a few parts in the grammar, that when parsed would mean that the result became ambiguous.

Later on I found out there were some problems with the way that I had initially implemented the parser. Changing parts in the grammar to create the solutions for those problems was not very difficult. Most of the problems were solved by simply adding or changing small parts in the grammar.

Transformation Rules Phase

Creating the transformation rules was a bit more difficult than creating the parser was. This was mainly because I did not know much about how Stratego was used. I did not know much about the parser part either, but all the information I needed was for the parser was described in the manual of Stratego [9, 31]. There is also a lot of information on how to create the transformation rules in there as well, but I believe that in the case of the transformation rules, looking at more running examples would have been better, which was not so much necessary for the parsing part.

If I had looked more toward those examples then I would have find out that there were more standard strategies that I could have used and examples of how to use them. For instance, the use of dynamic rules, I did not even know that those rules existed until I was told about them.

I also underestimated the amount of work. Everything took so much longer then I thought, especially in comparison to the creation of the parser. And during testing I found some mistakes that I should have spotted on my own before testing.

Test Phase

I knew before hand that the testing phase should take a lot of time to get as much problems solved. However, I could not spent as much time testing as I wanted, because the transformation rules phase did run a bit longer then I thought. That is why I found the most difficult problems during tests of other people and later on found extensions of those problems in my own tests.

The hardest part was solving the problems that showed up in the parser. The easier it was to change parts of the parser the harder it was to change the corresponding parts in the transformation rules. One part changed in the compiler, meant that at least a dozen more parts had to be changed in the transformation rules. Other problems that were found were reasonably easy to find and solve.

Writing Phase

The writing phase was not actually a separate phase, but occurred before, during and after all the other phases and the result is what you are looking at right now. I am not really a big fan of writing, but I did quite well for this project if I may say so myself. I started early with the writing process, and because of that I did not have any problems when the deadlines arrived. I also have had an extra course "*Written English for Technologist*" during my research assignment, to help me along.

I did have some moments of writers block during this project, but by changing my focus to other parts of my project at those times helped me get through them.

8.2.2 Looking Back

When I look back at what I have been doing during this project I can find a few things that I would do differently if I had the chance to do it all over again. First of all I would have changed the way I started with the creation of the transformation rules. I would have looked into more examples before just simply starting to get things to work.

I would also have started much earlier with testing. Especially, if I would take some time between the parser phase and the transformation rules phase to test and see if the parser would work correctly. I also would save all the tests that I have created in order to see if the results were the ones I expected to see during the creations of the parser and the transformation rules.

For the rest of the project I would have done exactly the same, except maybe changing my attitude a little. Just so much that I would work just a bit harder in the beginning of this project and that I would expect that creating the transformation rules takes a lot of time.

8.3 Future Work

The obvious part that can be added in the future is the embedded C code. This is the only thing that is still missing from the compiler that could be useful. This is mainly missing because of two reasons. One reason is the fact that the use of embedded C code is highly discouraged, because those pieces of code cannot be verified by SPIN [17, 19]. The other reason is time, it would simply take to long to incorporate the entire C language in the Promela2Java compiler as well as incorporate the entire Promela language.

Creating an option that can guide the execution of the generated Java code with the use of a trail file generated by SPIN could be added in the future, but will probably be very difficult to implement. The main reason for this is because right now the interleaving of the different proctypes is dealt with by the Thread scheduler of the Java Virtual Machine. The interleaving would need to be scheduled in a different manner in order to guide an execution. The trail would also need to be interpreted by the generated Java code in order to know which execution path would need to be followed. This would probably mean that the trail file would also need to be compiled to get a better connection with the generated Java code.

The amount of time needed to compile a large Promela model into Java code, using the current version of the compiler, is very large. Any optimizations that can be done to speed things up are wanted and can be done at a future point in time. A large amount of time is also needed to create the transformations executable, so probably any optimizations on the time that is needed to compiler a Promela model will probably also shorten the amount of time needed to create the transformations executable.

8.4 Conclusions

My intention at the beginning of this project was to create a tool that can translate a Promela model into an executable application in the Java language. This has been done successfully by creating a parser and transformation rules with the Stratego/XT tool set. The code generator from the Java-front package of this tool set has been used as well. The result is the unique Promela2Java compiler that transforms a model written in Promela into an executable Java application.

The example from Chapter 6 shows that the compiler can easily generate executable Java code from a Promela model. This generated Java code can be used immediately or can be adapted to be used in a larger application.

Bibliography

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An Efficient Metalock for Implementing Ubiquitous Synchronization. *SIGPLAN Notices*, 34(10):207–222, 1999. Also published in *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*.

