The Pomegranate compiler for the Erigone model checker

Author:
Trishank Karthik Kuppusamy[1]

Advisor:
Professor Edmond Schonberg[2]

Second Reader:
Professor Benjamin Goldberg[3]

Second Reader:
Professor Mordechai Ben-Ari[4]

Submitted in partial fulfillment of requirements for the Master of Science in Computer Science degree, Courant Institute of Mathematical Sciences, New York University

May 2009

[1] MSCS candidate, Courant Institute of Mathematical Sciences, New York University.

[2] Professor Emeritus, Department of Computer Science, Courant Insitute of Mathematical Sciences, New York University. Vice President, AdaCore, Inc.

[3] Associate Professor, Department of Computer Science, Courant Insitute of Mathematical Sciences, New York University.

[4] Associate Professor, Department of Science Teaching, Weizmann Institute, Israel.

Contents

Acknowledgements

I wish to thank Professor Edmond Schonberg for his sage advice and guidance on the compiler, Professor Mordechai Ben-Ari for the collaboration on the Erigone model checker and Professor Benjamin Goldberg for reviewing my thesis.

Last but not least, I must thank my family, especially my young cousins Ram and Nandika, for tolerating my work on the compiler over my vacations with them. This thesis is especially dedicated to my small family — my father Kuppusamy Balasubramaniam, my mother Soundararajan Venkata Jayakumari and my brother Trishank Navdeep — without whom this and everything else would be impossible.

Abstract

In which we discuss Pomegranate, a compiler written in Ada 2005 that translates a subset of Promela, the programming language of the SPIN model checker, for the Erigone model checker.

Chapter 1

Introduction

Day by day, software systems are growing to be more ubiquitous in our daily lives while also in complexity. Yet unlike some other criticial human enterprises, most software do not specify any guarantee of correctness of behaviour on their part. In fact, most such software go out of their way to specify in lengthy and inscrutable legal agreements that the user must accept that no correctness in behaviour is ever promised by the developers of the said software[1].

To be fair, correct and secure software engineering is a difficult scientific art that requires considerable education, experience and skills. More generally speaking, incompleteness theorems such as the halting problem prevent us from ever establishing the correctness of all possible computer programs[4]. To add further injury to our pride, randomness in Nature also has a special gift for rendering useless some of our most well-considered models of physical systems[8]. Is there any way at all to alleviate the conundrum?

1.1 Model Checking

The answer to the question for some software is the technique of *model checking*.

A model checker accepts a model which is a description of a physical, usually concurrent and computerized, system. Next, correctness properties are specified for which the model must hold against. Finally, the model checker either verifies the correctness of the model or demonstrates the incorrectness of the model by producing at least one counterexample which fails the given properties of correctness[3][Chapter 1].

A simple model checker is easy enough to construct. All it has to do is to enumerate all possible states of a given model and check its correctness properties in every state. However, even the introduction of a simple 32-bit integer to the model would completely throw off this simple model checker simply due to the fact that the integer may cause 2^{32} possible states. In practice, sophisticated and clever algorithms from automata and logic theory are needed to fight such tremendous odds against computational complexity. The interesting challenge in authoring a model is that it must simultaneously be complex enough to capture the details of original system and yet be simple enough to fit the scope of

[1] Some software make the matter considerably worse by presenting themselves as impenetrable black boxes with no accompanying source code for the inspection of the user.

a model checker[3][Preface].

We now present 3 classes of problems[6][Chapter 1] for which model checking would be an appropriate solution.

- *Circular blocking*: An example of this category of problems would be the traffic protocol in some European countries. An implicit rule in Netherlands states that at unmarked intersections, traffic approaching from one's right have the right of way. Unfortunately, as pictured by Figure 1.1, this rule can lead to circular blocking. This problem cannot even be avoided by traffic lights which regularly reverse priority rules. On the other hand, a fixed priority rule would lead to the classic starvation problem. While in everyday life such problems may be solved by breaching legal protocols, such a solution is not appropriate for software systems.

- *Deadly embrace*: This category of problems is similar if not identical to circular blocking. Computer scientists will be familiar with an instance of this category, which is the problem of resource allocation to processes in operating systems as pictured in Figure 1.2. Another familiar instance is the problem of two telephone users trying to call each other at the same time.

- *Mismatched assumptions*: Sometimes, certain otherwise well-engineered components do not perform as expected when interacting with other components in a larger system. An example of this problem was the tragedy involving a Lufthansa Airbus A320-200 airplane at the Warsaw airport in Poland on September 14 1993. The airplane was attempting to land on the runway in the midst of heavy rain and it was not receiving good traction from the wheels on the landing gear. The pilots thought that they could count on the thrust reversers on the main engines to slow down the airplane but unfortunately, the thrust reversers failed to deploy in time, causing the airplane to overshoot the end of the runway. 2 people lost their lives in the tragedy. In order to prevent accidents during flight, the thrust reversers will function only when 3 conditions are met: the landing gear must be down, the wheels must be turning and the weight of the airplane must be bearing on the wheels. Unfortunately, the landing gear was down, but the wheels were hydroplaning and a tailwind provided sufficient lift on the wings of the airplane. The control software did not decide that the airplane had actually landed until 9 seconds after touchdown. It would be difficult to blame the tragedy solely on the engineering of the airplane given the poor human foreseeability of such unusual circumstances[8]. This is where a model checker could be put to very good use: in the verification of a software given all possible conditions no matter how unlikely an event may be.

An excellent philosophical and technical treatise on the design and verification of computer protocols may be found in [5].

1.2 The SPIN model checker

SPIN is a popular open-source model checker that can be used for the formal verification of concurrent software systems. It is considered to be one of the most

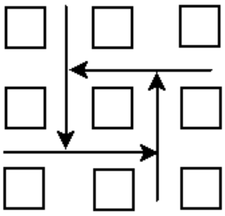

Figure 1.1: Circular blocking

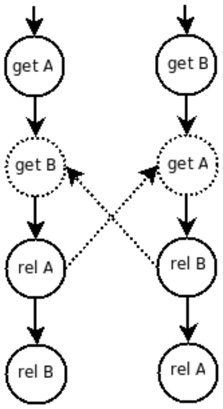

Figure 1.2: Deadly embrace

powerful model checkers around. SPIN was primarily developed, beginning in 1980, by Gerard J. Holzmann at Bell Labs in the original UNIX group of the Computing Sciences Research Center. SPIN has been publicly available as open source software[2] since 1991 and continues to evolve to keep pace with new developments in the field. (Recent versions have seen the adaption of SPIN to multi-core processors.) In April 2002, SPIN was awarded the prestigious System Software Award for 2001 by the Association for Computing Machinery (ACM).

The name SPIN was chosen because it stood for Simple Promela INterpreter, although it has long since outgrown two of three of these terms. Promela is the programming language that is used to specify to the SPIN model checker *communicating processes* as a model of a physical system.

Three interesting applications of SPIN are:

- *Flood control*: SPIN was used to verify the control algorithms for the new flood control barrier built in the late 1990s near Rotterdam, Netherlands.

- *Call processing*: In what was perhaps the largest application of model checking to that date, SPIN was used to verify the call processing software for a commercial data and phone switch built at Lucent Technologies. A cluster of 16 CPUs was used to perform the verifications overnight everyday for a few months before the switch was marketed.

[2] http://www.spinroot.com

- *Mission critical software*: SPIN was used to verify and select algorithms for a few space missions by the National Aeronautics and Space Administration (NASA). The missions include Deep Space 1, Cassini, the Mars exploration rovers, Deep Impact and so on. For the Cassini mission, SPIN verified the correct behaviour of the handoff algorithms for dual-control CPUs. For the Deep Space 1 mission, researchers at the Ames Research Center used SPIN to verify certain key algorithms. For the Mars exploration rovers mission, SPIN verified the correct behaviour of the resource arbiter that manages the use of all motors on the rovers. Finally, for the Deep Impact mission, SPIN verified aspects of the flash file system module that had demonstrated problems before the encounter with the comet.

1.2.1 Promela

Promela is the programming language that is used to specify to SPIN a model of communicating processes. Although a fair share of its syntax and semantics is borrowed from the C programming language, there are certainly significant differences between the languages.

Promela data types include primitive data types, user-defined data types (which are fairly similar to the *struct* construct of the C language) and 1-dimensional arrays of data types. The primitive data types are easy to understand: *bit/bool* are 1-bit data types, *byte* is an 8-bit data type, *mtype* allows for enumeration of constants (similar to the *enum* construct of C), *short* is a 16-bit integer, *integer* is a 32-bit integer and *unsigned* is an unsigned integer up to 31 bits in size as specified by the user. Higher-dimensional arrays may be simulated with user-defined types.

```
active proctype p()
{
  if
    :: 1      ->  printf( ”p1” );
    :: 2      ->  printf( ”p2” );
    :: 3 == 1 ->  printf( ”p3” );
  fi;
}

active proctype q()
{
  do
    :: 4      ->  printf( ”q4” );
    :: 5      ->  printf( ”q5” );
  od;
}
```

Listing 1.1: A trivial example of nondeterminism in Promela

Unlike some other programming languages, Promela allows for some interesting *nondeterministic* control statements. There are 5 kinds of control statements: sequence, selection, repetition, jump and *unless* statements. A sequence of statements is simply a linear chain of statements in which one statement is followed by another. A selection statement is implemented by the *if* construct. A repetition statements is implemented by the *do* construct. The most important detail about the selection and repetition statements is that they both allow specifications of multiple sequences prefixed by boolean conditions called “guards.” At any point in time, more than a single guard may be executable which leads to the nondeterminism of Promela: out of multiple possible execution of sequences, SPIN nondeterministically (or, at the user’s option, deterministically) chooses one of them. The jump statement is implemented by the *goto* construct, which simply tells SPIN to jump to another labeled statement. Finally, the *unless* statement allows the user to approximately model exceptions as found in other programming languages.

