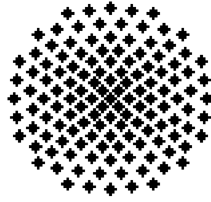

University of Stuttgart

Faculty of Computer Science

Course of Study: Computer Science

Examiner: Prof. Dr. Kurt Rothermel

Supervisor: Dipl. Inf. Stella Papakosta

Commenced: September 1st, 2001

Completed: March 28th, 2002

CR-Classification: C.2.2, I.6.3

Studienarbeit Nr. 1826

Generating Implementations from Formal Specifications: A Translator from Promela to Java

Matthias Papesch

Institute of Parallel and
Distributed High Performance Systems
University of Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Abstract

HiSPIN stands for **H**ighly **i**nteractive SPIN. SPIN itself is an abbreviation for **S**imple **P**ROMELA **IN**terpreter. The HiSPIN project aims at supporting the handling of protocols and algorithms. It tries to combine the advantages of the existing tools PROMELA, SPIN and HiSAP.

The main parts of this work comprise a JAVA implementation of the SPIN simulator and the development of a communication model which enables the simulation process to control visual elements of the HiSAP visualization toolkit.

Contents

Contents **1**

List of Figures **3**

1 Introduction **4**
1.1 The HiSPIN Project 4
1.2 Motivation 4
1.3 Goals 5
1.4 Task 5
1.5 Overview 5

2 Background **6**
2.1 About PROMELA 6
2.2 The SPIN simulator 7
2.3 The HiSAP Visualization Toolkit 7
2.3.1 The HiSAP-*node* 8
2.3.2 The HiSAP-*connection* 8
2.3.3 The HiSAP-*message* 8
2.4 Protocol Considerations 8
2.4.1 Sliding Window Protocol 8

3 Requirements **12**
3.1 General Issues 12
3.2 The Simulator 12
3.3 HiSAP Connectivity 13
3.4 A Demonstrative Example 13

4 Design **14**
4.1 General Issues 14
4.2 The Graphical User Interface 14
4.3 The Simulator 15
4.3.1 Overview 15
4.3.2 The Scanner 16
4.3.3 The Parser 16
4.3.4 Static data 18
4.3.5 Expressions 18
4.3.6 Variables 19

4.3.7 Statements and Sequences 20
4.3.8 Control Flow Statements 20
4.3.9 Proctypes 20
4.3.10 The Specification 21
4.3.11 The Simulation 21
4.3.12 The Scheduler 22
4.3.13 Dynamic Data 22
4.3.14 Message Passing 24
4.4 Event Generation 24
4.4.1 Events in the Sliding Window Protocol 25
4.5 The Communication Model 25
4.5.1 Plain Event Model 26
4.5.2 Extending PROMELA 27
4.5.3 Managed Event Model 32
4.5.4 Event Model Summary 33

5 Implementation **35**
5.1 The Simulator 35
5.1.1 The Scanner 35
5.1.2 The Parser 37
5.1.3 Static Data 39
5.1.4 Expressions 39
5.1.5 Variables 43
5.1.6 Terminals and Scanner Adaption 43
5.1.7 User defined Compund Types 47
5.1.8 Statements and Sequences 49
5.1.9 Control Flow 52
5.1.10 Proctypes and other Units 54
5.1.11 Specification 56
5.1.12 Dynamic Data 57
5.1.13 Scheduler Functionality 58

6 Conclusion & Outlook **60**
6.1 Conclusion 60
6.2 Outlook 60

Bibliography **61**

List of Figures

2.1 Sending data packets with the sliding window protocol. 10
2.2 A Promela implementation of the sliding window protocol. 11

4.1 How plain text is rendered into simulation data. 17
4.2 The difference between the static specification and the dynamic simulation. 23
4.3 Events for the sliding window protocol, processes and channels. 26
4.4 Generating events with language elements. The model component generates the event when it encounters a special statement. 28
4.5 Realizing events with Promela extensions using a designated event-channel 30
4.6 Using event comments for the communication model. 31
4.7 The task of the Event Manager. 33
4.8 A sample mapping for a visual component for the Sliding Window Protocol. 34
4.9 Summary of the considered Event Model strategies. 34

5.1 Parts of the scanner description file to demonstrate the content different sections 36

Chapter 1

Introduction

To become accommodated with the subject of this work, this section presents a very coarse grained overview on how the different matters overlap with each other. Each part will be covered in more detail in later sections.

1.1 The HiSPIN Project

HiSPIN stands for **H**ighly **i**nteractive SPIN. SPIN itself is an abbreviation for **S**imple **P**ROMELA **IN**terpreter. More background information about SPIN will be presented in section 2.2.

The project, as described in [11], aims at supporting the handling of protocols and algorithms. Most initial model descriptions of protocols and algorithms are abstract and static, e.g. (pseudo-) code fragments or automata. During the design phase of a new protocol or when teaching how an existing protocol works the ability to interact with existing models is of high value.

This is where simulation and visualization come in. With tools such as HiSPIN a protocol comes to life. Anybody may play around with certain parameters and actually *see* the effects.

So far, this is a two-step process. First, the protocol has to be designed. The result of the design process – some kind of specification – is then used as the base for developing refined explanatory models used for teaching and improvement of the same protocol. The goal of HiSPIN is the automation of the second step. A visual and interactive model is to be generated from the specification almost automatically.

1.2 Motivation

The **H**ighly **i**nteractive **S**imulation of **A**lgorithms and **P**rotocols (HiSAP) toolkit, as presented in [12] and covered in more detail in section 2.3, provides the means to create JAVA applets that visualize and animate the message flow in communication protocols. While the resulting applets perform remarkably well there are certain drawbacks to use this approach in a large scale. In part, they attribute to the application of the two-step process mentioned above.

- every single protocol must be programmed in JAVA.

- user interaction is hard to accomplish
- considerable overhead for implementing variations of the same protocol

SPIN somehow resembles a counterpart to the fancy HiSAP toolkit. As a platform dependent program written in C, SPIN typically operates with textual representation. It simulates protocol specifications given in the **pro**cess **me**ta **la**nguage (PROMELA). It does provide user interaction and session replay. However, the lack of a user-friendly interface and the textual representation of the simulation activity severely restrict its applicability.

PROMELA, the language in which specifications accepted by SPIN have to be written, as described in [3], will be covered in more detail in section 2.1. It has proven an adequate choice for the design and validation of communication protocols and the like.

1.3 Goals

Working with each of these three entities – PROMELA, SPIN and HiSAP – in the domain in which it excels sparked the desire to have a single application that combines

- the fanciness and comfort of HiSAP,
- the easy way to interactively simulate of SPIN,
- the flexibility of PROMELA to alter protocol specifications.

Since visualization is the key goal sacrifices in performance may be tolerated to a certain degree. Obviously, no JAVA implementation will ever even get close to the performance of SPIN. As long as the results become available within a reasonable time span to be used interactively this is acceptable.

1.4 Task

The task at hand consists of four distinct parts

1. A JAVA implementation of the SPIN simulator
2. The development of a communication model which enables the simulation process to control visual elements of the HiSAP visualization toolkit
3. A sample protocol specification.

1.5 Overview

The remainder of this work is organized as follows. Chapter 2 introduces various aspects that the reader should be familiar with. The requirements for this work are presented in Chapter 3. Chapters 4 and 5 deal with the major concepts and their realization. Chapter 6 provides a summary and gives an outlook for the future.

Chapter 2

Background

The purpose of this Chapter is to introduce the reader to the basic concepts to which later Sections refer. Among these are the tools used and toolkits which are to be considered. Another part consists of the examples that will be used to demonstrate certain facts later on.

This Chapter provides vital information for the following Chapters. However, the experienced reader may choose to skip any of these, if they feel acquainted with the topic.

2.1 About Promela

Promela is used to describe models of protocols or distributed systems. These models focus on process interaction while they neglect details not related to this primary goal.

In [3] a very detailed description of the initial version of Promela is given. There also is an extensive online Promela Language Reference [7] which keeps track of the current version. Although these two sources are a highly recommended reading, there will be a very compact overview on Promela next.

Syntactically, Promela looks similar to C. The two languages also share a subset of data types. The concept of a function subroutine in C somehow resembles a process in Promela.

Processes

A Promela model consists of one or more process definitions. These comprise a sequence of statements and parameter declarations. Multiple processes, even multiple instances of a single process definition, may run concurrently. There are two possibilities to start a process: either at the begin of a simulation run or by applying the run operator.

Variables and Channels

Variables in Promela can be one of the following types: bit(1), bool(1), byte(8), short(16) or int(32). The numbers in the parentheses denote the number of bits used to store the value. Floating point numbers have been deliberately omitted from Promela. Variables may either be declared global or local to a process.

Although channels may also be declared global or local, any channel may be accessed from within any process. This makes sense, since the main purpose of channels is to provide the means for processes to communicate with each other.

An attribute to a channel definition is its queue size. This property specifies how many messages fit into the buffer of the channel. If a channel is not full a process may append a messages to the queue and continue its execution. This makes up asynchronous communication. When a queue size of 0 is given, the channel is treated as a synchronous one. This means that a process can only execute a send operation when there is another one that executes a receive operation on the same channel at precisely the same moment.

Statements, Expressions and Conditions

A process definition comprises a sequence of statements. A statement can either be executable or blocking. If a statement is blocking, this process has to wait until the statement becomes executable before it can continue. Basically, any statement that evaluates to 0 or false blocks. Channel operations can also block, like sending to a channel with a full queue or receiving from a channel with an empty queue.

Control Flow Statements

A key feature of PROMELA is non-determinism. A process can be given multiple choices to continue. The first statement of such an option sequence is called the *guard*. It is this guard, which decides whether an option is executable or not. If all guards of a compound statement block, the entire statement blocks and the process has to wait until at least one guard becomes executable.

Repetitions add a loop functionality. After the last statement of an option has been executed, the control flow returns to the repetition statement instead of the next statement in the sequence. The latter can only be reached with a *break* statement. These constructs may be arbitrarily nested.

PROMELA also provides a goto statement to perform unconditional jumps to labels within the same process. These jumps never block.

2.2 The SPIN simulator

As the name already implies the **S**imple **P**romela **IN**terpreter [3, 4] brings the model of a protocol specified in PROMELA to life. Since the model is inherently nondeterministic, each time the SPIN is invoked it will execute one possible sequence. Several ways to influence this sequence exist. The simulation halts when either all processes finish (or reach an end-state) or a deadlock occurs.

Another option SPIN offers is to generate a protocol analyzer. This is basically the C source code of a state machine which performs a state search on the model of a given protocol. That is, it tries a large number of execution possibilities and verifies that the given model does not reach an invalid state.

2.3 The HiSAP Visualization Toolkit

The HiSAP toolkit divides the visualization process into two distinct layers.

Model Layer

The model layer takes care of the internal protocol state and. message flow

View Layer
The view layer handles the visualization and user interaction.

The model layer itself uses three basic entities *node*, *connection* and *message*. These are kept very simple, but can be combined into units of arbitrary complexity and descriptive power. The HiSAP toolkit has been re-modeled. The version referenced here can be found in [5, 6].

2.3.1 The HiSAP-*node*

An arbitrary protocol entity is represented by a *node*. It keeps any number internal states, incoming and outgoing connections. JAVA code controls the way how the instance reacts to any *message* or state changes.

2.3.2 The HiSAP-*connection*

A *connection* has exactly two end-points, which are *node* objects. The number of directions in which communication can take place makes up an important characteristic of a connection. By default HiSAP provides simplex and half-duplex *connection*. These two types combined with any number of nodes result in a custom *connection* with any functionality one can imagine. For instance to acquire a full-duplex link between two nodes, simply combine two simplex *connection*s for each direction into a single one. For many-to-many communication add a single node in between that acts like a router.

2.3.3 The HiSAP-*message*

The user data that the protocol delivers is represented in a *message*, which is fed into a *connection* by a *node* and received from a connection by another – not necessarily different – *node*.

2.4 Protocol Considerations

The entire project aims to visualize how protocols work. For this reason, it is important to have at least one protocol in mind for which a realization is planned. Most protocols fall into one out of three classes. First of all, there are protocols for point-to-point connections. These consider exactly two parties that communicate with each other. The sliding window protocol falls into this category. For the second class, protocols that operate on multiple stations that are arranged in a ring, the token ring protocol is described. Finally, a protocol that works on a broadcast network is presented, the two-phase-commit protocol.

2.4.1 Sliding Window Protocol

The sliding window protocol is applied in point-to-point communication. It expects from the lower layer to only deliver correct packets and makes sure that these are passed on to the higher layer in the correct order. Therefore, it is typically applied on the data link layer. The basic concept provides several levels of complexity. In [1, pp. 202], a detailed tour from a very simple design to a more complex and efficient one is provided.