[2] James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.

[3] Ian Barland. *Promela and SPIN reference*. The Connexions Project, 2004. http://cnx.org/content/m12318/latest/.

[4] Samik Basu and Scott A. Smolka. Model checking the Java metalocking algorithm. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(3):12, 2007.

[5] Robert V. Binder. *Testing Object-Oriented Systems - Model, Patterns, and Tools*. Addison-Wesley, 2000. ISBN 0 201 80938 9.

[6] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language Specification*. OMG.

[7] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998. ISBN 0 201 57168 4.

[8] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2006.

[9] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. *Stratego/XT 0.16. Components for Transformation Systems*. ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06).

[10] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[11] Erik D. Demaine. C to java: converting pointers into references. *Concurrency: Practice and Experience*, 10(11-13):851–861, 1998.

[12] Alan Dennis, Barbara Haley Wixom, and David Tegarden. *Systems Analysis & Design: An Object-Oriented Approach with UML*. John Wiley & Sons, Inc., 2002. ISBN 0 471 41387 9.

[13] Apache Software Foundation. *The Apache Ant Project*. The ANT Homepage. http://ant.apache.org/.

[14] The Eclipse Foundation. *Eclipse*. The Eclipse Homepage. http://www.eclipse.org/.

[15] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*. John Wiley & Sons, Ltd., 2000. ISBN 0 471 97697 0.

[16] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24:43–75, 1989.

[17] Gerard J. Holzmann. *ON-THE-FLY, LTL Model Checking with SPIN*. The SPIN Homepage. http://www.spinroot.com/.

[18] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):17, 1997.

[19] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003. ISBN 0 321 228628.

[20] Anneke Klepper, Jos Warmer, and Wim Bast. *MDA Explained, The Model Driven Architecture: Practise and Promise*. Addison-Wesley, 2003. (ISBN 0 321 19442 X).

[21] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall PTR, 1999. ISBN 0 201 32577 2.

[22] Siegfried Löffler. From Specification to Implementation: A Promela to C Compiler. *Universität Stuttgart and Ecole Nationale Supérieure des Télécommunications*, page 135, 1996. http://citeseer.ist.psu.edu/288981.html.

[23] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model Driven Architecture*. Addison-Wesley, 2004. (ISBN 0 201 78891 8).

[24] Sun Microsystems. *The Source for Java Developers*. The Java Homepage. http://java.sun.com/.

[25] Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992. ISBN 0 521 40347 2.

[26] ParaDiSe. *BEEM: Benchmark for Explicit Model Checkers*. Parallel and Distributed Systems Labaratory. http://anna.fi.muni.cz/models/.

[27] C. Pronk. Promela to Java - Automatic translation. *Slides of TU Delft course IN4023: Advanced Software Engineering*, 2007. Adapted with permission from slides of Twente University by T. C. Ruys.

[28] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. *In Proceedings of the 9th International Conference on Computer-Aided Verification (CAV 97)*, 1997.

[29] David Saff, Kent Beck, and Erich Gamma. *JUnit*. The JUnit Homepage. http://junit.org.

[30] Edwin Vielvoije. *Promela to Java Using a MDA approach*. Research Assignment, 2008.

[31] Eelco Visser. *Stratego/XT*. The Stratego/XT Homepage. http://www.stratego-language.org/.

[32] Michael Weber. Parallel Algorithms for Verification of Large Systems. *RWTH Aachen - Department of Computer Science*, pages 97–106, 2006. http://aib.informatik.rwth-aachen.de/2006/2006-02.pdf.

[33] Michael Weber. An Embeddable Virtual Machine for State Space Generation. *Lecture Notes in Computer Science - Model Checking Software*, 4595:168–186, 2007.

Appendix A

Compiler Files

The files that are needed to create the compiler will be described here. The actual files can be found on: http://members.tele2.nl/edwin.v/

The link Promela2Java Compiler will get you to the place where the files can be found as well as instructions on how to use them. This can be seen at the top of Figure A.1.