Listing 1.1 shows a trivial Promela program which does nothing except to instantiate a single copy of each process p and q. The process p has a single *if* statement that is protected by 3 guards. Since the first two guards are simply the numbers 1 and 2, which are executable because they are not equal to 0, the process p is free to choose to nondeterministically execute between either one of these *guarded commands*. However, the third guard is interesting because it highlights an interesting feature of Promela: since the third guard is always a false boolean condition, it is never executable and hence its following *printf* statement is “blocked.” The executability of Promela statements means that it is easy to implement *busy-waiting*. Similarly, the process q has a single *do* statement that is protected by two guards which are, again, trivially executable. The difference between the process p and q is that p finishes execution after its single selection statement whereas q keeps executing its single repetition statement, in principle, forever because no termination conditions were specified.

Promela also allows for the definition of *channels* of data and *processes* which may communicate over these channels. A channel may be either be specified to be synchronous, which allows for no buffering of data and hence require blocking communication between the sender and the recipient, or asynchronous, which allows for buffering of data and hence not require blocking communication.

Promela is an inherently concurrent language which allows for the simultaneous, nondeterministic execution of processes. A sequence of statements may be specified to be executed atomically or deterministically. Linear temporal logic (LTL) expressions may also be supplied by the user as an easier alternative to *never claims* to express correctness properties which the model must satisfy. Promela has no notion of procedures or functions but a procedure may be simulated with *inline macros*, another concept borrowed from C.

A complete description of Promela is beyond the scope of this document. For a complete introduction to the language, we highly recommend the reader to see [3]. For more details on the formal theory and techniques of model checking, see [6].

Examples

The following listing presents in Promela the classic Dekker’s mutual exclusion algorithm for two processes alternating exclusive access to a shared resource[3][Chapters 3-5].

```
bool wantp = false, wantq = false;
byte turn = 1;
bool csp = false, csq = false;

active proctype p() {
    do
    ::  wantp = true;
        do
        :: !wantq -> break;
        :: else ->
            if
            :: (turn == 1)
            :: (turn == 2) ->
                wantp = false;
                (turn == 1);
                wantp = true
            fi
        od;
        csp = true;
        assert (!(csp && csq));
        csp = false;
        wantp = false;
        turn = 2
    od
}

active proctype q() {
    do
    ::  wantq = true;
        do
        :: !wantp -> break;
        :: else ->
            if
            :: (turn == 2)
            :: (turn == 1) ->
                wantq = false;
                (turn == 2);
                wantq = true
            fi
        od;
        csq = true;
        assert (!(csp && csq));
        csq = false;
        wantq = false;
        turn = 1
    od
}
```

Listing 1.2: Dekker's mutual exclusion algorithm in Promela

The following is the output from the SPIN verifier, *pan*, for this Promela program:

```
(Spin Version 5.1.7 -- 23 December 2008)
  + Partial Order Reduction

Full statespace search for:
  never claim           - (none specified)
  assertion violations  +
  cycle checks          - (disabled by -DSAFETY)
  invalid end states    +

State-vector 16 byte, depth reached 74, errors: 0
  172 states, stored
  153 states, matched
  325 transitions (= stored+matched)
    0 atomic steps
hash conflicts:         0 (resolved)

  2.501 memory usage (Mbyte)

unreached in proctype p
  line 25, state 23, "-end-"
  (1 of 23 states)
unreached in proctype q
  line 47, state 23, "-end-"
  (1 of 23 states)

pan: elapsed time 0 seconds
```

As the reader will notice, *pan* reports that both processes never stop computing, which is a good indicator that the specified assertions of correctness are never violated in any state of this system. We can rest assured that this formulation of Dekker's algorithm is safe and sound.

1.3 The Erigone model checker and Pomegranate compiler

Erigone is a partial reimplementation of the SPIN model checker. Its goal is to facilitate the learning of concurrency and model checking. It is a single, self-contained, executable file so that installation and use are trivial. It produces a detailed, customizable trace of the model checking algorithms and a uniform keyword-based format is used that can be directly read or used by other tools. Extensive modularization is used in the design of the Erigone program to facilitate understanding of the source code. This will also enable researchers to easily modify and extend the program.

The Erigone model checker was created by Professor Mordechai Ben-Ari of the Weizmann Institute of Science, Israel. We have spent the past year working in collaboration with Professor Ben-Ari to produce Pomegranate, a compiler that recognizes a subset of Promela that is sufficient for demonstrating the basic concepts of model checking for the verification of concurrent programs by the Erigone model checker. The Pomegranate compiler is now a modular component of the Erigone model checker which has been publicly released as a free software project[3].

No language constructs are added so that programs for Erigone can be used with SPIN when more expressiveness and better performance are desired.

Erigone and Pomegranate are written in Ada 2005 for reliability, maintainability and portability. This thesis documents the construction of Pomegranate, a 4-phase Promela compiler for the Erigone model checker, over the past year.

[3] http://code.google.com/p/erigone/

Chapter 2

Compiler Structure

2.1 Abstract

At its heart, the structure of the Pomegranate compiler is quite simple. Given a Promela source file, it is first tokenized into lexemes by the Lexer. This series of tokens is then parsed by the Parser according to certain grammatical rules, with which it builds an abstract syntax tree.[1] Next, this tree is passed to the semantic analyzer in the AST package which proceeds to decorate it with further details or check or even rearrange it for correctness.[2] Finally, this decorated tree is sent to the code generator in the CodeGen package which outputs, in a text file, a non-deterministic finite automaton which describes the evolution of processes.[3] This data flow in pictured in Figure 2.1.

The design is such that any model checker (for example, Erigone) is free to read as its input the automaton that was the output of the Pomegranate compiler. Assuming the correctness of the automaton, the model checker can perform a verification or simulation of it.

The interested reader may find the source code at Google Code[4].

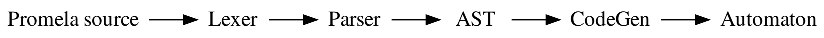

Figure 2.1: The data flow of the Pomegranate compiler

[1] These first two steps do not strictly describe what is happening; the parser actually reads the lexeme one by one, but these descriptive steps are easier concepts to understand. The technical details must not muddle the general picture.

[2] A prime example of rearranging the tree is for correcting binary operation precedence. This will be discussed in the chapter on semantic analysis.

[3] In detail, the automaton file includes a symbol table of numbers and strings. Each process has an initial state and a set of transitions from one state to another, each of which may involve the execution of byte code for a stack machine.

[4] http://code.google.com/p/erigone/

2.2 Major compiler components

In this section, we will explore interconnections, or how the Ada packages of the Pomegranate compiler depend on each other.

Firstly, there are the major packages that correspond to the 4 stages of the compiler: Lexer (lexical analysis), Parser (recursive descent parser), AST (semantic analysis) and CodeGen (code generator).

Each of these packages make use of the Logger package, which writes all manners of interesting technical information to a log file for the user who wishes to read them.[5] Since the logger is not described elsewhere, we will proceed to briefly describe it. The simple package is approximately modelled on the logging "module" of Python 2.5.[6] It has 5 'levels' of output in order of increasing importance : 'Debug', 'Info', 'Warning', 'Error', 'Critical'. The notion of a level is used to control the threshold of output; the user could specify, for the example, that only messages as important as the 'Info' level or above should be recorded. Besides choosing the name of a log file, the user is also free to switch off logging altogether.

In addition, each of the major packages also depends on a corresponding child package of the parent Types package. It used to be the case earlier in the development of the compiler that data structures and functional code (that is, the actual code of interest) were present together in large, monolithic packages. It was decided that such a design is not conducive for further development and maintenance of the program and as such, a considerable amount of effort was spent on refactoring the code so that essential and common data structures were separated from the code of interest. Accordingly, the child Types packages are Types.Lexer, Types.Parser, Types.AST and Types.CodeGen. The ways in which the major packages depend on them are picture in Figure 2.2.

Each edge from A to B in Figure 2.2 means that package A depends on package B for some functionality. The parent Types package simply depends on the standard Ada.Containers[2][Chapter 22] package. It can be seen that while some child types packages are used exclusively by a major package, others are used by most, if not all, of the major packages. An example of the former is Types.Parser and examples of the latter are Types.Lexer, Types.AST and Types.CodeGen. Types.Lexer is the package on which most others depend because it hosts the number and string tables. Types.Parser is chiefly used to decode a token type to a printable string, for the purpose of collating the text of a Promela statement so that it can be recorded as a component of a process transition.[7] Types.AST is the most complicated of them all; it is the host of all the abstract syntax tree nodes corresponding to Promela constructs. Finally, Types.CodeGen contains data structures (such as the instruction set for an idealized stack machine, the structure of the symbol table and a list of byte code) that are crucial to the production of an automaton output.

A relatively late addition in the development of the compiler was the ability to compile Linear Temporal Logic (LTL) expressions[3][Chapter 5] on the fly. An LTL expression as a whole is translated into a Büchi automaton[6][Chapter 6] by the Erigone model checker. The Büchi automaton need not concern us. What matters is that the Pomegranate compiler is used to translate the Promela

[5] In our case, there are 4 log files which are invaluable for debugging purposes.

[6] http://www.python.org/doc/2.5.2/lib/module-logging.html

[7] Once more, this is for the convenience of debugging.