The sliding window protocol is responsible for assuring to the above layer that the order of the packets is maintained. For this purpose, sequence numbers for the packets are necessary. These typically range from $0 \ldots 2^{n-1}$, to optimally use a n-bit field.

The protocol derives its name from the *windows* it uses as buffers to store packets which are currently being processed in. The size of these windows does not really matter. Some implementations use the same fixed sizes for both, the sending and the receiving window, some use different sizes and the sizes may even shrink or grow with time.

Before the sender transmits a packet it calculates a sequence number (seq_no) for it and checks if it can be stored in the *sending window*. The sending window contains all packets which were sent, but which have not yet been acknowledged. Be the current sending window size sws. To find out whether all slots are full, one more number is needed: the sequence number of the last packet which was acknowledged by the receiver, $last_ack$. If $seq_no < last_ack + sws$ holds, the packet may be stored in slot no. $last_ack - seq_no \% sws$. Otherwise, the sender is forced to wait for an acknowledgement to increase $last_ack$ or a retransmit timeout to occur.

The bigger the sending window, the more packets may be in transit at any single point in time. Be the sending window size $sws = 1$. This means, whenever the sender transmits a packet and stores it in the sending window, the latter will be full. This fact forces the sender to block until the acknowledgement for this very packet comes in.

When a connection inhabits a high propagation delay, having a sending window size of $sws = 1$ forces the sender block way too long and does not use the available bandwidth efficiently. Increasing $sws = n$ will allow the sender to keep sending until at most n acknowledgements stand out before it is forced to block. The name of this technique is *pipelining*.

On the other end of the connection, the receiver keeps a *receiving window* of size rws which provides buffer space for all packets it is willing to accept. It also needs to remember the sequence number seq_no of the packet to which all packets have been received.

The use of *cumulative* acknowledgements helps to save network traffic. Instead of acknowledging every single packet $n \ldots m$, all these packets are implicitly acknowledged with the acknowledgement for packet m.

When the packet with the sequence number new_seq_no arrives, the receiver must determine whether it may be accepted. This is the case if its sequence number fits in the expected range, $new_seq_no < seq_no + rws$. If so, it will be put into buffer slot $new_seq_no - seq_no \% rws$. Whenever the above layer receives packets $seq_no \ldots seq_no + m$ a cumulative acknowledgement is sent and $seq_no = seq_no + m$ frees m buffer slots. See Figure 2.1 for an example transmission.

All incoming packets which do not fit into the receiving window will be dropped. If the next expected packet does not arrive in a reasonable time span the last acknowledgement will be retransmitted.

At the receiver the buffers become very useful, whenever packets are lost or the order in which they are received is mangled. The out of order packets will be buffered in the receiving window, until the missing packet is received. Thus, this single packet enables the whole bunch of buffered packets to be acknowledged. This *selective repeat* prevent the sender from having to retransmit packets that have already been successfully transmitted due to an error of a different packet.

Figure 2.2 shows a sliding window protocol with the fixed sizes for the sending (SWS) and the receiving (RWS) window that uses pipelining and selective repeat.

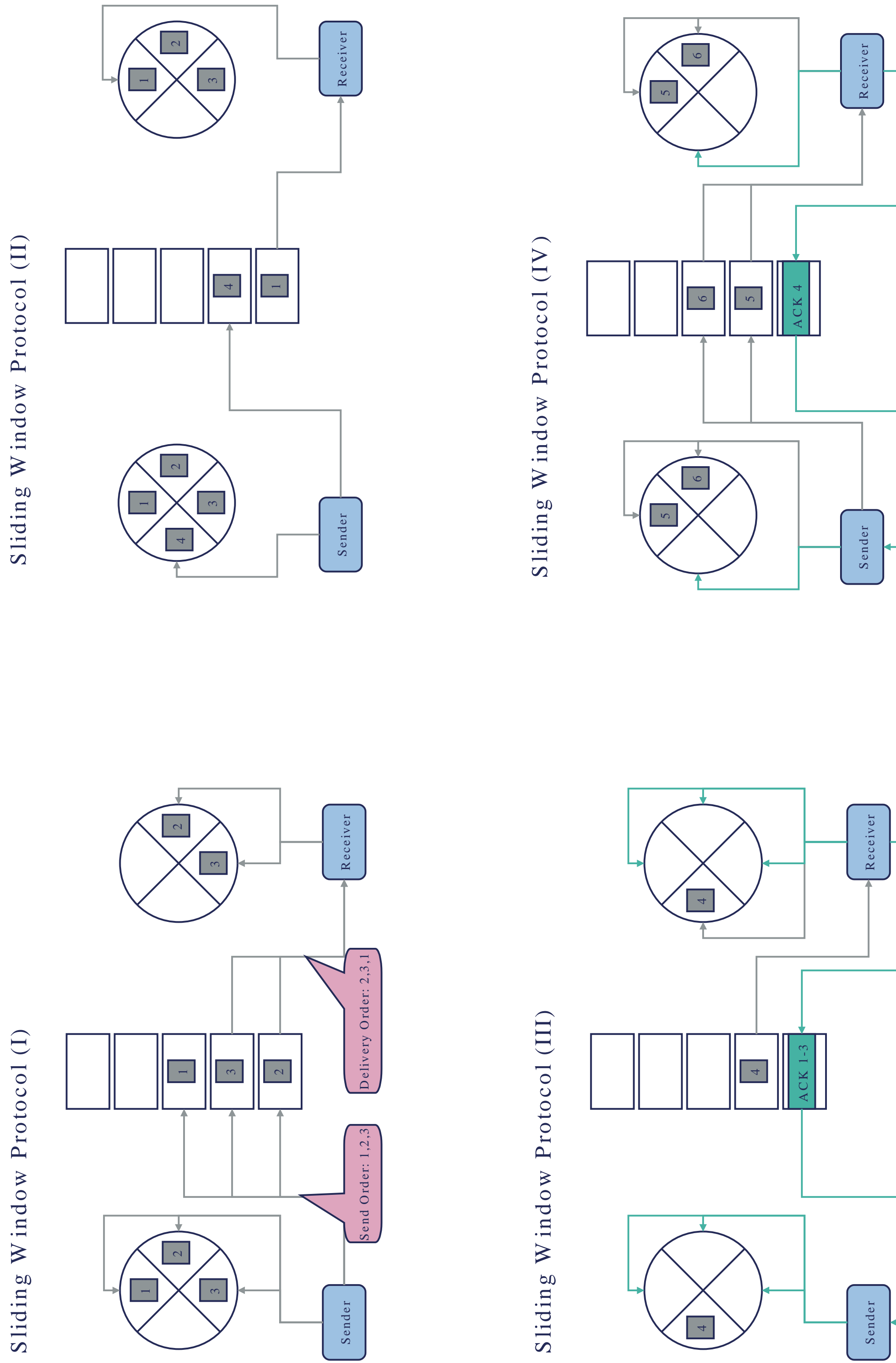

Figure 2.1: Sending data packets with the sliding window protocol.

```
1  mtype = {data, ack, fin, fin_ack};
2  chan  connection = [DLEAY] of {mtype, int}
3
4  proctype sliding_window_sender(){
5      boolean buffer[SWS];
6      int ws       = SWS;
7      int seq_no   = 0, last_ack = −1, new_ack;
8      do
9       :: (  seq_no < packet_count ) −> /∗ Comment ∗/
10         if
11          :: (  seq_no − last_ack < ws ) −> // noch einer
12              connection!data(seq_no);
13              buffer [  ws % seq_no ] = true;
14              seq_no++;
15         ::  connection?ack( new_ack ) −>
16              do
17               :: (  new_ack > last_ack ) −>
18                   last_ack++;
19                   buffer [  last_ack  % ws ] = false;
20               ::  else −> break;
21              od;
22         ::  timeout −> connection!data( last_ack + 1 );
23         fi
24      ::  else −> connection!fin; break;
25     od;
26 }
27 proctype sliding_window_receiver(){
28     boolean buffer[RWS];
29     int ws       = RWS;
30     int seq_no   = −1, new_seq_no;
31     do
32      ::  connection?data( new_seq_no ) −>
33         if
34          :: (  new_seq_no > (seq_no + 1) && new_seq_no < seq_no + ws ) −>
35              buffer [  new_seq_no − seq_no ] = true;
36          ::  else −> skip;
37         fi ;
38      :: (   buffer [(seq_no + 1) % ws ] ) −>
39         do
40          :: (   buffer [(seq_no + 1) % ws ] ) −>
41              seq_no++;
42              buffer [  seq_no % ws ] = false;
43          ::  else −> break;
44         od;
45         connection!ack(  seq_no  );
46      ::  channel?fin −> break;
47     od;
48 }
```

Figure 2.2: A PROMELA implementation of the sliding window protocol.

Chapter 3

Requirements

This Chapter breaks the topic into different parts and describes what has to be done to achieve the desired goal.

3.1 General Issues

It is desirable that the entire program is independent of any specific platform. Of course, it must be able to cooperate with HiSAP. The execution speed must be tolerable for interactive sessions.

3.2 The Simulator

The simulator generates the information needed to visualize the activity of a simulation in progress. It uses the parse tree and name spaces derived from the given protocol specification by the parser and performs the simulation. It should behave in the same way SPIN does.

There are multiple processes running at the same time and each process may have several non-deterministic choices for the next statement to execute. Some means to determine the execution sequence is needed. There are at least three ways to decide how to do so. The simulator should implement one and provide the possibility to add the others.

- randomly choose the next statement
- allow the user to choose among all possible statements
- the simulate according to a previously generated trail-file

The choice which model to use may be changed at any time during the simulation. The degree of user interaction should be as high as possible. The user should be able to suspend the simulation at any time and take back any number of steps or alter the system state.

3.3 HiSAP Connectivity

The driving desire for this implementation is to have a simple way to visualize the message flow of any given protocol. HiSAP provides the visualization toolkit and the SPIN implementation delivers the data to perform the animation.

The tricky part is, that these two pieces do not share the same set of basic elements. Take a look at message passing, for example. HiSAP provides nodes and connection between nodes, which allows two nodes to explicitly use point-to-point communication. PROMELA, on the other hand, makes each and every message queue a global object and thus accessible for each and every process (i. e. node).

Clearly, these incompatibilities create the need for either adapting one model to exactly match the other or to create an intermediate communication model with interfaces for both instances. Several ways to accomplish the visualization of a simulation in progress need to be developed and compared to find the best solution possible.

3.4 A Demonstrative Example

The example should try to demonstrate the given possibilities. A protocol specification should be created and simulated.

Chapter 4

Design

This Chapter addresses issues that have been considered during the development period. It tries to make the reader understand why certain decisions were taken. For a more in-depth coverage of specific implementation details see Chapter 5.

4.1 General Issues

The very first issue originates from two key requirements: the desire to have a platform independent program and the need to incorporate HiSAP determine the choice which programming language to use. Under these circumstances there is no alternative to JAVA.

HiSAP already uses the SWING toolkit which is both more recent and superior to the **A**bstract **W**indow **T**oolkit introduced with the very first JAVA release. Clearly, using SWING components whenever possible is sound choice for this work also.

4.2 The Graphical User Interface

The preliminary **G**raphical **U**ser **I**nterface (GUI) for the time being needs to provide the following features

- Load/Edit/Save PROMELA specifications.
- Syntactically support the editing.
- Present the result of the parsed specification.
- Show the state of the ongoing simulation.
- Trace the events generated by the simulation.

To not clutter other package hierarchies, all GUI related files should have their own namespace.

package HiSPIN.gui

4.3 The Simulator

The simulator operates on the HiSAP model layer (see Section 2.3). It comprises all the elements involved from a plain text protocol specification given in PROMELA to the simulation in progress. Figure 4.1 shows the different stages a specification passes through on its way to the view layer.

4.3.1 Overview

The design of the different parts will be presented next. More detail on the actual implementation is given in section 5.

1. The *scanner* reads the protocol-specification from a file. It groups characters from the continuous stream into tokens. The design is covered in section 4.3.2, for implementation details see section 5.1.1.

2. The *parser* asks the scanner to deliver the tokens one by one. It then matches them against the rules of the PROMELA grammar and derives a parse tree, also referred to as specification.

3. *Static data* does not change during a simulation run. All static data together reflects the content of the protocol-specification.

   Expressions
   The most simple elements of a protocol specification are expressions. Their main task is to be evaluated and to return a result.

   Symbol A symbol is an identifier which does not belong to the PROMELA keywords. It is not bound to a any type.

   Variables
   Variables serve two different purposes. They are used to store information and to allow inter process communication. They also appear in two different shapes. A *declaration* assigns a certain type to a symbol and makes it accessible. The actual access is handled by a *reference*.

   Statements and Sequences
   Sequences of statements manipulate the system state of a running simulation, e.g. change values of variables or defer execution.

   Control Flow Statements
   This is a sub-category of statements which comprises special statements which influence the execution sequence of statements.