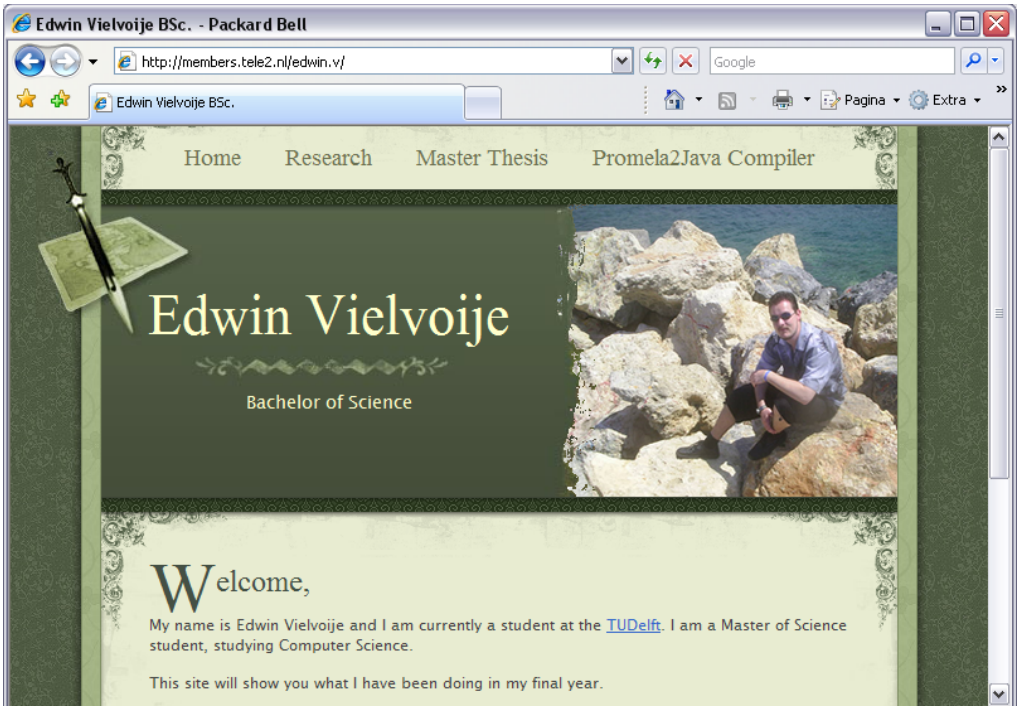

Figure A.1: http://members.tele2.nl/edwin.v/ Website of the Promela2Java Compiler

A.1 Parser Files

The parser consists of a number of files. These separate files are linked together by their `import` statements. All the files are inside a directory called `parser` except for the main file "`PromelaPars.sdf`", that is why the names of all the other files start with the prefix `parser/`. The files that are included are:

- `PromelaPars.sdf`
- `parser/Lexical.sdf`
- `parser/Modules.sdf`
- `parser/Sequences.sdf`
- `parser/Statements.sdf`
- `parser/Declarations.sdf`
- `parser/Variables.sdf`
- `parser/Expressions.sdf`

A.2 Transformation Rule Files

The parser consists of a number of files. These separate files are linked together by their `import` statements. All the files are inside a directory called `transformer` except for the main file "`generation.str`", that is why the names of all the other files start with the prefix `transformer/`.

There are also a couple of files that are simply there for the use of type checking, these files reside within the `Typechecking` directory within the `transformer` directory. These files that are used for type checking have the prefix `transformer/Typechecking/`.

A number of files also Java code within some rules, a number of files are added that simply contain the text `Meta([Syntax("Stratego-Java-15")])` and will have the same filename as the file of the rules that uses Java code except the extension of the file will be "`.meta`" instead of "`.str`". This is done in order to let Stratego know that these files use Java code. The files that are included are:

- `generation.str`
- `transformer/Program.str`
- `transformer/MainPart.str`
- `transformer/Classes.str`

- `transformer/Globalclass.str`
- `transformer/Executing.str`
- `transformer/Declarations.str`
- `transformer/Expressions.str`
- `transformer/DoIf.str`
- `transformer/SeparatingLabels.str`
- `transformer/Labels.str`
- `transformer/Varref.str`
- `transformer/Print.str`
- `transformer/Receiving.str`
- `transformer/Sending.str`
- `transformer/Atomic.str`
- `transformer/Unless.str`
- `transformer/UnlessAtomic.str`
- `transformer/PropegateStop.str`
- `transformer/gener.str`
- `transformer/NewSignatures.str`
- `transformer/Inline.str`
- `transformer/Typechecking.str`
- `transformer/Typechecking/TypedefCheck.str`
- `transformer/Typechecking/Separating.str`
- `transformer/Typechecking/Globalvars.str`
- `transformer/Typechecking/Localvars.str`

A.3 Java Files

There are some extra files needed to create a Java application from a Promela Model. These files will not need to be generated every time a new Promela model is to be transformed. These files will be the same for every generated Java application from a given Promela model. The Java classes that will be generated have some dependencies on the following files and some of those files has some dependencies on the classes that will be generated. The files reside in the directory `JavaCode` and the files that are included are:

- `JavaCode/Scheduler.java`
- `JavaCode/Prooctype.java`
- `JavaCode/Channel.java`
- `JavaCode/Message.java`