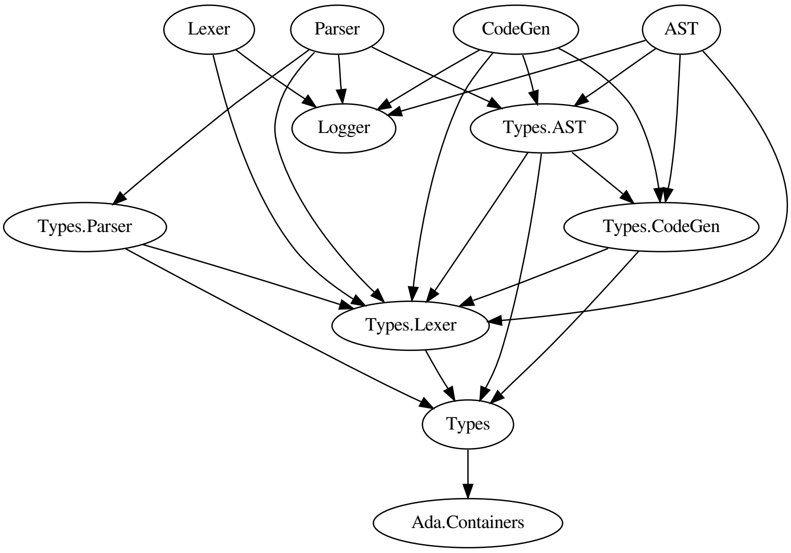

Figure 2.2: The internal package dependencies of the Pomegranate compiler

statements in LTL expressions into the appropriate sequence of byte code. After some discussion, we observed that it must be the case that LTL expressions may appropriately refer only to the global variables of a Promela program. (Following the convention set by the original SPIN model checker, Erigone accepts an LTL expression as an additional file to be evaluated during or after the compilation of a specified Promela program.) Most importantly, because program scopes are preserved between the semantic analyzer and the code generator, Erigone is able to repeatedly call the Pomegranate compiler to translate at will the Promela statements within LTL expressions into byte code.

It must be said that, following the principle of modularization, the only interface between the compiler and the model checker is a text file which describes an automaton.

2.3 Notes

The entire development of the compiler was done with AdaCore's GNAT GPL 2008 (20080521) compiler suite.[8]

[8] https://libre.adacore.com/

Chapter 3

Lexical Analysis

3.1 Abstract

Given a valid Promela source file as an input, the lexical analyzer, or "lexer", returns a series of tokens for each lexical construct. More precisely, each call to the lexer's Scan procedure sets information about the current lexical construct in a global object called Current_Token of type Token in the lexer package.

The lexer is essentially a 20-state[1] deterministic finite automaton[1] (DFA). The DFA for the lexer may be found in Appendix A while the equivalent grammar for the lexer may be found in Appendix B.

The parser uses the lexer with essentially a loop, in which each iteration calls the Scan procedure, after which we read values off the static Current_Token object. We terminate the loop when we see that:

Lexer.Current_Token.Kind = Lexer.Token_EOF.

The list of reserved Promela keywords were obtained by a careful reading of the manual pages in the SPIN book[6]. It is available in Section 3.2.2.

3.2 The Lexer Package

3.2.1 Public features

Files

The lexer package consists of the following files:

- lexer.ads: The lexical analyzer package specification.
- lexer.adb: The lexical analyzer package body.
- scanner.adb: An example program that makes use of the lexer package.

Exceptions

The lexer may raise the Generic_Lexical_Error exception in one of the following cases:

[1] Technically, it is a 21-state DFA, but the final state does not need to be remembered.

- When the lexer seen a character it does not recognize. (See the lexer grammar in Appendix B for more details.)
- When the lexer sees the character '#', on the suspicion that the source file has not been transformed with a C preprocessor.
- When the lexer sees a misformed (i.e. missing closing quote) character literal.
- When the lexer cannot open the specified source file.
- When the lexer does not expect to see the end of file (EOF).
- When any other unforeseen exception occurs at runtime. (A notable example being the case of a lexeme that is larger than the upper bound constant in the compiler for a bound string.)

In all of the cases above, the lexer will raise the exception with an informative error message.

Procedures

The lexer exposes the following procedures:

- Set_Source_Filename(S : String): The parser must call this procedure with a valid source input filename in order to make proper use of the lexer.
- Scan: The parser repeatedly calls this procedure in order to receive a series of tokens until the EOF token is seen. After each call, the parser inspects the Lexer.Current_Token object. The DFA is implemented as a loop of the form:

```
loop
   case State is
      when State_X =>
         Read_Character;
         case Peek is
            when 'A' =>
               Action;
               State := State_Z;
         end case;
      when State_Z =>
         exit;
   end case;
end loop;
```

- Fill_Buffer_With_String(S : String): This procedure is used either before or after lexical analysis of a source input file for the purpose of deliberately injecting a string into the lexer. For instance, it is used to compile Promela expressions in LTL formulae.

- Reset: This procedure is automatically called at the end of every call to the Scan procedure to assign default values to the Lexer.Current_Token object.

Communications with the parser

The lexer exposes the following variables:

- Current_Token: A global variable, of the type Token, used by the Scan procedure to hold data about the most recently seen token.
- Token:

```
type Token is record
   Kind           : Token_Type := Token_Null;
   Lexeme_Cursor  : Bounded_String_Hashed_Set.Cursor;
   Value_Cursor   : Natural_Hashed_Set.Cursor;
   Line_Number    : Positive;
end record;
```

Early in the implementation of the lexer, it was seen that there were two approaches to the problem of communicating about a token with the parser. On one hand, we could implement an object-oriented system with different types to convey the kind of a token but the implementation overhead would have been high for something as simple as a lexer. On the other hand, we could use a single covering type at the cost of maintaining all necessary variables in order for a global variable to be able to communicate about any possible token. It was decided that the latter approach is most suitable for our purposes, since the minimum number of variables needed for such a global variable turned out to be very small (in our case, 4).

3.2.2 Private features

Reserved Keywords & Special Names

Reserved keywords: *active, assert, atomic, bit, bool, break, byte, c_code, c_decl, c_expr, c_state, c_track, chan, d_proctype, d_step, do, else, empty, enabled, eval, false, fi, full, goto, hidden, if, init, int, len, local, mtype, nempty, never, nfull, notrace, np_, od, of, pc_value, pid, printf, printm, priority, proctype, provided, run, short, show, skip, timeout, trace, true, typedef, unless, unsigned, xr, xs* .

Special names: *_, _last, _nr_pr, _pid, accept, progress, STDIN* .

Procedures

The lexer includes the following internal procedures:

- Set_Reserved_Keywords: This procedure is used to initialize the names table with Promela's reserved keywords as Ada bounded strings.
- Set_Special_Names: This procedure is used to initialize the names table with Promela's global variables as Ada bounded strings.

- Read_Character: This procedure is used to read each character from the source input. Originally, there was no buffering and each call to this procedure involved a system call. The next iteration used the Gnat.OS_Lib package to implement buffering. The final iteration uses the standard Ada.Text_IO package for buffering.
- Cleanup: This procedure is used to clean up after used system resources at the end of lexical analysis.

Variables

There are many variables that are used within the lexer to keep track of the state of the automaton. These include the pointer to the source file, the current line number, the most recently observed character, the state number, the table of reserved keywords, a buffer to more efficiently parse the source file, the number of lines this buffer should store, the size of this buffer, a counter to keep track of how much the lexer has read from the buffer.

See the source code for more details.

3.3 The Types.Lexer Package

3.3.1 Overview

The Types.Lexer package hosts the data structures, procedures and functions needed to chiefly implement tables of names, numbers and strings.

3.3.2 Public features

Types

The Types.Lexer package has the following types:

- Token_Type: This is an enumerated type to identify the 'kind' of a token. Suffice it to say that there is a token type for each of the lexical construct as described by the grammar in Appendix B.
- Promela_Natural: This is a type of range $0..2^{31} - 1$.

Packages

The Types.Lexer package consists of the following packages:

- Bound_String: An instance of the generic package Ada.Strings.Bounded.Generic_Bounded_Length with its size parameter set to 32.
- Bounded_String_Hashed_Map: An instance of the generic package Ada.Containers.Hashed_Maps which maps from Bound_String.Bounded_String to Token_Type.
- Bounded_String_Hashed_Set: An instance of the generic package Ada.Containers.Hashed_Sets which stores unique elements of Bound_String.Bounded_String. It is used to implement the Name_Table and String_Table variables.

- Natural_Hashed_Set: An instance of the generic package Ada.Containers.Hashed_Sets which store unique elements of Promela_Natural. It is used to implement the Number_Table variable.

Functions

The Types.Lexer package contains the following functions:

- Bounded_String_Hash: This function is an instance of the Ada.Strings.Bounded_Hash function which is used to hash an element of the type Bounded_String.
- Natural_Hash: This function is used to hash an element of the type Promela_Natural.
- Element_In_Table: There are 2 overloaded versions of this polymorphic function, one which accepts a cursor to an element in a Bounded_String_Hashed_Set and returns a Bound_String.Bounded_String and another which accepts a cursor to an element in a Natural_Hashed_Set and returns a Promela_Natural.
- Add_To_Name_Table: Accepts a lexeme in the form of Bound_String.Bounded_String and returns its cursor.
- Add_To_Natural_Table: Accepts a number in the form of Promela_Natural and returns its cursor.
- Add_To_String_Table: Accepts a string in the form of Bounded_String.Bounded_String and returns its cursor.

3.3.3 Private features

Variables

The Types.Lexer package contains 3 tables that respectively store names, numbers and strings.