   Proctypes
   Each proctype references several variable declarations and a sequence of statements. Each proctype can be instantiated at the beginning of a simulation run or with the special run operator.

   Specification
   The specification comprises several variable and proctype declarations.

4. The *scheduler* handles the execution of statements. It needs to be aware of all existing processes and their executable options. The key task of the scheduler is to choose which process should run next.

5. During a simulation run, *dynamic data* is instantiated from *declarations*, a subset of static data. Other than static data, dynamic data changes during the course of the simulation.

   Simple and Channel Instances
   When a simple or channel declaration is instantiated a simple or channel instance is created within the given context.

   Compound Instances and Contexts
   An instance which can keep map symbols to instances.

   Processes
   A simulation run consists of one or more processes. A process is instantiated from a proctype.

   Simulation
   The most important and complex piece of dynamic data is a simulation. It is instantiated from a specification and provides everything to control the simulation run.

6. *Message Passing* allows processes to communicate with each other. The declaration of channel type variables specifies what kinds of messages this channel accepts and how many of them it can store. Statements allow processes to operate on channels in various ways.

4.3.2 The Scanner

The scanner, or lexical analyzer, reads a protocol-specification from a plain text file and groups characters from the continuous stream into tokens. Actually, it does not act on its own but is controlled by the parser.

The most important design issue for the scanner is the decision to either implement it by hand or to use a tool to have it generated. There is virtually no advantage to a custom implementation as it is more complicated to create and the product is less transparent. The input for a generator boils down to the minimum of information needed to describe the features of the language to be recognized.

What remains is the decision to a choice which tool to use to generate the lexical analyzer. In [3], the standard UNIX tool lex is used. This choice is not adequate for this project as lex produces a scanner which is implemented in the C programming language. However, there are very similar (not only in the name, also in terms of usage) tools which generate scanners with JAVA as the target language. Two of them – JLex and JFLex – have been considered for use. The decision to prefer JFLex over JLex results from the slightly more mature interaction with the chosen parser generator and the faster execution times of the generated scanner as shown on [8]. For more information on scanners in general and lex in particular see [2, ch. 3].

4.3.3 The Parser

The parser requests the tokens one by one from the scanner. Again, in [3] another common UNIX tool YACC (**Y**et **A**nother **C**ompiler **C**ompiler) is used to generate the parser. Its target language is also C, and for the very same reason it is not suitable for this project. Instead, CUP(**C**onstruction of **U**seful **P**arsers) has been chosen, as it is written in JAVA

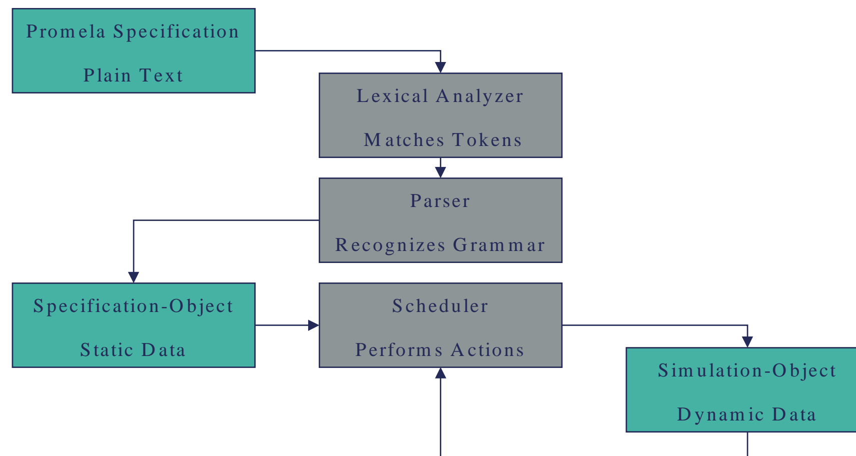

Figure 4.1: How plain text is rendered into simulation data.

and produces parsers in JAVA. As stated in [10], it has been designed as a successor for YACC for the JAVA language. For more information on syntax analysis see [2, ch. 4].

So, the parser requests token after token from the scanner. It matches those tokens against the production rules of the PROMELA grammar. It will complain about syntactical errors. Assuming for now that a correct PROMELA specification is given, the parser builds the parse tree which is also referred to as *specification*.

Helpers

Since the final specification object will be some kind of tree a reasonable approach is to have some base classes which provide the basic operations on a tree-like structure. Such structures are tree nodes with multiple and leaf nodes without any children.

So, if the grammar expects a list of parameters for example, this list can easily be constructed if the parameters derive from these base classes. These classes will be kept under

package HiSPIN.promela.parsing

interface HiSPIN.promela.ParseTreeElement

class HiSPIN.promela.parsing.LeafNode

A leaf node in the parse tree. It has no more children.

class HiSPIN.promela.parsing.TreeNode
A node in the parse tree which has a left and a right son. Used to build lists.

4.3.4 Static data

Static data comprises how the data derived from the tokens delivered by the scanner and matched by the parser is organized. Static data provides the templates from which dynamic data will be derived during the simulation.

All static data is directly related to PROMELA and it therefore makes sense to keep it all under a package hierarchy.

package HiSPIN.promela

4.3.5 Expressions

The most simple unit of the PROMELA grammar are expressions. Their common features are summarized in an interface which any class which represents an expression must implement. The most important functionality of an expression is that it can be evaluated in an arithmetical or boolean way, e.g.

$2 + 5 * 4$
$(a > 0)$ && $(b < 6)$

and yield an integer or boolean result. There is a variety of different forms of expressions:

Constants
which evaluate to their value

Unary Operators
which operate on a single operand, e.g. −, ˜, etc.

Binary Operators
which operate on two operands, e.g. +, ∗, /, etc.

Special Operators
which perform some kind of action and return some value as a result, e.g. run, len. These do not share many common features and the cannot really be applied to anything, but act more like statements.

Variable References
which evaluate to the value of the referenced variable

Finally, any expression other than a constant needs a *context*(see Sections 4.3.13 and 5.1.12) to be evaluated. Since both, expressions and operators, comprise quite a few classes each, these are grouped into two packages. For implementation details see Section 5.1.4.

package HiSPIN.promela.expressions

interface HiSPIN.promela.Expression
This interface declares the characteristics of expressions.

package HiSPIN.promela.operators

interface HiSPIN.promela.UnaryOperator
: A unary operator applied to one expression forms a new expression. This interface defines the necessary methods such a unary operator must implement.

interface HiSPIN.promela.BinaryOperator
: Other than a unary operator, a binary operator takes two arguments two make a new expression.

4.3.6 Variables

Variables are a key part of the simulator. The values they store are part of the system state. There are two different different ways variables are used. PROMELA knows several variable types. These must be handled individually.

Declaration
: A declaration tells announces the presence of one or more new variables of a specific type. A variable can only be used after it has been declared.

Reference
: A reference refers to a previously declared variable. During a simulation run, it is used to determine the appropriate instance.

Symbol A symbol represents an untyped variable.

Simple type
: Simple types are the very common types like *bit, bool, int*, etc. A declaration of a simple type may contain multiple symbols of the same type.

Channel Type
: A channel type represents a channel for processes to communicate. See also Section 4.3.14 on message passing.

Compound Type
: Compound types act like containers for simple types. The declaration of a compound type contains several declarations of other types.

User Types
: User Types are one specific case of a compound type. A user type must be defined before it can be used.

All variable related classes are divided into two different packages. This decision is based on the two distinct appliances. See also Section 5.1.5.

package HiSPIN.promela

class HiSPIN.promela.Symbol
: A symbol contains the information about a variable before the type information can be determined. Among these attributes are the name, and whether the variable is a scalar or an array.

package HiSPIN.promela.types

interface HiSPIN.promela.Declaration
A declaration maps the information of what type a symbol is. One declaration may contain multiple symbols. A declaration instantiates all of the symbols it contains.

interface HiSPIN.promela.CompoundDeclaration
A compound declaration add the functionality to administer sub-declarations, also referred to as components.

package HiSPIN.promela.references

interface HiSPIN.promela.Reference
A reference points to a variable which has been declared before. During a simulation run, it manipulates the generated instances.

4.3.7 Statements and Sequences

When a statement is executed, it manipulates the system state, e.g changes the value of a variable. There are situations, when a statement is executable. A statement must provide some means to determine whether it is executable, without changing the system state. Every statement is part of a sequence. A sequence does not provide any functionality but provide a successor for each statement. All statements share the same functionality and are grouped into one package.

package HiSPIN.promela.statements

interface HiSPIN.promela.Statement
Declares all functions a statement implement.

4.3.8 Control Flow Statements

A sequence of regular statements provides exactly one successor for each statement. Control flow statements introduce more flexibility for the path of execution. Compound statements offer multiple possible statements to choose a successor from.

There are two types of compound statements – selections and repetitions. While a selection chooses one path for the execution and then continues with the next statement in the sequence, a repetition will return to its beginning and encounter the same choices over and over again until a *break* statement is encountered.

A label provides the possibility to continue the execution at the statement to which belongs. Unconditional jumps look up a label and take the accompanying statement as the successor. Control flow statements extend the set of regular statements, and are placed in the same package. See also Section 5.1.9.

4.3.9 Proctypes

A Proctype encapsulates a sequence of statements and an arbitrary number of declarations into an entity. A proctype extends the functionality of variables. Additional features are that it takes parameters and has a sequence of statements associated with it. It is also declared and referenced. A proctype allows variables to be declared within. Due to this similarity, a proctype extends a compound declaration.

class HiSPIN.promela.types.Proctype
A declaration which allows parametric instantiation and has a statement sequence associated with it.

4.3.10 The Specification

The specification comprises all elements described in the input protocol specification. It is the result generated by the parser and may contain these units:

- user type definitions
- global variable declarations
- proctype declarations
- the special init process
- a temporal claim

Just like proctype, specification extends a compound type, and will turn into a simulation object upon instantiation. See also Section 5.1.11

class HiSPIN.promela.Specification
The most complex specification and the base for each simulation.

4.3.11 The Simulation

The previous sections introduced the components of a protocol specification. The purpose is to create a simulation from these components. After a protocol specification has been read, a specification object exists. The next step is to begin a simulation run which executes the following steps:

1. Have all active proctypes to instantiate to processes with the simulation as parent.
    (a) Register with the parent. This adds the newly created process to the run queue and creates a new instance event.
    (b) Instantiate all declarations into instances with the process as parent.
    (c) Set the program counter to the first statement in the body.
2. Execute the run-loop
    (a) Is the simulation in an valid end state? → 3
    (b) Are there executable statements in the runlist?
        i. Yes. Let the scheduler pick one execute it. → 2
        ii. No. Signal a timeout.
    (c) Are there are executable statements in the runlist?
        i. Yes. Let the scheduler pick one and execute it. → 2
        ii. No. Deadlock – Abort.
3. Finish.

4.3.12 The Scheduler

The primary task of a scheduler is to keep the run list, a list of all instantiated processes. During a simulation run, it is the scheduler that decides which process gets to execute which statement next. The policy which determines how this decision is reached can be chosen arbitrarily. This is achieved by extending the base class and override the method which picks the next statement.

In case none of the existing processes is able to execute any statement, the scheduler identifies a timeout. It then queries all processes in the run list, if they can execute in a under these circumstances. If all processes are still blocked, the scheduler found a deadlock and aborts the simulation run.

The scheduler interfaces solely the simulation object. The simulation should have a default scheduler to begin with. This avoids an error message in case a simulation run is started without having set up a scheduler. The scheduler should be exchangeable between simulation runs only. Hot swapping the scheduler while a simulation is in progress causes problems and is not feasible for every combination of conceivable scheduling policies. For instance, a session which started interactively cannot be resumed by a replay session of a generated trail.

Since the scheduler keeps the run list and decides which process and statement to execute next, it should also record this information and produce the simulation trails for replay sessions. The development of an undo/redo mechanism seems to be a promising future improvement for the scheduler.

package HiSPIN.simulator.schedulers

abstract class HiSPIN.simulator.Scheduler
Implements the basic functions any scheduler needs to provide.

4.3.13 Dynamic Data

While static data represents the input to the program, dynamic data is generated during a simulation run. The table below shows which static *(compound) types* may be *instantiated* to which dynamic *(compound) instances*. The same static type can be instantiated to multiple instances. For example, multiple processes created from the same proctype may exist at the same time.

Static	**Dynamic**
Instances	
SimpleType	SimpleInstance
ChannelType	ChannelInstance
Compound Instances	
UserType	CompoundInstance
Contexts	
Proctype	Process
Specification	Simulation

Instances no longer belong to the Promela definitions, but to the runtime system – the actual simulator. For this reason, there is one more package hierarchy. It comprises all runtime and instance related classes.

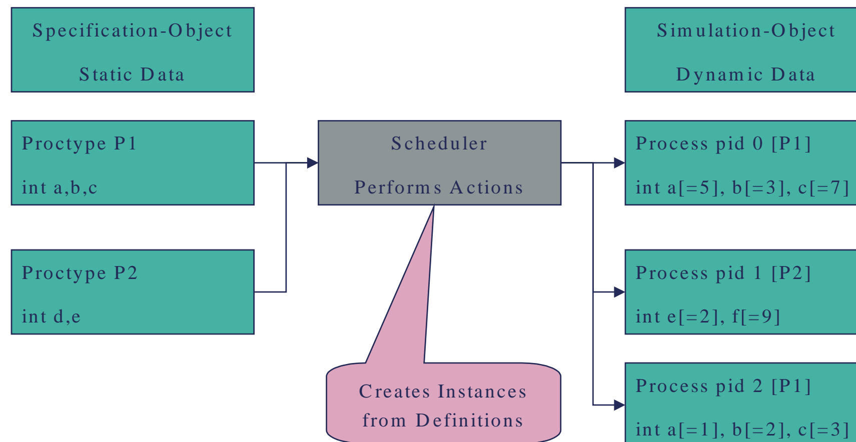

Figure 4.2: The difference between the static specification and the dynamic simulation.

package HiSPIN.simulator

Simple and Channel Instances

A simple instance holds the value for a simple type. It is created upon instantiation of any simple type. It stores an integer value which is casted according to its base type. It does not contain any sub-instances.

Just like a simple instances, a channel instance does not contain any sub instances, also called components. A channel instance stores and deliver but messages, a composite of multiple values. The message format is verified by the channel type from which it was instantiated.

package HiSPIN.simulator.instances

interface HiSPIN.simulator.Instance
This interface defines all functions any kind of instance must offer.

class HiSPIN.simulator.instances.SimpleInstance
A simple instance stores an integer value.

class HiSPIN.simulator.instances.ChannelInstance
A channel instance can store and deliver complex messages.

Compound Instances and Contexts

Compound instances add some more functionality. They are instantiated from compound declarations (see also Section 4.3.6). They contain the instances created from the sub declarations of a compound declaration as components.

For example, an instance of a user type, contains all the instances for the fields of the user type. This is the most simple case. But this functionality is also used by more complex instances.

A context extends the compound instance with the ability to access the simulation object and lookup global variables. This functionality is used by a process and by the simulation itself.

package HiSPIN.simulator.instances

class HiSPIN.simulator.instances.CompoundInstance
: A compound instance is created from a compound declaration. It may contain further instances, components of any kind.

class HiSPIN.simulator.instances.Context
: A context extends a compound instance with the ability to look up global instances also.

class HiSPIN.simulator.instances.Process
: A process is created from a proctype declaration. It provides everything needed to execute its statement sequence.

class HiSPIN.promela.Simulation
: A simulation adds the handling of a simulation run and controlling a scheduler to a context.

4.3.14 Message Passing

Message passing is a rather complex action. Everything related to a variable is used.

1. The declaration of a channel type is told which parameters the channel expects.
2. A send/receive statement is given a reference to the channel.
3. The reference looks up the instance for the channel.
4. The instance checks if the parameters match and stores/delivers the data.

4.4 Event Generation

With the variables, declarations and instances in mind, a strategy for event generation has to be developed. Should there be a single event source, that fires every single event? In that case, the event itself needs to carry descriptive attributes of the model component, the instance, which caused the event. This approach already begins to become contradictory. Each view component needs to register with the event source and decide according to the attributes of the event whether it should react.

It seems wiser to make each model component, each instance, an event source. The view components only receive the events that matter to them. The model components

themselves consist of smaller entities, i.e. variables, that may cause events. The same question appears: should these entities become event sources? This time the situation is completely different. In the previous scenario, the view components have to be aware of the model components. They need not care about the internal structure of a model component, however. The model component to make up the event source is an adequate choice.

What does this mean? Let us have a look at the sample protocols to get the understanding.

4.4.1 Events in the Sliding Window Protocol

By design, the sliding window protocol is intended for point-to-point connections. Hence, there are exactly two view components, one for the sender and one for the receiver. Due to the similarity of these two units, both model components may be represented by objects of the same view component class.

First of all Figure 4.3 identifies what kind of events may occur and how they affect the view component. With these events, a basic visualization can easily be realized. A more complex implementation might choose to add more features. For instance, internal states may be coded with a different or blinking buffer background. We need to find out how and where these events can be placed.

For now, the additional features have to be identified for the sender and the receiver instances. The background color shall change according to the level to which the buffer is filled. It shall be green for 0-90%(relaxed) and red > 90%(tight). For the sender, it shall blink, whenever a packet needs to be resent. Different possibilities to realize these events are analyzed in Section 4.5.

After having looked for protocol specific events, it appears that each category of model components shares some very basic – inherent – events. These are shown in Figure 4.3.

4.5 The Communication Model

The goal of this project is to visualize the progress of an ongoing simulation with independent view components. to this end a communication model has to be developed. The use of events is an adequate choice. The view components register with their model counterparts. Those deliver events whenever the internal state changes.

As PROMELA does not provide native means for other processes (e. g. a view component) to stay informed of the simulation progress, such means will be developed next. Key requirements for such a means are

Occurrence
: Where do the main features reside.

Flexibility
: Can the visual components be substituted by more/less extensive variants.

Changeability
: How much effort is it to alter the specification and/or the visualization.

Re-usability
: To which degree can the visual components be re-used.

Event	Parameters	Description
Sliding Window Events		
store	window size	Buffer slot out of *window size* has been filled with packet *sequence no.*
	position	
	sequence no.	
clear	window size	Buffer slot *position* out of *window size* has been cleared.
	position	
Sliding Window Extensions		
resend	sequence no.	Packet *sequence no.* has been resent.
tight	–	The buffer fills beyond 90%.
relax	–	The buffer-level drops below 90%.
Inherent Processes Events		
create	–	the process has been created
finish	–	the process has finished
blocked	–	no executable option available
resume	–	an option became executable
advance	–	a statement was executed
state	state	label *state* is reached
[as]snd	channel	(a)synchronous message *msg* sent to *channel*
	msg	
[as]rcv	channel	(a)synchronous message *msg* received on *channel*
	msg	
Inherent Channel Events		
empty	–	the last message has been removed
full	–	the last slot in the buffer has been occupied
push	msg	a process stores message *msg* on this channel
pop	msg	a received message *msg* from this channel

Figure 4.3: Events for the sliding window protocol, processes and channels.

Compatibility
Will it affect the usage of the specification with other tools.

To get an impression of how well the model matches the requirements it is checked against the sliding window protocol mentioned in Section 2.4.1

4.5.1 Plain Event Model

The most simple approach is to have the view components register directly with their model counterparts. That way, they have direct access to the model state and can react to any change that occurs.

For the sliding window protocol in the example, this means that the visual component for the proctypes sender and receiver needs to know the name of the variables that

store the window sizes and sequence numbers. It then registers for events that specify sending/receiving events and updates the view.

Occurrence
: The plain event model is completely implemented in the simulator. (+)

Flexibility
: This method is very powerful with regard to customizing single protocol simulation instances. However, it forces the user to be very familiar with the specification which is to be visualized. Since the visual components are tightly coupled to the specification, it is likely, that to improve the visualization new components have to be developed. (−−)

Changeability
: The visual components are tightly coupled with the specification. Therefore, changes applied to one do affect the other. (−−)

Re-usability
: Again, the visual components are tightly coupled with the specification. Therefore, the degree of re-usability is very low. (−−)

Compatibility
: As this approach does not alter the specifications, those can still be used with other tools. (++)

4.5.2 Extending Promela

Another way to provide some means to allow the model components to communicate with the view components is to extend Promela. There are several possibilities how these extensions might look.

The Attribute and Command Approach

In this case a few additional keywords are added to the Promela-grammar. For instance, an attribute *invisible* for proctypes and channels, that tells the simulator that there is no view component for any instance of this declaration. Also useful, non-blocking statements like *hide* and *show* to dynamically change the state of the view component. The major drawback of this solution resides in the chosen complexity of the extensions. Either they are very complex, which makes them difficult to use, or they are not powerful enough.

Occurrence
: The extensions will be placed in the specification and must be recognized by the simulator. (+)

Flexibility
: Once the set of statements and attributes has been determined, all visual components use it. Therefore, they can easily be substituted for each other. However, all visual components are more or less equally powerful. (+)

Changeability
: The visual components need not be adapted, when the specification is altered. (++)

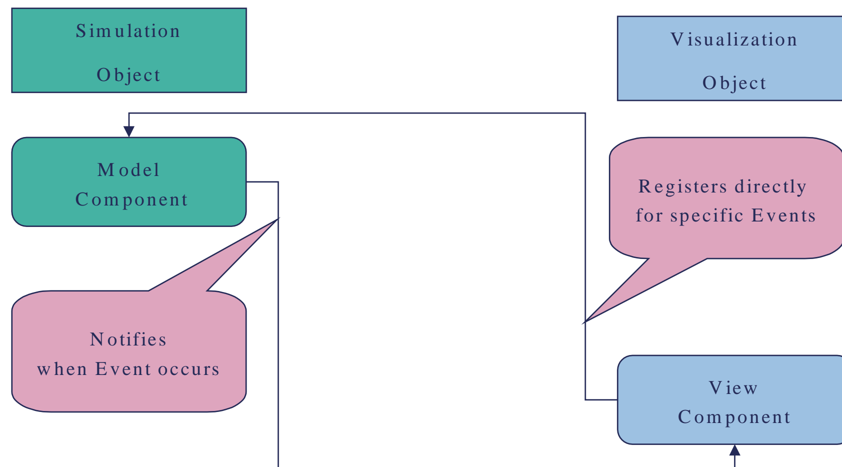

Figure 4.4: Generating events with language elements. The model component generates the event when it encounters a special statement.

Re-usability
: Again, since all visual components use the very same set of events, they can be interchanged at will. (++)

Compatibility
: Existing tools will not understand the new attributes and commands. Once the specification is changed it is rendered unusable for other tools. (−−)

A Generic Event-Directive Approach

Take a look at the attribute and command approach and its major weakness, the lack of different levels of complexity. Assume there is just a single, parametric event directive, that signals the event which is encoded within the parameters. The change in the PROMELA grammar is at a minimum and yet the solution is powerful and flexible. Such an event directive may be very similar to a channel. Instead of having *message types*, there are *event types*.

Occurrence
: The event directive will also be placed in the specification and must be recognized by the simulator. (+)

Flexibility

Once all desired events have been added to the specification, it is no problem to exchange the visual component as long as it supports a subset of the generated events. (+)

Changeability

When the specification is altered, one needs only take care that the set of events does not change dramatically. Otherwise, the visual components need to be adapted. (+)

Re-usability

Visual components can be used with all protocols that have similar sets of events. (++)

Compatibility

The same holds as for the previous approach. The grammar is altered, existing tools will not be able to use the new specification. (−−)

An Event-Channel Approach

Both, this and the following event-comment approach, do not represent an idea on their own. They rather try to get rid of the lack of compatibility the event directive introduces.

As mentioned in the previous section, an event directive is very similar to a channel. So, one could use a handshake channel to write the events to, and have a single process which has exclusive read access to this channel. This process does nothing but loop in an accepting state and receive messages (which are really events) on this one channel. This way the specification will still be readable by existing tools. The simulator can filter all messages on the event channel and deliver them to the visual components. The main evaluation remains the same, only

Occurrence

Again, the event channel will be placed in the specification and must be recognized by the simulator. (+)

Flexibility

see generic event-directive. (+)

Changeability

see generic event-directive. (+)

Re-usability

see generic event-directive. (++)

Compatibility

With some extra effort and a not so clean solution, existing programs can still handle the specification. (+)

```
1  chan event_channel = [0] of { ... };
2
3  proctype event_sink() {
4      xr event_channel;
5  accept:
6      do
7       :: event_channel ? ... ;
8      od;
9  }
10 proctype sliding_window_sender(){
11     boolean buffer[SWS];
12     int ws      = SWS;
13     int seq_no = 0, last_ack = -1, new_ack;
14     do
15      :: ( seq_no < packet_count ) ->
16          if
17           :: ( seq_no - last_ack < ws ) ->
18               connection!data(seq_no);
19               buffer [ seq_no % ws ] = true;
20               event_channel!store(ws, seq_no % ws, seq_no);
21               if
22                :: (((10*( seq_no-last_ack))/ws) > 90 ) ->
23                    event_channel!tight;
24                :: else -> skip;
25               fi;
26               seq_no++;
27          :: connection?ack( new_ack ) ->
28               do
29                :: ( new_ack > last_ack ) ->
30                    last_ack++;
31                    buffer [ last_ack % ws ] = false;
32                    event_channel!clear(ws, last_ack % ws, 1);
33                :: else -> break;
34               od;
35               if
36                :: (((10*( seq_no-last_ack))/ws) < 90 ) ->
37                    event_channel!relax;
38                :: else -> skip;
39               fi;
40          :: timeout ->
41               connection!data( last_ack + 1 );
42               event_channel!resend(last_ack + 1);
43          fi
44      :: else -> connection!fin; break;
45     od;
46 }
```

Figure 4.5: Realizing events with Promela extensions using a designated event-channel

```
1  proctype sliding_window_sender()
2  {
3      boolean buffer[SWS];
4      int ws     = SWS;
5      int seq_no = 0, last_ack = −1, new_ack;
6      do
7       :: ( seq_no < packet_count ) −>
8           if
9            :: ( seq_no − last_ack < ws ) −>
10               connection!data(seq_no);
11               buffer [ seq_no % ws ] = true;

12               //event!store(ws, seq_no % ws, seq_no);

13               if
14                :: (((10∗( seq_no−last_ack))/ws) > 90 ) −>

15                    //event!tight;

16                :: else −> skip;
17               fi ; }
18               seq_no++;
19           :: connection?ack( new_ack ) −>
20               do
21                :: ( new_ack > last_ack ) −>
22                    last_ack++;
23                    buffer [ last_ack % ws ] = false;

24                    //event!clear(ws, last_ack % ws, 1);

25                :: else −> break;
26               od;

27               if
28                :: (((10∗( seq_no−last_ack))/ws) < 90 ) −>
29                    //event!relax;
30                :: else −> skip;
31               fi ;

32           :: timeout −>
33               connection!data( last_ack + 1 );

34               //event!resend(last_ack + 1);

35           fi
36       :: else −> connection!fin; break;
37      od;
38  }
```

Figure 4.6: Using event comments for the communication model.