3.4 Comments on the lexer

At the moment, the lexer recognizes strictly 8-bit ASCII characters. No attempt is made at recognizing Unicodes characters, unlike, say, Ada 2005 or Python 3.0. The reasons for this have been primarily simplicity and a lack of critical demand, but this may change in the future.

Another important note is that the C SPIN model checker, at the time of writing, is *case-sensitive* and we follow this convention. It means that even reserved keywords can be circumvented by a simple case switch. (For example, ‘for’ is distinct from ‘For’, as it is in the Python programming language.) It will be good to make the Promela language more rigorous and user-friendly by introducing case-insensitivity.

We also need a C-like preprocessor for processing the #define and #include macros, constructs which were borrowed directly from the C language. (One will observe that Promela is very heavily influenced by C.) Actually, Promela does not recognize the #define macro, but it does have an equivalent construct called ‘inline’, which can be very simply transformed to a #define macro. The lexer is currently able to recognize and discard comments that are are delimited by the complementary character sequences '/*' and '*/'. There is a script that has been written for the 'Bash' shell that works around the problem of macros by using the venerable 'awk' program to remove the carriage return character (the reason for which will soon become clear) and transforming 'inline' constructs into equivalent #define macros. The script then proceeds to call the GNU C preprocessor 'cpp' to process the #define and #include macros. However, the script has not been publicly released because it cannot be integrated with the Erigone model checker to produce a single executable which can be compiled and executed on any operating system that is supported by Ada. This is due to a deliberate design decision to contain the Erigone model checker as a standalone program. It has been suggested that a future preprocessor for Pomegranate may be accomplished with the GNAT preprocessor.

A few difficulties were encountered during or after the construction of the lexer. An early iteration of the lexer involved making a system call to read each character off the input source file. A buffering scheme was introduced to reduce the number of system calls issued by the lexer. The first buffering scheme used the GNAT.OS_Lib package to read a fixed number of characters in each loop. However, the issue of depending on non-standard Ada 2005 packages was raised. We then had to switch to the current scheme of using "variable buffering", in the sense that we read a fixed number of lines instead but now there is no way to tell how many characters we would read in each loop. (Why did we not read a fixed number of characters again? This is because it would collapse to our first buffering scheme if done incorrectly and reading a fixed number of lines is a simple procedure.) The only problem with variable buffering is that the assumption that there are a reasonable number of characters per line may be challenged by a malicious user who would try to overwhelm the lexer by a deliberate crafting of an input source file. This is not as bad as it can be because the buffer is implemented with an Ada unbounded string; consequently, it will be a question of pushing system resources.

Another difficulty is that the Erigone model checker imposes an upper bound on the length of identifiers and strings. Since the Pomegranate compiler started its development by being tightly integrated with Erigone instead of being loosely coupled as it did become later, a similar upper bound was imposed on identifiers and strings scanned by the lexer too. This artificial constraint is at 32, but it has been recommended that the compiler increase this bound to 64 and adopt the Pascal method of using the acceptable prefix of an identifier rather than to completely abandon lexical analysis when a longer identifier is encountered.

Finally, there were 2 recent changes to the lexer that bear mentioning. Firstly, while UNIX End Of Line (EOL) characters were readily recognized and discarded, DOS EOL characters were not entirely recognized. The only character blocking recognition was the so-called "carriage return" character. The lexer is now capable of recognizing both UNIX and DOS EOL characters and should be able to do so for the Apple Mac operating system EOL characters as well. Secondly, the "on the fly" byte code compilation of Promela statements within linear temporal logic (LTL) formulae required that the buffer of the lexer be filled, as seen necessary, with a string that represents a Promela expression. The parser, semantic analyzer and code generator are called separately to handle the new input to the lexer.

Chapter 4

Parsing

4.1 Abstract

The recursive descent parser is responsible for building an abstract syntax tree from the sequence of tokens returned by the lexical analyzer on a given Promela program. At the same time, it also checks to see whether this Promela program is grammatically correct.

4.2 The official grammar specification

The official Promela grammar specification may be found either in the SPIN book[6][Chapter 16] or at the SPIN web site[1]. At the time of writing, the grammar specification on the web site is for the SPIN model checker from versions 4 to 5. The interested reader may find more explanations and examples of Promela in [3].

4.3 Adapting the grammar

Before the compiler work on the Erigone model checker was started, the original plan was to actually focus on implementing a subset of the Promela grammar. However, the author decided that lexically analyzing and parsing the entire grammar early in the development of the compiler will render it much more amenable to further development later on. Once the easier tasks of lexical analysis and parsing have been completed, we can devote our attention to the harder tasks of semantic analysis, code generation and code optimization. Indeed, this is precisely the trajectory that the compiler followed with a few minor exceptions (chiefly a couple of corrections to the parser that were about proper data accounting).

The project requirement dictated that the parser was to be recursive descent in nature. The first task at hand, then, was to study the official grammar specification to see how it can be adapted to fit a recursive descent parser.

Predictive, or recursive descent, parsers are known to be constructible for a class of grammars denoted as LL(1). The first 'L' means that we scan the input

[1] http://www.spinroot.com/spin/Man/grammar.html

from left to right whereas the second 'L' means what we produce a leftmost derivation from the grammar. The number '1' means that we use only one "lookahead" token of the input to make parsing decisions. The interesting note is that the class of LL(1) grammars is rich enough to describe most programming language constructs, although certain care must be taken in constructing such a grammar. For example, LL(1) grammars can be neither ambiguous nor left-recursive and two productions cannot have right-hand sides with the same prefix. Another interesting property of LL(1) grammars is that because we can build recursive descent parsers for them, we can also build equivalent nonrecursive predictive parsers that instead explicitly use stacks. This can be very useful in building general predictive parsers that need only their predictive parsing tables modified in order to parse different LL(1) grammars. See [1][Chapter 4] to learn more about context-free grammars and parsing techniques.

Our question now becomes: can a LL(1) predictive parser be built from the Promela grammar? The answer is yes, but with a few modifications to the grammar. First, we had to do two things: we had to *left factor* the grammar and eliminate *left recursions*.

Left-factoring is used to assist the predictive parser in choosing between similar production rules. (This is a different problem from ambiguity, which is defined to be the mapping of more than one parse tree to a given input.) The basic idea is that we factor away the longest common prefix between these similar production rules to generate new but equivalent production rules. It can be described succinctly by the following general transformation rule.

A production of the form. . .

$\langle A \rangle$ ::= $\alpha\ \beta_1$
 | $\alpha\ \beta_2$

. . . is equivalent to:

$\langle A \rangle$ ::= α $\langle A' \rangle$

$\langle A' \rangle$ ::= β_1
 | β_2

An algorithm is given in [1][Section 4.3.4] for left factoring a grammar. A simplified example of left factoring the Promela grammar follows. The original rules were:

⟨*stmt*⟩ ::= ⟨*send*⟩
 | ⟨*receive*⟩
 | ⟨*assign*⟩

⟨*send*⟩ ::= ⟨*varref*⟩ ! ⟨*send_args*⟩

⟨*receive*⟩ ::= ⟨*varref*⟩ ? ⟨*recv_args*⟩

⟨*assign*⟩ ::= ⟨*varref*⟩ = ⟨*any_expr*⟩

These were transformed to:

⟨*stmt*⟩ ::= ⟨*varref*⟩ ⟨*stmt_prime3*⟩

⟨*stmt_prime3*⟩ ::= ⟨*send_prime*⟩
| ⟨*receive_prime*⟩
| ⟨*assign_prime*⟩

⟨*send_prime*⟩ ::= ! ⟨*send_args*⟩

⟨*receive_prime*⟩ ::= ? ⟨*recv_args*⟩

⟨*assign_prime*⟩ ::= = ⟨*any_expr*⟩

Left recursions pose a serious problem to LL(1) predictive parsers precisely because they are recursive and derivations are made from left to right. Right recursions do not pose an immediate problem to LL(1) parsers, except for operator precedence[1][Section 2.4.5]. The problem is easy to see: the poor parser gets stuck in an infinite loop. We assist the parser by strategically removing *immediate* left recursion. It can be described succinctly by the following general transformation rule.

A production of the form…

⟨*A*⟩ ::= ⟨*A*⟩ α
| β

…is equivalent to:

⟨*A*⟩ ::= β ⟨*A'*⟩

⟨*A'*⟩ ::= α ⟨*A'*⟩
| ϵ

An algorithm is given in [1][Section 4.3.4] for eliminating left recursions from a grammar. A simplified example of eliminating left recursion in the Promela grammar follows. The original rules were:

⟨*any_expr*⟩ ::= ⟨*any_expr*⟩ ⟨*binarop*⟩ ⟨*any_expr*⟩
| ⟨*const*⟩

These were transformed to:

⟨*any_expr*⟩ ::= ⟨*const*⟩ ⟨*any_expr_prime3*⟩

⟨*any_expr_prime3*⟩ ::= ⟨*binarop*⟩ ⟨*any_expr*⟩
| ϵ

An important note is that we did not find any case of parsing ambiguity in the official Promela grammar specification.

4.3.1 The adapted grammar

We present an almost completely specified, mostly left-factored, left-recursion-eliminated grammar; see Appendix C. The author will soon explain why it is not completely specified and why it has not been completely left-factored.

4.3.2 Computing the FIRST/FOLLOW sets

It is necessary to compute the so-called FIRST and FOLLOW sets for an LL(1) grammar in order to build a predictive parser. This was achieved by a program that the author originally wrote to assist his coursework for Prof. Allan Gottlieb's course on "Compiler Construction" in Spring 2008 at the Courant Institute of Mathematical Sciences[2]. The program was written in the Python programming language[3]. Let us call it *ComputeSet*.