Event Comments

A slightly better looking solution is, to use exactly the very same event directive, but to wrap it in a comment as shown in Figure 4.6. With very little extra care the solution works quite well. Better still, once the use of the event comments spread, the directives themselves can still be integrated into the grammar with very little effort.

Occurrence
: Once more, the event comments will be placed in the specification and must be recognized by the simulator. (+)

Flexibility
: see generic event-directive. (+)

Changeability
: see generic event-directive. (+)

Re-usability
: see generic event-directive. (++)

Compatibility
: With some extra care, a clean solution. Existing programs can still handle the specification. (++)

4.5.3 Managed Event Model

The goals for the communication model, to be highly customizable and yet easy to use are met by a managed event model. The idea is to introduce a third component, the *Event Manager*, that negotiates between the model and the view components.

Every visual component provides a set of *parameters* and a set of *states*. The latter splits into two different parts. First there are the *basic states* which every component must provide. These may be inherited from the base components. Basic states include visible, hidden, blocked and the like. Further, there are *customized states*. A blinking node may serve as an example for a customized state.

To get a fast non-customized simulation view, only the basic states will be served with the default events. For instance, process instantiation could trigger a *show*-event .

In case of a customized visualization, the user configures the Event Manager when they assign the model component its visual counterpart. The Event Manager queries the visual component for the states it offers. It then demands the user to map these states to states or events of the model component. To stick with the example above, the node could start blinking on arrival of a message with a high priority.

Another useful feature PROMELA provides are *hidden variables*. These serve as temporary, insignificant storage. They do not affect the system state, but they can be evaluated by the simulator at any time. If a user decides that the state of their model component is not detailed enough they may add some hidden variables to have easier access through the Event Manager. This does not affect the correctness of the protocol and any other PROMELA interpreter will still be able to handle the specification.

Flexibility
: At any time can the visual component be replaced by any other. States and parameters need to be re-mapped. (++)

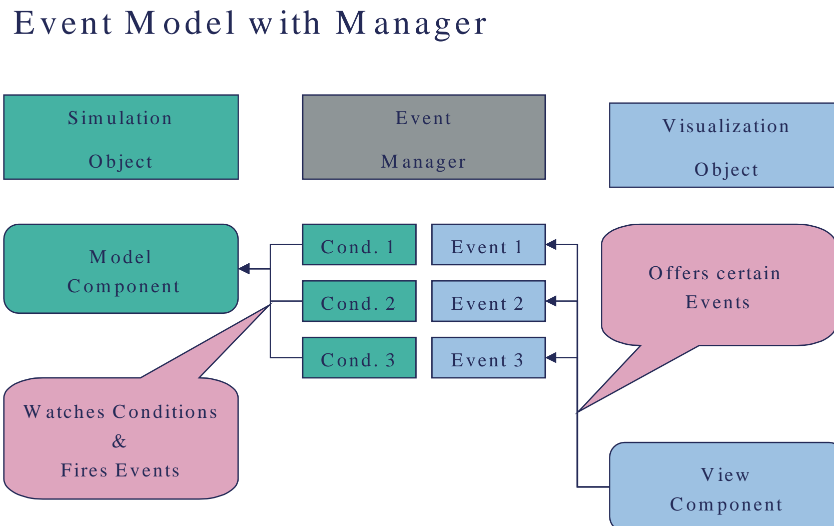

Figure 4.7: The task of the Event Manager.

Changeability
: Both, specification and visualization can independently be changed. If the interface changes, the mapping needs to be adjusted. (++)

Re-usability
: When anything changes, only the mapping of states and parameters needs to be adapted. (++)

Compatibility
: The specification need not be changed and can still be used by any other program. (++)

4.5.4 Event Model Summary

In the previous sections three basic models to implement a communication model have been described and analyzed. Obviously the plain event model does not come close to meet the requirements. Therefore, the focus remains on the Event Manager and the Promela-Extension model.

Let me try to use a picture to illustrate my personal preference. Imagine a bicycle with a single gear. As long as one keeps pedaling on an even road, everything is fine. When the path winds up a mountain, the specific gear might no longer be appropriate. One faces two choices:

Specification	**View**
ws	window_size
snd(connection, msg)	store
$seq_no == msg_field(1)$	
seq_no % ws	store_slot
seq_no	store_data
rcv(connection, msg)	clear
$new_ack > last_ack$	
last_ack % ws	clear_first_slot
$new_ack - last_ack$	clear_slot_count
Extensions	
snd(connection, msg)	resend
$seq_no > msg_field(1)$	
msg_field(1)	resend_data
snd(connection, msg)	tight
$seq_no == msg_field(1)$	
$((10 * (seq_no - last_ack))/ws) > 90$	
rcv(connection, msg)	relax
$new_ack > last_ack$	
$((10 * (seq_no - new_ack))/ws) < 90$	

Figure 4.8: A sample mapping for a visual component for the Sliding Window Protocol.

	Plain	PROMELA-extensions				Manager
		Statements	Directive	Channel	Comment	
Occurrence	Simulator	Specification				Simulator
Flexibility	−−	+	+	+	+	++
Changeability	−−	++	+	+	+	++
Re-usability	−−	++	++	++	++	++
Compatibility	++	−−	−−	+	++	++

Figure 4.9: Summary of the considered Event Model strategies.

1. exchange the cog wheels to alter transmission ratio, or

2. while one is at it, add an entire gear change with multiple cog wheels.

Wile option no. 1 seems easier and less fuzz for the moment, there will never be a one-size-fits-all solution. Choice no. 2, on the other hand, will provide a quite flexible solution for everyday use. Still, if one will head out for a tour in the mountains, it might still be wise to change all the cog wheels to a more appropriate set.

What I am trying to say is: the Event Manager is a flexible solution, which I favor above PROMELA extensions, **but** in the future there might be situations, where it is useful to have the extensions. So, whenever the Managed Event model becomes too limited, there still might be some way to find a solution altering PROMELA.

Chapter 5

Implementation

This Chapter explains the details of the implementation. The focus lies on the functionality of the various parts.

5.1 The Simulator

As mentioned before, the simulator reads the input in two parts. The scanner, also called lexical analyzer, reads the actual input and breaks it into tokens. These tokens are passed to the parser which matches them against given grammar production rules.

5.1.1 The Scanner

JFLex was chosen as the scanner generator. A scanner generator accepts a file which describes the scanner to be generated. The format the description must adhere to differs from generator to generator. In the case of JFLex, the description consists of three parts, separated by *%%* on a separate line.

```
User Code
%%
Options and Declarations
%%
Lexical Rules
```

When JFLex is invoked on a description file, it will produce the source file for a JAVA class. This source file may then be compiled along with the other JAVA source files.

The generated scanner takes a character stream as its input and groups parts of this stream into *tokens*. A token is described with a regular expression and may carry attributes, such as its position in the input stream. The scanner reads the stream character by character and matches them with an internal state machine until a single token has been identified. Every character of the input must be matched, it is not legal to skip any thing to make a match. In case no token can be recognized an error is generated.

Upon a request a token will be handed to the parser. Apparently, the scanner and the parser will need to share the same set of tokens. As mentioned in Section 4.3.2, one reason to choose JFLex was the cooperation with CUP, the parser generator (see Sections 4.3.3 and 5.1.2). All tokens are declared in the input file for CUP, which generates the source file for a JAVA class, which defines a numerical constants for each token.

```
1  // code placed before the scanner class
2  package HiSPIN.promela.parsing;
3
4  import java_cup.runtime.Symbol;
5  import HiSPIN.promela.operators.*;
6  %%
7  %{
8    // this code goes into the Scanner class
9    /**
10    * Creates a new Symbol from the given token and object
11    */
12   private Symbol makeSym(int sym, Object value)
13     {
14       return new Symbol( sym, yychar, yychar + yylength(), value );
15     }
16 %}
17 // options
18 %cup
19 %char
20 // macro
21 const     = [:digit:]+
22 %%
23 {const}  { return this.makeSym( sym.CONST, new Integer( yytext() )  ); }
24 "+"      { return this.makeSym( sym.PLUS, new Add()                  ); }
25 "*"      { return this.makeSym( sym.MULT, new Multiply()             ); }
26 "/"      { return this.makeSym( sym.DIV,   new Divide()              ); }
27 "%"      { return this.makeSym( sym.MOD, new Modulo()                ); }
```

Figure 5.1: Parts of the scanner description file to demonstrate the content different sections

User Code
: This section contains code which will be placed before the actual class definition in the target file, e.g. a package declaration or import statements.

Options and Declarations
: The middle section contains code which is to be included into the defined class as well as *macro* definitions. The usage of regular expressions to match tokens easily leads to complex definitions. A macro replaces an arbitrary regular expression with a simple and hopefully descriptive identifier.

Lexical Rules
: A regular expression and a fragment of JAVA code – an action – to execute make up a lexical rule. The generated scanner will contain a single function which matches the input against the regular expressions from all lexical rules. Whenever this function has recognized a distinct token it will execute the associated action. Since this function is typically called by the parser, the action often just returns the appropriate token.

Figure 5.1 shows very small parts of the scanner for PROMELA input files. Note the function makeSym, which will be used to create a token. The created token will have its position in the input stream added as attributes. In the following sections more and more specific code will be added.

5.1.2 The Parser

Just like the scanner, the parser is also generated from a description. The parser generator used is called **C**onstruction of **U**seful **P**arsers (CUP). CUP also produces a file containing the source of a JAVA class, which represents the actual parser. However, this generated parser also relies on an additional runtime system, which comes along with CUP. Along with the source of the parser class, CUP generates another source file which contains a special class for declaring and handling tokens.

One criteria for choosing the combination of the two generators was how well they are prepared for each other. The parser needs to interact with the scanner to retrieve the recognized tokens one by one. Since JFLex was designed to work seamlessly with several parsers – CUP among them – this interaction is already assured. When passed the corresponding option, the class generated by JFLex anticipates classes and interfaces from the CUP runtime environment, such as java_cup.runtime.Symbol. By this prerequisite, the generated parser can access the scanner without any modification.

Package and Import Specifications

User Code Components

Symbol Lists

Precedence and Associativity Declarations

The Grammar

A description file accepted by CUP consists of up to 5 sections. Unlike JFLex, CUP does not require an explicit separation of the different sections of the description. A more complete manual on CUP, is given at [10].

Package and Import Specifications

These statements will be put at the very beginning of the generated JAVA class source file, before the class definition.

User Code Components

The generated class will also contain an inner – the action – class. Code within the action class is accessible from within the grammar rules.

action code {: ... :}

Everything contained herein will be put into the inner action class.

parser code {: ... :}

This will end up in the main parser class.

init with {: ... :}

Any code which needs to be executed before the parser asks the scanner for the first token should be put in here. This section will end up in a function of its own.

scan with {: ... :}

This section can override the default behavior for how to acquire the next token.

Symbol Lists

Contain all symbol declarations, terminals which are also tokens for the scanner and non terminals which only the parser uses. This section is rendered into a second JAVA class source file.

Precedence and Associativity Declarations

Sometimes multiple reductions are possible. By assigning different precedence values, the highest precedence will be preferred. In this section, terminals are associated with a precedence level.

The Grammar

The actual rules and productions which make up the grammar.

At this point, it shall be enough to know what a description file as input to CUP must look like. In the following Sections, the description will step by step acquire new rules and functionality. However, the first section with the package and import specifications will remain unchanged and should be included right here. The following section will analyze the production rules in the parser and explain the actions and the resulting data structures.

```
package HiSPIN.promela.parsing;

import java_cup.runtime.*;
import HiSPIN.promela.types.*;
import HiSPIN.promela.expressions.*;
import HiSPIN.promela.operators.*;
```

Helpers

Since the final specification object will be some kind of tree a reasonable approach is to have some base classes which provide the basic operations on a tree-like structure. Such structures are tree nodes with multiple and leaf nodes without any children.

package HiSPIN.promela.parsing

interface HiSPIN.promela.ParseTreeElement

function getElement

Access the element associated with this position in the tree.

function getNext

Get the next parse tree element in the tree.

function iterator

Get an iterator for the elements below this one.

function hasNext

Is there an element below this one?

function isLeaf

Is this a leaf node?

5.1.3 Static Data

Static data structures reflect the content of the protocol specification. The parser creates them according the the production rules of the specified grammar. Since the parses depends on the scanner to break the input stream into tokens, the following Sections comprise three parts:

1. Terminals and Scanner Adaption
2. Non Terminals and Productions
3. Functionality

5.1.4 Expressions

Arithmetical and logical expressions make up a small universe of their own within the entity of terminals and productions. This basic set of expressions consist of:

1. Arithmetical Operators
2. Logical Operators
3. Constants

The difference between logical and arithmetical operators may be neglected. A zero-value corresponds to a logical *false*(0), anything else to a logical *true*(1).

Terminals and Scanner Adaption

When introducing a new feature to the parser and the scanner, the first step is to add new tokens, or terminals, to the declaration file for the parser. CUP will then include these terminals into an additional source file when it generates the parser.

```
terminal                                MINUS, SND;
terminal HiSPIN.promela.BinaryOperator  PLUS, MULT, DIV, MOD;
terminal HiSPIN.promela.BinaryOperator  BAND, BOR, LT, GT;
terminal HiSPIN.promela.BinaryOperator  SR, SL, LE, GE, EQ, NE, LAND, LOR;
terminal HiSPIN.promela.UnaryOperator   BCOMP;
terminal Integer                        CONST;
```

The *MINUS* token has no type assigned because it can be both, a unary prefix operator, as in −5, and a binary infix operator, as in 5−3. This distinction is not made by the scanner which will always recognize a − for a *MINUS* token. The production rule according to which the terminal is reduced will make this determination at run time.

```
import HiSPIN.promela.operators.*;
```

Adding the above line to the *User Code* section in the scanner declaration allows to neglect the package part of the operator classes. A macro *const* is defined in the *Options and Declarations* section to match any number of consecutive digits.

```
const     = [:digit:]+
```

As mentioned in Section 5.1.1, the *Lexical Rules* section of the scanner description file consists of regular expressions followed by some code to execute. In this case the regular expressions describe operators and constants. The executable code is simply a return statement. This means the parser will receive the generated token directly when it has been matched.

```
{const}  { return this.makeSym( sym.CONST,  new Integer( yytext() )    ); }

"+"      { return this.makeSym( sym.PLUS,   new Add()                 ); }
"*"      { return this.makeSym( sym.MULT,   new Multiply()            ); }
"/"      { return this.makeSym( sym.DIV,    new Divide()              ); }
"%"      { return this.makeSym( sym.MOD,    new Modulo()              ); }
">>"     { return this.makeSym( sym.SR,     new ShiftRight()          ); }
"<<"     { return this.makeSym( sym.SL,     new ShiftLeft()           ); }
"<="     { return this.makeSym( sym.LE,     new LessEqual()           ); }
">="     { return this.makeSym( sym.GE,     new GreaterEqual()        ); }
"=="     { return this.makeSym( sym.EQ,     new Equal()               ); }
"!="     { return this.makeSym( sym.NE,     new NotEqual()            ); }
"&&"     { return this.makeSym( sym.LAND,   new LogicAnd()            ); }
"||"     { return this.makeSym( sym.LOR,    new LogicOr()             ); }
"&"      { return this.makeSym( sym.LAND,   new LogicAnd()            ); }
"|"      { return this.makeSym( sym.LOR,    new BitwiseOr()           ); }
"<"      { return this.makeSym( sym.LT,     new LessThan()            ); }
">"      { return this.makeSym( sym.GT,     new GreaterThan()         ); }
"~"      { return this.makeSym( sym.BCOMP,  new BitwiseComplement()   ); }
"!"      { return this.makeSym( sym.SND                               ); }
"("      { return this.makeSym( sym.LPAREN                            ); }
")"      { return this.makeSym( sym.RPAREN                            ); }
```

Non Terminals and Productions

The parser requests the tokens from the scanner and derives *non terminals* from them. These non terminals have to be declared, just like the terminals. For expressions, there are only three non terminals used.

```
non terminal HiSPIN.promela.BinaryOperator bin_op;
non terminal HiSPIN.promela.UnaryOperator  un_op;
non terminal HiSPIN.promela.Expression     expr;
```

All the operators are handled in the very same way. There are unary and binary operators, depending on how many operands they take. This means, that they can be

reduced to two non terminals. Also take notice of the precedence declaration for the unary operators.

```
bin_op ::= EQ:op      {: RESULT = op;              :}
         | NE:op      {: RESULT = op;              :}
         | LT:op      {: RESULT = op;              :}
         | GT:op      {: RESULT = op;              :}
         | LE:op      {: RESULT = op;              :}
         | GE:op      {: RESULT = op;              :}
         | MULT:op    {: RESULT = op;              :}
         | DIV:op     {: RESULT = op;              :}
         | MOD:op     {: RESULT = op;              :}
         | PLUS:op    {: RESULT = op;              :}
         | MINUS      {: RESULT = new Minus();     :}
         | BAND:op    {: RESULT = op;              :}
         | BOR:op     {: RESULT = op;              :}
         | LAND:op    {: RESULT = op;              :}
         | LOR:op     {: RESULT = op;              :}
         | SL:op      {: RESULT = op;              :}
         | SR:op      {: RESULT = op;              :};

un_op ::=  BCOMP:op {: RESULT = op;              :}  %prec BCOMP
         | SND      {: RESULT = new LogicNot();  :}  %prec NEG
         | MINUS    {: RESULT = new Minus();     :}  %prec NEG;
```

Finally, the production rules for the simple expressions allow to recognize any arithmetical or logical expression.

```
expr ::= LPAREN expr:e RPAREN {:
           HiSPIN.promela.operators.ParenOperator pop;
           pop = new HiSPIN.promela.operators.ParenOperator();
           RESULT = new SingleArgExpression( e, pop );
      :}

       | expr:l  bin_op:op expr:r
      {: RESULT = new DoubleArgExpression( l, r, op );     :}

       | un_op:op expr:r
      {: RESULT = new SingleArgExpression( r, op );        :}

       | CONST:c
      {: RESULT = new ConstExpression( c );                :};
```

Functionality

Expressions are assembled from constants, operators and expressions. What function these classes need to provide and in which packages they are put can be seen in the list below.

package HiSPIN.promela.operators

interface HiSPIN.promela.UnaryOperator

function test
Returns the result which an execution *would* produce, if the operator was applied right now.
function apply
Applies the operator to the operand(s) and returns the result.

interface HiSPIN.promela.BinaryOperator
function test
Returns the result which an execution *would* produce, if the operator was applied right now.
function apply
Applies the operator to the operands and returns the result.

As can be seen in the production rules for *expr*, there are three different sub-classes of Expressions involved. See below how they derive the results for the important functions.

package HiSPIN.promela.expressions

interface HiSPIN.promela.Expression
function eval
Evaluates the expression with all consequences and returns resulting value.
function test
Calculates the value which an evaluation of the expression would yield. The system state does not change. This function is used to check whether a statement is executable.
function isConstant
Provides the possibility to check whether an expression is a constant.

interface HiSPIN.promela.expressions.ConstExpression
attribute constant
An Integer constant.
function eval
```
constant.intValue()
```
function test
```
constant.intValue()
```

interface HiSPIN.promela.expressions.SingleArgExpression
attribute e
The Expression to which the operator is to be applied.
attribute op
The unary operator which is to be applied to the expression.
function eval
```
op.apply( e.eval () );
```
function test
```
op.test ( e.test () );
```

interface HiSPIN.promela.expressions.DoubleArgExpression
attribute e1
The Expression to the left of the operator

attribute e2
  The Expression to the right of the operator.
attribute op
  The binary operator which is to be applied to the two expressions
function eval

```
op.apply( e1.eval (), e2.eval () );
```

function test

```
op.test ( e1.test (), e2.test () );
```

5.1.5 Variables

Variables are used to store values. PROMELA provides a restricted set of possible integer sizes. Floating point numbers have been omitted deliberately. Thus, the simple types differ only in the number of bits they use to encode their values. Internally, all types use 32-bit integers and mask out the additional bits. If the casted value and the original value differ, and error is generated.

Type	Size in Bits
boolean	1
bit	1
byte	8
short	16
int	32

Additionally to the simple types, PROMELA provides a special channel type which will be examined in more detail later on. For now, the focus will remain on simple value types. Most of the time, all simple types are handled indifferently. One single terminal, *TYPE*, suffices to pass the information from the scanner to the parser.

A variable declaration consists of a type keyword which is followed by a list of one or more variable identifiers. Optionally, the variables may be initialized, assigned an expression in the declaration. An array is denoted by an expression in brackets which follows the variable name. The variables in the list are separated by commas. Some more terminals need to be realized, identifiers, the comma and the assignment operator as well as an opening and a closing bracket.

5.1.6 Terminals and Scanner Adaption

These new tokens needs to be introduced in the *Symbol Lists* section of the parser description.

```
terminal HiSPIN.promela.Declaration    TYPE;
terminal                               ASGN;
terminal                               COMMA;
terminal String                        NAME;
terminal                               LBRACK, RBRACK;
```

The adaption to the section of the scanner are pretty much straight forward.

```
"int"     { return this.makeSym( sym.TYPE, SimpleType.newInteger()  ); }
"short"   { return this.makeSym( sym.TYPE, SimpleType.newShort()    ); }
"byte"    { return this.makeSym( sym.TYPE, SimpleType.newByte()     ); }
```

```
"bool"    { return this.makeSym( sym.TYPE, SimpleType.newBoolean() ); }
"bit"     { return this.makeSym( sym.TYPE, SimpleType.newBit()     ); }

"["       { return this.makeSym( sym.LBRACK                        ); }
"]"       { return this.makeSym( sym.RBRACK                        ); }

","       { return this.makeSym( sym.COMMA                         ); }
"="       { return this.makeSym( sym.ASGN                          ); }
{name}    { return checkNames( yytext()                            ); }
```

The last line uses two unfamiliar pieces, the regular expression *name* and the function checkNames. The regular expression is a macro introduced in the *Options and Declarations* section of the description file for the scanner generator.

```
name            = [a-zA-Z_][a-zA-Z_0-9]*
```

The function *checkNames* simplifies the way keywords are recognized. Keywords do not carry any additional information other than their token id. This allows the token ids to be stored in a hash-map, indexed by the keyword. Since keywords and identifiers are matched by the same regular expression, *name*, a keyword can be recognized if it is present in the lookup table. If the name is in the table, the token stored will be returned, otherwise an *NAME* token is generated. This function is also placed in the *Options and Declaration* section.

```
java.util.Map symbolTable = new java.util.HashMap();

/**
 * checks if a recognized name is a keyword or if it already has an
 * entry in the symbol table. if neither is the case a new entry is
 * added to the symbol table.
 *
 * @param name the name that is to check for
 * @return     the corrsponding token.
 */
private java_cup.runtime.Symbol checkNames( String name )
  {
    int t = sym.NAME;

    if ( this.names.containsKey( name ) )
      {
        /*
         * the name is present as a keyword, so override the token
         */
        t = ((Integer)this.names.get( name )).intValue();
      }

    return makeSym( t, name);
  }
```

Non Terminals and Productions

The parser needs to derive some more non terminals. The declaration takes place in the usual place.

```
non terminal HiSPIN.promela.ParseTreeElement   var_list ;
non terminal HiSPIN.promela.ParseTreeElement   one_decl;
non terminal HiSPIN.promela.Symbol             var , ivar ;
```

Before a variable can be used, it must be declared. At that point the decision is taken of what type the variable is and, if it is an array, how many elements it has. A single declaration may declare a list of variables. For this reason, the name and size of a variable are stored in a *Symbol*, which is then added to the *Declaration*.

Symbol An identifier without any type.

- *name* The name of the variable.
- *size* In case of an array, the size parameter specifies the number of elements to be stored. The default is one.
- *init* An expression used as initializer.