Briefly speaking, the FIRST set of a terminal or nonterminal is every terminal that starts the expansion of this terminal or nonterminal. The FOLLOW set of a nonterminal is every terminal that follows or appears at the end of this nonterminal, particularly if this nonterminal can produce an empty string or ϵ.

FIRST and FOLLOW sets, then, are powerful constructions that allow us to build a map, a predictive parsing table, with which we can then proceed to build a recursive descent parser. The parser is naturally recursive because we can parse a syntactic construct in terms of other syntactic constructs; for example, a binary expression may be defined to be a binary expression followed by a binary operator followed by a binary expression. (Clearly this is too simplified a picture; it does not allow for unary expressions and it also has no base case to terminate a recursion, but it suffices for explanatory purposes.)

Some FIRST and FOLLOW sets produced by the ComputeSet program were curated by hand. As we will soon see, this was largely due to the fact that the author did not sufficiently left-factor the Promela grammar.

The following FIRST and FOLLOW sets computation rules are adapted from [1][Section 4.4].

FIRST and FOLLOW sets computation rules

1. FIRST(a) = { a } for all terminals a.
2. Initialize FIRST(a) = φ for all nonterminals A.
3. If A → ϵ is a production, add ϵ to FIRST(A).
4. For each production A → $Y_1 \dots Y_n$,
   - add to FIRST(A) any terminal a satisying
     - a is in FIRST(Y_i) and
     - ϵ is in all previous FIRST(Y_j).
   - add ϵ to FIRST(A) if ϵ is in all FIRST(Y_j).

   Repeat this entire step until nothing is added.
5. FIRST of any string X = $X_1 X_2 \dots X_n$ is initialized to φ and then
   - add to FIRST(X) any non-ϵ symbol in FIRST(X_i) if ϵ is in all previous FIRST(X_j).
   - add ϵ to FIRST(X) if ϵ is in every FIRST(X_j). In particular if X is ϵ, FIRST(X) = $\{\epsilon\}$.

[2] A historical side note is that the author learned of this project at a guest lecture delivered by Prof. Schonberg in this course.

[3] http://code.google.com/p/erigone/source/browse/trunk/src/compile/ComputeSet.py

6. Initialize FOLLOW(S) = \$ and FOLLOW(A) = φ for all other nonterminals A, and then apply the following 3 rules until nothing is added to any FOLLOW set.

    - For every production A $\rightarrow \alpha B \beta$, add all of FIRST(β) except ϵ to FOLLOW(B).
    - For every production A $\rightarrow \alpha B$, add all of FOLLOW(A) to FOLLOW(B).
    - For every production A $\rightarrow \alpha B \beta$ where FIRST(β) contains ϵ, add all of FOLLOW(A) to FOLLOW(B).

Predictive parsing table computation rules

For each production A $\rightarrow \alpha$:

1. For each terminal a in FIRST(α), add A $\rightarrow \alpha$ to M[A, a].
2. If ϵ is in FIRST(α), then add A $\rightarrow \alpha$ to M[A, b] (resp. M[A, \$]) for each terminal b in FOLLOW(A) (if \$ is in FOLLOW(A)).

4.4 Implementation of the parser

We will now inspect the technique of transforming entries of the predictive parsing table for a particular nonterminal into a corresponding recursive descent parsing functions. We use functions instead of procedures because we wish to return an abstract syntax tree node for each nonterminal.

For the sake of simplicity, the following are *some* entries from the predictive parsing table for the nonterminal *stmt* (a Promela statement):

Each row in the table above means this: in the *stmt* production, if we see the terminal in the first column, then we are to take the production in the second column.

The *stmt* production is a good example because it is a straightforward LL(1) production. Nevertheless, in this example, we see two kinds of entries: non-conflicting and conflicting ones. The non-conflicting entries are indicated by a terminal key in in the predictive parsing table which has only one production rule as its value. The conflicting entries are indicated by a terminal in the table which unfortunately has multiple production rules as its value. The reason there are conflicting entries is because the author did not sufficiently left-factor this production.

It is clear that the terminals *empty, enabled, false, full, len', (, nempty, nfull, !, np_, number, ~, pc_value, run, skip, subtract, timeout* and *true* lead to only one production rule, which is simply *expr*. In this case, it is easy to see what we must do: simply expand the *expr* nonterminal. Ultimately, a nonterminal must consist of a terminal, in which case we "match" the terminal to see whether the next token in the stream from the lexer is equivalent to our expectation. If it is equivalent, we proceed to either call the next nonterminal function or match the next terminal. Otherwise we report a syntax error. As we match a terminal or call a nonterminal function, note that we are building an abstract syntax node from observed attributes.

Nonterminal	Production
EMPTY	expr
ENABLED	expr
FALSE	expr
FULL	expr
LEN	expr
LNBRACKET	expr
NAME	assign receive send labeled statement expr
NEMPTY	expr
NFULL	expr
NOT	expr
NP_	expr
NUMBER	expr
ONES_COMPLEMENT	expr
PC_VALUE	expr
RUN	expr
SKIP	expr
SUB	expr
TIMEOUT	expr
TRUE	expr

Table 4.1: Predictive parsing table entries for the *stmt* production

One can see from the *name* terminal that the entry consists of multiple production rules. This goes against our expectations of a proper LL(1) predictive parsing table. What should we do? Technically, it means that we have an ambiguity in choosing the right production rule in order to parse the given input. In this case, the 'name' nonterminal leads to 5 possible production rules. The reason why there are conflicting entries for this particular entry is because the author did not sufficiently left-factor the Promela grammar.

Instead of completely left-factoring the grammar at this stage of development of the parser, the author decided to temporarily work around the problem by looking ahead a number of tokens until a distinguishing token appears in the sequence to indicate the next production rule to be chosen. The number of look-ahead tokens for each production rule vary and the precise, distinguishing look-ahead tokens were determined by manual computation on certain nonterminal productions of the grammar. In practice, a small number of look-ahead tokens are needed to distinguish between conflicting production rules.

Finally, we arrive at the matter of handling syntax errors. Some common solutions to handling such errors are described in [1][Section 4.1.4]. As a temporary working solution, the author has chosen to simply abandon further parsing of the input (and consequently, any semantic analysis of and code generation from the input) when a syntax error is encountered. A descriptive error message is passed to the user in order to assist him in correcting the Promela source input. If this should not be sufficient, then the parser may also have written a log of the history of the parsing until the encounter with the syntax error. This log has proved itself invaluable in debugging the construction of the parser.

4.4.1 Problems

As we have seen in the previous section, the author's adapted Promela grammar turned out to be insufficiently left-factored. In summary, this meant that there were cases where 1 look-ahead token was insufficient in deciding on the choice of the next production rule. Examples of nonterminal productions that needed this rule were *stmt* and *any_expr*.

Another problem in the parsing of the Promela grammar occurred in the *expr* nonterminal production. The astute reader will have noticed that this production is rather conspicuously missing from our adapted Promela grammar. Why is this curiously the case? Let us first look at what the original production was.

⟨*expr*⟩ ::= ⟨*any_expr*⟩
  | (⟨*expr*⟩)
  | ⟨*expr*⟩ ⟨*andor*⟩ ⟨*expr*⟩
  | ⟨*chanpoll*⟩ (⟨*varref*⟩)

Let us also inspect the entry in the predictive parsing table for the *expr* production.

Nonterminal	Production
EMPTY	chanpoll LNBRACKET varref RNBRACKET expr_prime
ENABLED	any_expr expr_prime
FALSE	any_expr expr_prime
FULL	chanpoll LNBRACKET varref RNBRACKET expr_prime
LEN	any_expr expr_prime
LNBRACKET	any_expr expr_prime
	LNBRACKET expr RNBRACKET expr_prime
NAME	any_expr expr_prime
NEMPTY	chanpoll LNBRACKET varref RNBRACKET expr_prime
NFULL	chanpoll LNBRACKET varref RNBRACKET expr_prime
NOT	any_expr expr_prime
NP_	any_expr expr_prime
NUMBER	any_expr expr_prime
ONES_COMPLEMENT	any_expr expr_prime
PC_VALUE	any_expr expr_prime
RUN	any_expr expr_prime
SKIP	any_expr expr_prime
SUB	any_expr expr_prime
TIMEOUT	any_expr expr_prime
TRUE	any_expr expr_prime

Table 4.2: Predictive parsing table entries for the *stmt* production

It can be seen that the entry for a single terminal, (, rendered the task of parsing this production more complicated than it was expected to be. Although ultimately it might not have been generally impossible to choose between the two production rules for the entry, it turned out to be quite difficult in practice at the time. The author decided instead that it was better to “flatten”, “collapse” or “roll” the *expr* into the *any_expr* production. The only foreseeable problem with this approach is that the naturally higher precedence attached to channel polling operations might be destroyed by this modification of the grammar. In practice, it has not yet proven to be a problem. In any case, it can in principle be corrected during semantic analysis, a technique which we will inspect next.

It will not have escaped the notice of the careful reader that neither the original nor adapted grammar takes into account the hierarchy of operator precedence rules in binary expressions. Indeed, does the compiler take care of this problem? The anxious reader may now rest assured that the compiler responsibly handles the matter. The problem with the original and adapted grammars is that they both attach a “flat” hierarchy of operator precedence which increases in depth or, equivalently, priority from left to right until the end of the binary expression. While this may be correct for some expressions, it is certain to be wrong for almost all expressions. We will see in the next chapter that while the parser does nothing to correct this error, the semantic analyzer will do so by recursion.

There is also a simple mechanism in the *Stmt* procedure to collect tokens as text so that we may attach it to the automaton; this is useful for debugging purposes.