Declaration
A type which declares *Symbols* to be of a certain type.

```
one_decl  ::= TYPE:t var_list:v
          {:
              for( java.util.Iterator it = v.iterator ();
                   it.hasNext();
                   /* nothing */ )
                 {
                        HiSPIN.promela.Symbol symbol;
                        symbol = (HiSPIN.promela.Symbol)it.next();
                        t.addSymbol( symbol );
                 }
                 /*
                  * add these declarations to the current context
                  */
                 parser.theParseTree.addComponent( t );
              RESULT = t;
          :};

var_list ::= ivar:v                      {: RESULT = v;                     :}
         |   ivar:v COMMA var_list:l     {: RESULT = new TreeNode( v, l );  :};

ivar     ::= var:v                       {: RESULT = v;                     :}
         |   var:v ASGN expr:e           {: v. initialize ( e ); RESULT = v; :};

var      ::= NAME:n
         {:  RESULT = new HiSPIN.promela.Symbol( n );                       :}

         |   NAME:n LBRACK CONST:c RBRACK
         {:  RESULT = new HiSPIN.promela.Symbol( n, c.intValue() );         :};
```

When a declaration begins with the recognition of *TYPE* token, a list of variables – symbols – is expected. After this list has been read, symbol is added to the declaration.

Functionality

package HiSPIN.promela

interface HiSPIN.promela.Symbol

attribute getSize / setSize
If the size is greater than one, this symbol represents an array.

attribute getNoOfInstances / setNoOfInstances
How many instances should be automaitcally generated of this symbol. Typically one, zero or any number for a proctype.

attribute initialize / getInitializer
Access the elements with which the symbol was initialized.

package HiSPIN.promela.types

interface HiSPIN.promela.Declaration

function addSymbol / getSymbol
Add / access a symbol within this declaration.

function symbolIterator
Iterate over all symbols

function instantiate
Create instances of all symbols of this declaration.

function duplicate
Simply duplicates a declaration without any symbols.

function addSymbolEventListener
Add a listener for symbol specific events for all symbols within this declaration.

interface HiSPIN.promela.CompoundDeclaration

function addComponent / getComponent
Add / access a component, another declaration within this declaration.

function componentIterator
Iterate over all components.

package HiSPIN.promela.references

interface HiSPIN.promela.Reference

function getIndex / setIndex
References a member of an array.

function getInstanceValue / setInstanceValue

function canSend / canReceive
Tests if a channel is ready to send / receive.

function queue / dequeue
Adds / removes a message from a channel.

function isFull / isEmpty
Checks the status of a referenced channel.

function assign
Assigns a value to the referenced instance.

function eval / test
A reference can also reduce to an expression.
function setField / getFieldReference
A user defined type has components. These functions are used to *dive into* a user type.

5.1.7 User defined Compund Types

The very simple example below shows how PROMELA supports the definition and usage of custom types. The way they are declared is the only thing special about user types. Afterwards they are addressed just like regular variables. Therefore, the reference part is covered in a section of its own, ...

```
typedef custom_type {
  int x,y;              /* two integers: x and y */
  bool state;           /* one booelan           */
}

init {
  custom_type t;        /* declare variable t to be of custom_type */
  t.x = 5;              /* assign a value to field x of t          */
}
```

Terminals and Scanner Adaption

There are only two new terminal symbol introduced for these declarations. However, some more productions are necessary.

```
terminal                TYPEDEF;
terminal                DOT;
```

While *TYPEDEF* is caught as an identifier and turned into the correct token by a simple lookup, the full stop needs a separate rule.

```
%init{
  this.names.put( "typedef" , new Integer( sym.TYPEDEF ) );
%init}

%%
"."        { return this.makeSym( sym.DOT   ); }
```

Non Terminals and Productions

```
non terminal HiSPIN.promela.ParseTreeElement utype;
non terminal HiSPIN.promela.Reference        cmpnd, pfld, sfld;
non terminal HiSPIN.promela.Reference        varref;

utype      ::= TYPEDEF NAME:n
           {:
               HiSPIN.promela.Symbol symbol;
               HiSPIN.promela.CompoundDeclaration decl;
```

```
            decl = new HiSPIN.promela.types.Usertype( n );
            parser.theParseTree.beginContext( n, decl );
        :}
          LBRACE decl_lst RBRACE
        {:
            HiSPIN.promela.ParseTreeElement node;
            parser.theParseTree.endContext();
        :};
```

The existence of user defined data types affects the declarations introduced in the previous section. The parser cannot include the names of all data structures at compile time. Instead, it must accept that any name might represent a data type when it is used in a certain context.

Of course, this must be verified and wrong names must be recognized. When a user type is declared the parser notifies the specification. The specification keeps track of all these definitions. When a declaration with a name is derived, the parser requests the definition of this type from the specification. This only succeeds if the name has been declared as a user type before.

```
one_decl ::=  NAME:n var_list:v
        {:
            HiSPIN.promela.Declaration decl;

            decl = parser.theParseTree.getComponent( n );
            decl = decl.duplicate();

          for( java.util.Iterator it = v.iterator();
               it.hasNext();
               /* nothing */ )
            {
                   HiSPIN.promela.Symbol symbol;
                   symbol = (HiSPIN.promela.Symbol)it.next();
                   decl.addSymbol( symbol );
            }
            parser.theParseTree.addComponent( decl );
          RESULT = decl;
        :};

varrefs     ::= varref
            |   varref COMMA varrefs ;

varref      ::= cmpnd:c
            {:  RESULT = c; this.activeRef = null;                    :};

cmpnd       ::= pfld:p
            {:
                if( this.activeRef == null ){
                       this.activeRef = p;
                } else {
                       this.activeRef.setField( p );
                }
                RESULT = p;
```

```
              :}   sfld :s   ;

pfld          ::= NAME:v                     {: RESULT=this.lookUpRef(v, null);  :}
              |   NAME:v LBRACK expr:e RBRACK {: RESULT=this.lookUpRef(v, e); :}
              ;

sfld          ::= /* empty */
              |   DOT cmpnd;
```

5.1.8 Statements and Sequences

The set of statements can be divided into two subsets, simple statements and control flow statements. Due to extra-ordinary features, the next Section (5.1.9) covers control flow statements. Statements never appear by themselves, but come in sequences. A sequence can also contain declarations at arbitrary points.

Terminals and Scanner Adaption

For the simple statements and sequences, new tokens need to be introduced in the usual place.

```
terminal                     ASSERT, PRINT, TIMEOUT, UNLESS;
terminal                     STRING;
terminal                     SEMI;
```

The scanner matches all of these tokens except for *STRING* with the checkNames function. In the initialization section, the keywords and their corresponding tokens are entered into the lookup table. Whenever a name is recognized, it is checked whether it matches a keyword in the lookup table.

```
%{init
  this.names.put( "assert"   , new Integer( sym.ASSERT  ) );
  this.names.put( "printf"   , new Integer( sym.PRINT   ) );
  this.names.put( "timeout"  , new Integer( sym.TIMEOUT ) );
  this.names.put( "unless"   , new Integer( sym.UNLESS  ) );
%init}
```

A *STRING*, which is the first argument for a *print* statement needs some special treatment. A *SEMI* token separates two statements in a sequence. Some more macro is defined and matched.

```
string          = \"␣~␣([^\\]\")
semi            = ";" | "->"
%%
{semi}    { return this.makeSym( sym.SEMI                 ); }
{string}  { return this.makeSym( sym.STRING, yytext()     ); }
```

Non Terminals and Productions

The parser gains another non terminal for statements. The last two non terminals provide some comfortable freedom to the creator of a protocol specification. They allow the usage of multiple separators instead of insisting on the exact number.

```
non terminal HiSPIN.promela.ParseTreeElement sequence;
non terminal HiSPIN.promela.Statement        stmt, step;
non terminal                                 opt_semi, mult_semi;
```

A sequence consists of a step followed by either an optional semicolon or multiple semicolons and another sequence. The usage of tree nodes generates a list. The production given for a *step* allows declarations to appear at any place in a sequence. The special expressions like *run* and *len* fit in here better.

```
sequence  ::= step:s opt_semi
          {:  RESULT = new HiSPIN.promela.parsing.TreeNode( s, null );     :}

          |   step:s mult_semi sequence:q
          {:
                if( s != null ) {
                    s.followUp( ((HiSPIN.promela.Statement)q.getElement()),
                                this.spec );
                    RESULT = new HiSPIN.promela.parsing.TreeNode( s, q );
                } else {
                    RESULT = q;
                }
          :};

step      ::= one_decl
          |   stmt:s1 UNLESS stmt:s2
          {:
              s1.setSeqNo( this.spec.seqNo() );
              s1.setCondition( s2 );
              RESULT = s1;
          :}

          |   stmt:s
          {:  s.setSeqNo( this.spec.seqNo() );  RESULT = s;                 :};

stmt      ::= varref:v ASGN expr:e
          {:  RESULT = new HiSPIN.promela.statements.Assignment( v, e );   :}

           | varref:v INCR
          {:  RESULT = new HiSPIN.promela.statements.Increment( v );       :}

           | varref:v DECR
          {:  RESULT = new HiSPIN.promela.statements.Decrement( v );       :}

           | PRINT LPAREN STRING:s prargs:p RPAREN
          {: RESULT = new HiSPIN.promela.statements.Print( s, p );         :}

           | ASSERT expr:e
          {: RESULT = new HiSPIN.promela.statements.Assertion( e );        :}

           | expr:e
          {: RESULT = new HiSPIN.promela.statements.Condition( e );        :};

expr     ::= RUN NAME:n LPAREN args:a RPAREN
          {:
```

```
            HiSPIN.promela.references.ProcessReference ref;
            ref = new HiSPIN.promela.references.ProcessReference
                            ( n, this.spec );

            RESULT = new Run( ref, a);
        :}

        |   LEN LPAREN varref:v RPAREN {: RESULT = new Len( v );      :};
```

Some more minor productions which affect statements and sequences. The print statement does not only take a string as an argument. It inserts expressions at specific places into this string.

```
opt_semi   ::=  /*empty */
           |    SEMI;

mult_semi  ::=  SEMI
           |    mult_semi SEMI ;

prargs     ::= /* empty */
           |    COMMA arg:a           {: RESULT = a;                    :};

arg        ::= expr:e                 {: RESULT = e;                    :}
           |    expr:e COMMA arg:a    {: RESULT = new TreeNode(e, a);   :};
```

Functionality

A statement needs to offer the following methods. In the next section control flow statements will be added.

package HiSPIN.promela.statements

interface HiSPIN.promela.Statement

function getSeqNo / setSeqNo
: Access the sequence number of a statement. The sequence number is used to determine the system state.

function getLabel / setLabel
: Access the label associated with a statement.

function followUp / getSuccessor
: Access the next statement in the sequence.

function isExecutable
: Tests whether a statement is executable under the current conditions.

function execute
: Executes the statement.

function optionList
: Returns all executable options the statement offers.

function isJump
: Checks whether this statement is a jump.

5.1.9 Control Flow

A sequence determines the order in which statements are executed. Control flow elements manipulate this order. Unconditional jumps allow to continue at an arbitrary place in the sequence. Compound Statements introduce non deterministic choices how to continue. Two more keywords make a sequence an indivisible unit, so that they will execute in one step without being interrupted by another process.