4.5 Comments on the grammar

The official Promela grammar specification is missing all the details necessary to build a complete parser. Consider, for example, the definition for the *typename* production from the official specification:

⟨*typename*⟩ ::= bit
  | bool
  | byte
  | short
  | int
  | mtype
  | chan
  | ⟨*uname*⟩

Unfortunately, it missing the following crucial detail, which is to be found in the manual page for this construct:

⟨*typename*⟩ ::= unsigned ⟨*name*⟩ : ⟨*constant*⟩

It has been suggested that the author could nevertheless provide a complete specification of an adapted Promela grammar. The author plans to modify in the near future the version of the Promela grammar presented here in order to make it completely specified and left-factored. The parser will then see the necessary modifications in order to reflect the changes to the grammar.

Chapter 5

Semantic Analysis

5.1 Abstract

The semantic analyzer of the Pomegranate compiler is the part of the program which is responsible for checking the correctness of the abstract syntax tree that was built by the recursive descent parser from the series of tokens returned by the lexical analyzer on a given Promela source input. The grammar defines a superset of the language and the semantic analyzer diagnoses errors that are not captured by the syntax. It also decorates this tree with more useful information for the convenience of the code generator which will later operate on this inspected and decorated tree. Finally, the tree may also be modified in order to provide semantic correctness that the grammar does not.

5.2 Abstract Syntax Tree

The abstract syntax tree (AST) is specified in the most complicated data structures package (Types.AST) of the Pomegranate compiler. The reason is naturally due to the fact that every nonterminal in the Promela grammar requires an AST node. It was a fairly straightforward and mechanical procedure to build for each nonterminal its corresponding AST node: each node would contain attributes that either point to other AST nodes or are terminal values. The complete hierarchy of the AST node type hierarchy is given in Appendix D.

Given these productions. . .

⟨*spec*⟩ ::= ⟨*module*⟩ ⟨*spec_prime*⟩

⟨*spec_prime*⟩ ::= ⟨*module*⟩ ⟨*spec_prime*⟩
 | ; ⟨*module*⟩ ⟨*spec_prime*⟩
 | ϵ

⟨*module*⟩ ::= ⟨*proctype*⟩

⟨*proctype*⟩ ::= ⟨*proctype_prime1*⟩ ⟨*proctype_prime5*⟩ ⟨*name*⟩ (⟨*proctype_prime2*⟩) ⟨*proctype_prime3*⟩ ⟨*proctype_prime4*⟩ { ⟨*sequence*⟩ }

⟨*proctype_prime1*⟩ ::= ⟨*active*⟩
  | ϵ

⟨*proctype_prime5*⟩ ::= PROCTYPE
  | D_PROCTYPE

⟨*proctype_prime2*⟩ ::= ⟨*decl_lst*⟩
  | ϵ

⟨*proctype_prime3*⟩ ::= ⟨*priority*⟩
  | ϵ

⟨*proctype_prime4*⟩ ::= ⟨*enabler*⟩
  | ϵ

. . . let us see how we may transform this specification into AST nodes:

```
type Node is interface;

type Module_Node is abstract new Node with null record;

type Proctype_Module_Node is new Module_Node with record
   Proctype      : Proctype_Node_Ptr;
end record;

type Spec_Node is new Node with record
   Module_List   : Module_Node_List.List;
end record;

type Proctype_Node is new Node with record
   Active        : Active_Node_Ptr;
   Version       : Lexer.Token_Type;
   Name          : Lexer.Bounded_String_Hashed_Set.Cursor;
   Parameters    : Decl_Lst_Node_Ptr;
   Priority      : Priority_Node_Ptr;
   Enabler       : Enabler_Node_Ptr;
   Sequence      : Sequence_Node_Ptr;
end record;
```

As was mentioned earlier, an AST node either points to other nodes or contains terminal values. Some AST nodes may have additional attributes that are not directly inferred from the grammar; for example, a *One_Decl_Node* node keeps track of the size, in bits, of the type of a variable declaration. As the reader who is familiar with Ada will note, AST nodes are implemented as *access types*[2][Chapter 10] to records[2][Chapter 8]. This simply means that the AST constructed by the parser is essentially a tree of dynamically allocated records or nodes. The cost of dynamically allocating memory to create this tree of records is, of course, manual management of memory on the part of the programmer. However, the cost of memory allocation for Promela machines, which are typically small in size, is negligible and as such, memory deallocation is left to the operating system at the end of runtime. *Node* is the parent or root of all AST nodes.

It is important to note that the author has chosen to instantiate a number of definite Ada containers in order to hold, for example, lists of variables and modules. (For example, see the *Spec_Node* node above.) While this approach offers programming simplicity and compile-time safety, it does come with the price of additional run-time executable size. In order to reduce this size, it has been suggested that these definite containers should be replaced with a single indefinite container type. However, while this will alleviate the problem of executable size, it will come at the price of programming complexity and less compile-time safety.

Most of the Types.AST package is dedicated to defining the AST nodes in terms of Ada 2005 access types. Besides the AST nodes, there are other data structures in the package that do not directly correspond to the grammar. An example is the structures that define Promela data types. These include built-in/native/primitive, array, channel and user-defined data types. Each data type has at least a size specified in units of bits. The primitive Promela data types are given by:

Name	Size in bits	Minimum	Maximum
bit/bool	1	0	1
byte/mtype	8	0	$2^8 - 1$
short	16	-2^{15}	$2^{15} - 1$
pid/unsigned	31	0	$2^{31} - 1$
int	32	-2^{31}	$2^{31} - 1$

Table 5.1: Promela primitive data types

Actually, a variable of the *unsigned* data type is supposed to be flexible in size up to 31 bits. However, thanks to the incomplete official specification of the Promela grammar, the Pomegranate compiler makes no effort to support this data type. An mtype variable is technically no different from a byte variable but we distinguish between them at compile-time in order to build a table of mtype constants. A similar case applies to the pid and unsigned data types. There are no floating point numbers in Promela. Note also that we distinguish between primitive, user-defined and channel data types.

Mtype or enumerated type constants are collected during semantic analysis to produce a complete compile-time table of named byte constants. This could be thought of as very simple 'constant folding.' An mtype constant is naturally constrained between 0 and 255 because the SPIN model checker stores it as an unsigned character data type. It has not been thoroughly investigated whether the SPIN model checker uses a particular convention to map a named mtype constant to a value as a byte. In any case, it does not matter. The Pomegranate compiler uses the simple convention of mapping named mype constants to increasing values in bytes based on the order of appearance. For example, if the compiler sees the mtype constants 'b', 'a' and 'c' in that order, then these constants would be respectively mapped to the values 0, 1 and 2. An informative exception will be raised in case the compiler observes duplicate or more than 255 mtype constants.

Finally, another important data structure specified in the Types.AST package is the symbol table. Traditionally, symbol tables are specified with linked lists[1][Section 2.7] but in our case they are specified with trees. Why is this so? Specifically, the symbol table is a tree of “scopes,” where each scope contains a hash table, a pointer to its parent scope and a list of children scopes. The hash table simply maps names to node pointers. It is easy to see that the outermost or global scope will necessarily contain as its children the scopes of many processes. At the moment, Pomegranate follows the SPIN convention of hosting a single scope for all of the variables in a process; as such, there is no nesting of scope inside of processes. Crucially, we will need to pass this tree to the code generator. We will look more closely at the symbol table in the next section.

5.3 Analysis

Most of the analysis in the semantic analyzer is encapsulated in procedures that are specific for each node in the abstract syntax tree. The remaining procedures and functions in the analyzer are used to manage symbol tables, correct binary tree expressions, assist type checking and report semantic errors.

5.3.1 Node analysis

We will step through interesting examples of semantic analysis on a few important nodes in the abstract syntax tree. A very common analytical check for most nodes is to ensure that certain pointers to other nodes are, in fact, not null or empty.

Mtype: This procedure is responsible for collecting an enumerated mtype constant, map it to the next available number in the byte range and put it in the global scope so that the compiler may find it later in a reference. It performs certain error checking, such as making sure that there are no two mtype constants with the same name and that there are no more than 255 mtype constants.

Utype: This procedure is responsible for making sense of user-defined types. Being rather lazy and clever, it uses the *Decl_Lst* procedure to work on its own Decl_Lst node in order to figure out what the features of its variables must be, including their total size in bits. A local scope is given for the variables in a user-defined type so that they may not clash with the current (global) scope. Even though all this work is done, the compiler makes no further effort to analyze user-defined types and as such, they remain an unsupported construct of Promela; the reasoning is that they are a relatively less-frequently used construct and as such they are rather low in priority on our development radar.

(Before we proceed to the next example, let us make a note. The *one_decl* production for a variable declaration has as its children the *typename* and *ivar* productions. This explains why a *typename* and an *ivar* node can be siblings.)

$\langle \textit{one\_decl} \rangle ::= \langle \textit{one\_decl\_prime1} \rangle \ \langle \textit{typename} \rangle \ \langle \textit{ivar} \rangle \ \langle \textit{one\_decl\_prime2} \rangle$

Typename: This procedure determines the type of a declared variable, whether it is a primitive, channel or user-defined data type. However, due to the syntax of Promela, it cannot know whether or not it is dealing with an array. This crucial piece of information is later passed on to or, equivalently, inherited by its sibling, the *Ivar* node.