Terminals and Scanner Adaption

Control Flow elements introduce quite a few new tokens. The braces are used to group sub-sequences. A colon after a name identifies a label, which attaches to the following statement. The double colon separates the different options in a compound statement.

```
terminal        LBRACE, RBRACE;
terminal        ATOMIC, D_STEP;
terminal        IF, FI, DO, OD, SEP, ELSE, BREAK;
terminal        GOTO, COLON;
```

To make the scanner recognize the keywords, the symbol table gets some more entries.

```
%{init
  this.names.put( "atomic"  , new Integer( sym.ATOMIC  ) );
  this.names.put( "d_step"  , new Integer( sym.D_STEP  ) );

  this.names.put( "goto"    , new Integer( sym.GOTO    ) );
  this.names.put( "if"      , new Integer( sym.IF      ) );
  this.names.put( "fi"      , new Integer( sym.FI      ) );
  this.names.put( "do"      , new Integer( sym.DO      ) );
  this.names.put( "else"    , new Integer( sym.ELSE    ) );
  this.names.put( "od"      , new Integer( sym.OD      ) );
  this.names.put( "break"   , new Integer( sym.BREAK   ) );
  this.names.put( "timeout" , new Integer( sym.TIMEOUT ) );
%init}
%%

"{"     { return this.makeSym( sym.LBRACE     ); }
"}"     { return this.makeSym( sym.RBRACE     ); }
":"     { return this.makeSym( sym.COLON      ); }
"::"    { return this.makeSym( sym.SEP        ); }
```

Non Terminals and Productions

Control Flow elements extend existing definitions and introduce options for Compound statements.

```
non terminal HiSPIN.promela.ParseTreeElement option, options;
```

The previous section already added statements and sequences. Most control Flow elements fall exactly into these categories.

```
sequence ::= step:s opt_semi
          {:    RESULT = new TreeNode( s, null );                :}
```

```
            |   step:s multi_semi sequence:q {:
                    if( s != null ) {
                        s.followUp( ((HiSPIN.promela.Statement)q.getElement()),
                                    parser.theParseTree );

                        RESULT = new HiSPIN.promela.parsing.TreeNode( s, q );
                    } else {
                        RESULT = q;
                    }
             :};

stmt        ::= DO
            {:      parser.theParseTree.pushBreak();                              :}
                options:o OD        {:
                    String bl;
                    HiSPIN.promela.Symbol symbol;
                    HiSPIN.promela.statements.Repetition rep;

                    bl  =  parser.theParseTree.breakLabel();
                    rep =  new HiSPIN.promela.statements.Repetition( o, bl );
                    parser.theParseTree.popBreak();
                    parser.theParseTree.seqNo();
                    parser.theParseTree.seqNo();

                    RESULT = rep;
            :}

            |   BREAK               {:
                     RESULT = new HiSPIN.promela.statements.Break(
                            parser.theParseTree.breakLabel(),
                            parser.theParseTree.getActiveContext() );
            :}

            |   IF options:o FI     {:
                    RESULT = new HiSPIN.promela.statements.Selection( o );
                    parser.theParseTree.seqNo();
            :}

            | NAME:n COLON stmt:s {:
                    HiSPIN.promela.Symbol symbol;
                    HiSPIN.promela.Declaration decl;

                    symbol = new HiSPIN.promela.Symbol( n, s );
                    parser.theParseTree.addLabel( n, s );
                    s.setLabel( n );
                    RESULT = s;
            :};

options     ::= option:o            {: RESULT = new TreeNode( o, null ) ;         :}
            |   option:o options:s  {: RESULT = new TreeNode( o, s );             :};

option      ::= SEP sequence:s      {: RESULT = s;                                :};
```

5.1.10 Proctypes and other Units

The Statements and Sequences introduced in the previous Sections cannot appear at any point in a protocol specification. Their appearance is restricted to within proctype definitions. These include the special proctype *init* and an optional never-claim.

Proctypes provide a wrap a sequence into a unit. However, these are not the only types of units. Type, like message type or user type, definitions as well as global variable declarations are also units.

Terminals and Scanner Adaption

The terminals which identify units are exclusively keywords. The tokens are declared in the parser and added to the lookup table upon initialization of the scanner.

```
terminal        INITP, PROCTYPE;
terminal        ACTIVE, PRIORITY, PROVIDED;
terminal        CLAIM;
terminal        MTYPE, TYPEDEF;
```

```
this.names.put( "init"     , new Integer( sym.INITP     ) );
this.names.put( "active"   , new Integer( sym.ACTIVE    ) );
this.names.put( "mtype"    , new Integer( sym.MTYPE     ) );
this.names.put( "proctype" , new Integer( sym.PROCTYPE  ) );
this.names.put( "priority" , new Integer( sym.PRIORITY  ) );
this.names.put( "provided" , new Integer( sym.PROVIDED  ) );
```

Non Terminals and Productions

The same procedure as usual is followed to declare the non terminals.

```
non terminal HiSPIN.promela.ParseTreeElement   unit, units;
non terminal HiSPIN.promela.ParseTreeElement   mtype, utype;
non terminal HiSPIN.promela.ParseTreeElement   initp, proc, claim;
non terminal                                   priority, enabler;
non terminal HiSPIN.promela.ParseTreeElement   body;
```

Every unit provides their own context. Proctype and Usertype definitions allow declarations within and may not be nested. For these reasons, whenever such a *context* is opened, the final specification object is notified about this. When nesting is attempted, the specification produces an error. The specification also assigns optional parameters to a context.

```
unit    ::= mtype
          | utype
          | one_decl
          | initp
          | proc
          | claim
          | SEMI
          | error
          ;

mtype   ::= MTYPE ASGN LBRACE args:a RBRACE
```

```
            {:  this.spec.setMType( a );                                  :};

initp       ::=  INITP {: this.beginContext( ":init:" , new Integer(1) );   :}
                 body:b {: this.spec.endContext( new TreeNode( null, b ) ); :};

proc        ::= inst : i  PROCTYPE NAME:n
            {:     this.beginContext( n, i  );                               :}
                LPAREN decl:d RPAREN priority:p enabler:e body:b
            {:     this.spec.endContext( new TreeNode( d, b ), p, e );       :};

inst        ::=                                  {: RESULT = new Integer( 0 ); :}
            |   ACTIVE                           {: RESULT = new Integer( 1 ); :}
            |   ACTIVE LBRACK CONST:c RBRACK {: RESULT = c;                  :};

claim       ::= CLAIM
            {:
                 String n = ":never:";
                 HiSPIN.promela.Symbol symbol;
                 HiSPIN.promela.CompoundDeclaration decl;

                 symbol = new HiSPIN.promela.Symbol( n );
                 symbol.setNoOfInstances( new Integer( 0 ) );

                 decl = new HiSPIN.promela.types.Proctype();
                 decl.addSymbol( symbol );

                 this.spec.beginClaim( decl );
            :}
                body:b
            {:   this.spec.endContext( new TreeNode( null, b ) );           :};

priority    ::= /* empty */
            |   PRIORITY CONST:c           {: RESULT = c;                   :};

enabler     ::= /* empty */
            |   PROVIDED LPAREN expr:E RPAREN {: RESULT = e;                :};

utype       ::= TYPEDEF NAME:n
            {:
                 HiSPIN.promela.Symbol symbol;
                 HiSPIN.promela.CompoundDeclaration decl;

                 decl = new HiSPIN.promela.types.Usertype( n );
                 this.spec.beginContext( n, decl );
            :}
                 LBRACE decl_lst RBRACE
            {:   this.spec.endContext();                                     :};


body        ::= LBRACE sequence:s RBRACE
            {:
                 HiSPIN.promela.ParseTreeElement node;
                 HiSPIN.promela.Statement last = null, term;
```

```
        RESULT = new TreeNode( s, null );

        for( java.util.Iterator it = s.iterator();
                it.hasNext();
                /* nothing */ )
        {
                last = ((HiSPIN.promela.Statement)it.next());
        }
        term = new HiSPIN.promela.statements.Termination();
        term.setSeqNo( this.spec.seqNo() );
        last.followUp( term, this.spec );
:};
```

Functionality

A proctype extends a compound declaration with the ability to handle parameters and a statement sequence.

interface HiSPIN.promela.types.Proctype

function getParameters
: Provides access to the parameter list for the proctype.

function firstStatement
: Gives acces to the first statement in the body.

5.1.11 Specification

Finally, the specification comprises the entire input. It accepts any number of units arranged in any order.

Terminals and Scanner Adaption

The specification derives from non terminals only. No new terminals need to be introduced and the scanner does not change any more.

Non Terminals and Productions

The last two non terminals

```
non terminal HiSPIN.promela.Specification    program;
non terminal HiSPIN.promela.ParseTreeElement units;
```

The details have already been dealt with in the lower level productions. These two merely allow the arrangement of multiple units and provide a target to reduce. The parse function returns whatever program has as a value.

```
program  ::= units       {: RESULT = this.spec; :};

units    ::= unit
           |  units unit ;
```

Functionality

interface HiSPIN.promela.Specification

function beginContext / beginClaim / endContext
Opens a context during the parsing stage.
function setMType
Provides numerical constants.
function pushBreak / popBreak / breakLabel
Handles break labels for nested loops.
function addLabel
Keeps track of labels and the associated statement.
function seqNo
Returns the sequence number for the next statement.

5.1.12 Dynamic Data

Instances Functionality

Instances are used to handle runtime information for simple types and channel types. Here is a brief description of their functionality.

package HiSPIN.simulator.instances

interface HiSPIN.simulator.Instance
function duplicate
Duplicate this instance with the same values. Used for sending messages.
function replicate
Overwrite the values from a clone. Used for receiving messages.
function setIndex
For array members.
function setParent / getContext / getSimulation
Used to determine the global context.

interface HiSPIN.simulator.SimpleInstance
function setValue / getValue
Access the integer value of the instance.

interface HiSPIN.simulator.ChannelInstance
function isFull / isEmpty / canSend / canReceive
Query the state of the channel.
function queue / dequeue
Perform a channel action.
function getTypeList
Determine the message format for this channel.
function makeMessage
Generate a message from given arguments.

Complex Instances

A compound instance is instantiated from a compound declaration. Since a compound declaration contains further declarations, the compound instance contains the instances created from these sub-declarations.

A context adds the ability to look up a variable globally if it is not present. The simulation adds control functionality for the simulation run.

package HiSPIN.simulator.instances

interface HiSPIN.simulator.instances.CompoundInstance

function addInstance / getInstance
Access component instances.

function getEmptyClone
Create a clone without the data of this instance.

function instanceIterator
Iterate over the components.

interface HiSPIN.simulator.instances.Context

function lookup
Lookup a symbol.

function canReceive
Checks if this context is able to receive a certain message. Used for Rendez-Vous communication.

interface HiSPIN.simulator.instances.Process

function hasChildren
A process cannot terminate if other processes which it created still exist.

function terminate
Stop the running process and clean up.

interface HiSPIN.promela.Simulation

function queue / dequeue
Handle process creation and termination.

function begin / run / trace / pause / resume / abort
Control the simulation run.

function getPendingPid
Keep track of running processes.

5.1.13 Scheduler Functionality

The scheduler keeps the runlist for the simulation. Upon request it chooses a new process and statement to execute. There is a variety of strategies how to accomplish this. The implemented method is to pop up a user dialog.

package HiSPIN.simulator.schedulers

interface HiSPIN.simulator.Scheduler

function queue / deque
Add or remove a process from the runlist.

function pickNext
Choose the next process to run.

function pickOption
Pick a specific option to execute.
function runnableList
Get a list of all runnable processes.
function timeoutList
Get a list of all processes, which can handle a timeout.
function canReceive
Check if any process can execute a receive statement for a specific message.
function abort
Abort the current run and clean up.

Chapter 6

Conclusion & Outlook

Finally, it is time to recapture what has been accomplished with this work, how it fits into the HiSPIN project. Additionally, it is worth a closer look what still remains to be done and which new perspectives have been gained.

6.1 Conclusion

The main goal of this work was to set a base for improving the two-step process from a specification to visualization. This has been accomplished by creating a JAVA implementation of the SPIN simulator. This implementation offers a newly created interface for an event based visualization.

So far, the simulation is controlled interactively. Further possibilities have been considered during the design phase, so adding new ones will be easy. The current state of the program is, that PROMELA specifications can be edited and simulated. During the simulation several events are generated and displayed.

6.2 Outlook

For the future, there is still plenty of work to be done. The most important part will be to implement the actual visualization engine with the event manager. The development of a toolkit for the visual components to handle the events from the simulation in progress seems to be the most important part.

Another big issue is the implementation of the validating algorithm. This feature will render the program completely independent from SPIN. Although, at least for the moment, there seems to be a performance issue with a JAVA implementation for this kind of application.

Bibliography

[1] **Tanenbaum, A. S.**, *Computer Networks, Third Edition*, Prentice Hall, 1996

[2] **Aho, A. V, R. Sethi, J. D. Ullman**, *Compilers Priciples, Techniques and Tools*, Addison-Wesley, 1988

[3] **Holzmann, G. J.**, *Design and Validation of Computer Protocols*, Prentice-Hall International, 1991
http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html

[4] **Holzmann, G. J.**, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, Vol. 23, NO. 5, May 1997

[5] **Brodbeck, T.**, *Grundlegende Überarbeitung der Teile des Java Visualisierungsbaukastens für Protokolle*, Studienarbeit, Universität Stuttgart, 1999

[6] **Brodbeck, T.**, *Erweiterung des Animationssystems HiSAP um Synchronisationsmechanismen auf der Grundlage logischer Zeit*, Diplomarbeit, Universität Stuttgart, 2000

[7] **-**, *Online Promela Language Reference*, (Online)
http://cm.bell-labs.com/cm/cs/what/spin/Man/promela.html

[8] **-**, *JFlex - The Fast Scanner Generator for Java*, (Online)
http://www.jflex.de

[9] **-**, *CUP Parser Generator for Java*, (Online)
http://www.cs.princeton.edu/ appel/modern/java/CUP/

[10] **-**, *CUP User's Manual*, (Online)
http://www.cs.princeton.edu/ appel/modern/java/CUP/manual.html

[11] **-**, *HiSPIN*, (Online)
http://www.informatik.uni-stuttgart.de/ipvr/vs/en/projects/HiSPIN/

[12] **-**, *Project HiSAP*, (Online)
http://www.informatik.uni-stuttgart.de/ipvr/vs/en/projects/HiSAP/

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Matthias Papesch)