Ivar: This procedure knows everything about a declared variable, save for its type. This is not a problem since we can easily inherit this information from a sibling *Typename* node. Also, due to the grammar, only this node can determine whether or not a variable is an array. Common checks include ensuring that there is not already a variable with the same name in either this scope or the global one. If this variable is an array, then the array size must be a positive integer *and* must not exceed the known size of the memory of the Erigone model checker; its size in bits will be the number of elements multiplied by the size of each element in *bytes* multiplied by 8 bits. The size of a scalar variable will simply be the size of its primitive data type. If the variable has an initialization expression, then it will also be analyzed.

Stmt: This procedure mostly calls other procedures to analyze each kind of Promela statement. However, it does take care of the *printm* statement, which is used to print an mtype constant, on its own. It tries to find the declaration of the mtype constant in question; otherwise, it reports as an error that the constant is missing. This declaration will be useful for code generation later.

Assign: This procedure is responsible for analyzing assignment expressions. Unfortunately, there is no compile-time type checking as yet to ensure, for example, that the target of an assignment is large enough to handle the result of its right hand expression. We will see later why this is the case in a section on type checking. However, it does forbid any assignment to an mtype constant.

Ch_Init: Similar to the *Utype* node, nothing terribly useful is currently done save for the analysis of component typenames (or channel fields) and the total size of the channel. This should be useful to the future development of the compiler when channel operations will be introduced.

Proctype: This procedure, of course, defines a process. Since the Promela grammar is modular, most analysis is performed by calling other procedures. For example, the *Active* procedure ensures that no more than 255 instances of a process may be allowed. (While this may seem an arbitrary limitation, it must be remembered that the SPIN model checker is designed to simulate and verify *finite* state machines.) A most important observation is the nature of a scope in a Promela process: *it is actually valid Promela syntax to be able to refer to a variable that is only declared later in the process.* Needless to say, this is a somewhat unusual choice in programming language design. A solution to this problem would be to parse a given Promela program twice: once, to make careful note of all the variables in the program and once more to proceed to semantic analysis or code generation once this map of variables has been obtained. *We have not enforced this dangerous habit of referring to variables before their declarations.* In fact, the Pomegranate compiler adheres to the principle of simplicity and will report a semantic error should a variable be referred to before its declaration. Nevertheless, the semantic analyzer uses a *single* scope to host all of the variables in a process, which is in keeping with the original Promela semantics.

Any_Expr: Similar to the *Stmt* node, this procedure works mostly by calling other modular procedures. One of its most important tasks is to issue a series of calls until a binary expression (or a unary expression with a binary expression) is properly constructed in accordance with the operator precedence rules of Promela. This binary operator precedence correction is explained in more detail in Section 5.3.3. There are also some inactivated leftover experimental code that tried to introduce additional compile-time type checking to Promela; these will be useful to future development of the Pomegranate compiler.

5.3.2 Symbol tables

The semantic analyzer stores the scopes, global and local, of a Promela program as a tree so that it may be passed without modification to the code generator. The code generator will have to do no further processing of the program in order to find out where variables are. It will traverse the scope tree as necessary and delete a scope once code has been generated for it. Of course, any traversal of particular branches of the scope tree will yield a regular linked list of scopes. See Figure 5.1 for an example.

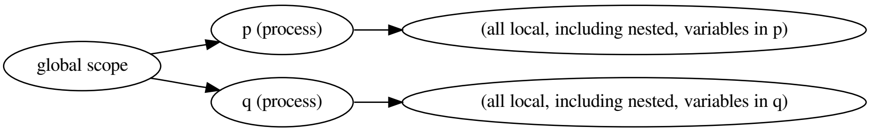
global scope
p (process)
(all local, including nested, variables in p)
q (process)
(all local, including nested, variables in q)

Figure 5.1: An example of a Promela scope tree

There are a few functions and procedures that manage the symbol table or scope tree in the semantic analyzer:

Scope_Put: This function associates a node of choice with a particular name in the current scope. It returns 'True' when an association has been made for the first time and 'False' otherwise.

Scope_Get: This function tries to find a node associated with a particular name in the current scope. If it is not found in this scope, it tries to do so for every scope above the current scope until it reaches the global scope. It will return either an associated node if it is found or 'null' otherwise.

Is_There_Global: This function works similarly to the *Scope_Get* function, but it returns 'True' if an association is found in the global scope and 'False' otherwise.

Scope_Push: This procedure creates a new scope as the next child of the current parent scope, sets the parent of the child scope and replaces the current parent scope with its child.

Scope_Pop: This procedure replaces the current scope with its parent scope.

Get_Root_Scope: This function returns the current scope of the abstract syntax tree, which will be the root of the scope tree if the semantic analysis was done properly. It is only used by the code generator to obtain the root of the scope tree.

5.3.3 Binary expression operator precedence correction

The problem with the original and adapted Promela grammar is that binary expressions do not reflect operator precedence. The original and adapted Promela grammar lead to purely right-associative binary expressions. For example, the binary expression $(1+2*3/4-5*6)$ would be parsed as $(1+(2*(3/(4-(5*6)))))$ instead of $(1+((2*3)/4)-(5*6))$.

A comprehensive, for our purposes, operator precedence map is given by:

Operator	Priority
||	3
&&	4
|	5
^	6
&	7
==	8
!=	8
<	9
<=	9
>	9
>=	9
<<	10
>>	10
+	11
-	11
/	12
*	12
%	12

Table 5.2: Promela binary operator precedence map

The higher the priority, the higher the operator precedence. For a complete list, please see [3][pp. 7].

It was decided that we would solve this problem by using a simple recursive algorithm during semantic analysis. This recursive algorithm would correctly restructure the subtree for the binary expression to ensure that a proper traversal of the tree will amount to the right arithmetic evaluation of the expression.

The following is a pseudocode of the algorithm used to correct a binary expression in the AST:

```
if an expression has a binary operator then
  binary_expression := restructure( expression );
  tree_changed := False;
  loop
    correct( binary_expression , tree_changed );
    exit when tree_changed = False;
  end loop;
end if;
```

Binary_Tree_Restructure is simply a function that begins a process of creating a more compact tree for a binary expression. It recursively creates a similarly right-associative but smaller tree for a binary expression, by returning a Binary_Any_Expr_Node for a generic Any_Expr_Node. The smaller tree also has distinguishing attributes such as left/right leaves and expressions.

```
function restructure( expression ) return binary expression is begin

  expr := new binary expression;
  new_expression.operator    := expression.binaryop.operator;
  new_expression.left_leaf   := expression;
  if expression.binaryop.expression.binaryop is null then
    new_expression.right_leaf := expression.binaryop.expression;
  else
    new_expression.right_expression :=
      restructure( expression.binaryop.expression );
  end if;
  free( expression.binaryop );
  return new_expression;

end function;
```

5.3.4 Other analysis and error reporting

Besides the analysis described above, examples of other kinds of analysis include checking that other nodes referred to by a node are not empty when they should not be. For example, a node for a variable declaration must have non-empty typename and variable nodes. On the other hand, the node specifying the priority of a process may be empty within the process node because it is not a crucial property. These checks are there to ensure that the parser built a proper AST.

The semantic analyzer, akin to the lexical analyzer and parser before it and the code generator after it, features basic error handling and reporting. The *Semantic_Assert* procedure is used to assert that a certain boolean condition must hold true; in case of failure, a useful error message is presented to the user. The *Raise_Unsupported_Construct* procedure is used to report to the user that certain constructs of the Promela language, such as channels and user-defined types, are unsupported by the Pomegranate compiler at the present time.

5.3.5 Type Checking

Unfortunately, the original SPIN model checker features very little compile-time type checking, as we will soon see. Instead, it relies on runtime type checking. Even more unfortunately so, the Pomegranate compiler follows this questionable tradition. The issue of runtime type checking will be discussed in the next chapter when we examine the Pomegranate virtual machine byte code specification.

Actually, there exists in Pomegranate some experimental code for performing compile-time type checking. This code has been deliberately inactivated due to the shifting nature of the implementation, lack of time and solving other problems with more pressing priorities. For example, there are algorithms to compute the "least upper bound" of any two primitive data types so that we may decide whether or not an expression could fit the target of an assignment without a narrowing operation.

The problem with the SPIN model checker's runtime type checking is that it relies on *truncation*[6][citation needed]. Truncation works by the following logic: if the result of an expression does not fit the data range of a variable of a primitive data type, then the result will be truncated to fit the data range of the primitive data type.

For an example of the problem with truncation, let us look at the following Promela program.

```
bit   stop_nuclear_reactor   = 0;
byte  a_malicious_byte       = 255;

init
{
  stop_nuclear_reactor =
    stop_nuclear_reactor + a_malicious_byte;
  if
    :: stop_nuclear_reactor == 1  ->
      printf( "Truncation␣side␣effect␣confirmed!\n" );
    :: else                       ->
      printf( "Truncation␣side␣effect␣not␣confirmed.\n" );
  fi;
}
```

The output from SPIN 5.1.7 (23 December 2008) for this program is:

```
spin: line   6 "test/truncation.pml",
  Error: value (255->1 (1)) truncated in assignment
      Truncation side effect confirmed!
      1 process created
```

Clearly, truncation is not the best idea for a language such as Promela whose objective is the correct verification of software.

5.4 Comments

Let us look at a few areas of improvement for the semantic analyzer.

- It could use more verbose, more comprehensive error checking and reporting instead of silent failures; there is no code for handling certain edge cases which would mislead the user in tracing the fault in an incorrect Promela program. For example, there are procedures in the semantic analyzer that act upon a node assuming it is a certain type but fail to report an “unexpected type” error if this condition is not met.

- The recursive binary operator precedence correction algorithm could certainly use a proof of correctness.

- Another low-hanging fruit is the introduction of the compile-time type checking.

Chapter 6

Code Generation

6.1 Abstract

Code generation for Promela presents a few interesting problems because it is not a conventional programming language. Promela was designed, above all, for the description and validation of computer protocols. This renders the compilation of Promela programs an interesting proposition.

Before we proceed to our techniques, it is very important to state this crucial insight: *every Promela program can be transformed into an equivalent nondeterministic finite state machine.* This is precisely what drives the C SPIN compiler and the Pomegranate compiler: both of the compilers transform a Promela program into an equivalent nondeterministic finite automaton (NFA).

This insight is actually quite intuitive and easy to understand. The formal theory underlying this insight is described with a few enlightening examples in [5][Chapter 8] and [6][Chapter 6]. We will explain the insight by presenting to the reader the following example of a transformation.

```
bit data[10];

init
{
  byte counter, parity, answer;
  do
    ::  counter < 10  ->  parity = parity + data[ counter ];
                          counter++;
    ::  else          ->  break;
  od;
  answer = parity % 2;
  if
    ::  answer == 0   ->  printf( ”even” );
    ::  else          ->  printf( ”odd” );
  fi;
}
```

Listing 6.1: A parity computer in Promela

Listing 6.1 presents a Promela program that computes the even or odd parity of a bit array. Let us call it a parity computer.

Figure 6.1 is the nondeterministic finite automaton that represents the parity computer. The state numbers correspond to arbitrary numbers of Promela or *noop* statements and a transition from state X to state Y implies the execution of the statement with the number Y. The *halt* transition is used as a convention to signal to the Erigone model checker that this process has finished executing. (A *noop* statement is an empty statement and a *noop* transition executes nothing.)

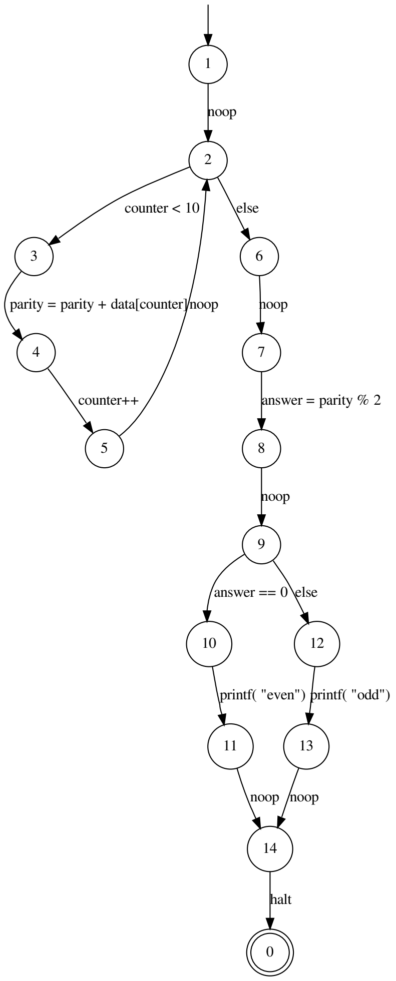

Figure 6.1: A naïve NFA for the parity computer

Figure 6.2 is the optimized nondeterministic finite automaton that represents the parity computer.

It will also be instructive to examine the output of the C SPIN compiler on this input:

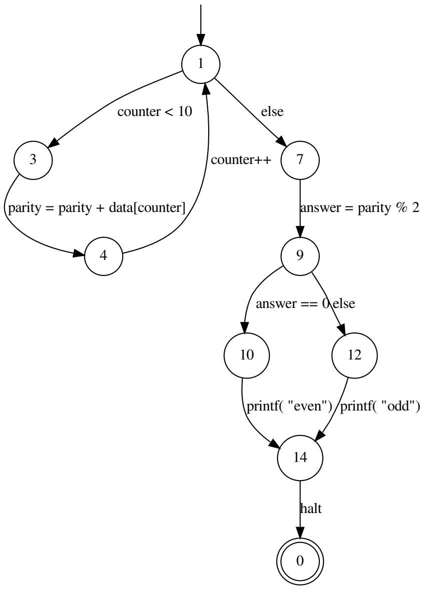

Figure 6.2: An optimized NFA for the parity computer

```
proctype init
  state   6 -(tr   3)-> state   2  [id   0 tp   2] [----L] line 6
    => ((counter<10))
  state   6 -(tr   2)-> state   9  [id   3 tp   2] [----L] line 6
    => else
  state   2 -(tr   4)-> state   3  [id   1 tp   2] [----G] line 7
    => parity = (parity+data[counter])
  state   3 -(tr   5)-> state   6  [id   2 tp   2] [----L] line 7
    => counter = (counter+1)
  state   9 -(tr   6)-> state  14  [id   8 tp   2] [----L] line 10
    => answer = (parity%2)
  state  14 -(tr   7)-> state  11  [id   9 tp   2] [----L] line 11
    => ((answer==0))
  state  14 -(tr   2)-> state  13  [id  11 tp   2] [----L] line 11
    => else
  state  11 -(tr   8)-> state  16  [id  10 tp   2] [----L] line 12
    => printf('even')
  state  16 -(tr  10)-> state   0  [id  15 tp 3500] [--e-L] line 15
    => -end-  [(257,9)]
  state  13 -(tr   9)-> state  16  [id  12 tp   2] [----L] line 13
    => printf('odd')
  state 6 line 6 is a loopstate

Transition Type: A=atomic; D=d_step; L=local; G=global
Source-State Labels: p=progress; e=end; a=accept;
```

The reader can see from the preceeding example that Promela presents fairly new and interesting puzzles for the compiler writer to solve. The problem of code generation for a Promela program can then be reduced into two major subproblems: construction of the nondeterministic finite automaton[1][Section 3.6] and byte code generation for Promela statements. These subproblems, while not independent from each other, can be considered separately.

6.2 Virtual machine specification

Let us focus our attention on the subproblem of byte code generation for Promela statements. One of the design requirements for the Erigone model checker is that the target computer for the Pomegranate compiler must be a stack-based machine.

Our first task at hand then is to define a virtual machine. The Pomegranate Virtual Machine (PVM) makes a few assumptions about the virtual machine and borrows heavily from the Java Virtual Machine (JVM) specification as an inspiration for its own instruction set[7]. The reason the JVM was chosen as the main source of inspiration is because it is a modern and efficient example of a stack-based virtual machine. We also tested basic byte code generation procedures by checking the correctness of the output with a Java assembler[1].

The specification defines the primitive data types of the virtual machine, runtime data areas (for example, a runtime constant pool that stores numbers and strings from the Promela program), frames (temporary working areas for process instances), the instruction set and some conventions for determining the executability of a statement (this will be explained in a later section).

The instruction set of the PVM is divided into these categories:

- Arithmetic: addition, multiplication and the rest of the usual binary operation set.
- Memory manipulation: loading from and storing into the random access memory and operand stack.
- Input and output: printing information to the console.
- Bitwise and logical operations including boolean conditions: logical (and, or, et cetera) and comparison (less than, greater than, et cetera) operators.
- Miscellaneous instructions including instructions to emulate built-in functions: assert, halt, noop (no operation) and so on.

We plan to introduce in the near future instructions to support channel operations.

The PVM and its instruction set is specified in Appendix E.

6.3 Preprocessing the Promela program

6.3.1 Symbol table

Before the code generator can produce the nondeterministic finite automaton for a Promela program, it first processes the program to form its symbol table,

[1] http://jasmin.sourceforge.net/

which determines directly the runtime memory layout of the program. This means that the code generator is a 2-pass compiler: the first pass is used to write the symbol table and the second pass is used to write the nondeterministic finite automaton. It is important that we generate the symbol table first so that we may later generate byte code instructions for loading from and storing into variables after we know how they are mapped in the random access memory of the Erigone model checker.

```
bit a;
bool b    = 1;
byte c    = 127;
byte d[5] = 42;
active proctype p() { byte e; }
```

Listing 6.2: A few variables in Promela

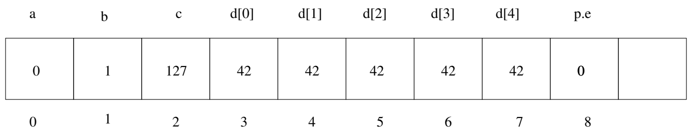

Figure 6.3: A state vector

The Erigone model checker uses a single state vector to store the state of all of the variables, including global and local ones, in a Promela program. The smallest unit of this state vector is a Promela byte. This means that a variable with a primitive data type such as the Promela bit or bool type will use a byte of storage in this state vector. It was observed that efficiently packing storage space in terms of units of bits is a difficult problem with negligible positive results. See Figure 6.3.

At the moment, the state vector of Erigone is bounded at 256 bytes, although the user can change this upper bound by modifying the corresponding compile-time constant. The number of possible state vectors is then $2^{256 \times 8}$, although a typical program will only visit a small subset of these states. Certain algorithms[6][Chapter 8] and optimizations[6][Chapter 9] are crucial for the feasibility of model checking.

Given a variable then, we use the following procedures to write its entry in the symbol table:

- Write_Symbol: This procedure writes in the automaton output file the symbol table entry for a variable.

- Store_Ivar: This procedure generates byte code instructions to store the result of a Promela expression into a variable, whether it is of a scalar or an array primitive data type. There is no difference between the storage instructions for variables of a scalar or an array primitive data type: the convention is that it is the duty of the Erigone model checker to properly initialize a scalar or an array.

- Ivar: This procedure calls the *Write_Symbol* and *Store_Ivar* procedures to write the proper symbol table entry for a variable. First, we ensure that the variable in question is indeed in this scope. Next, if the variable is of a primitive scalar or array data type, then we compute its size and consequently its index in the global state vector. If the size of the variable cannot fit the memory of the Erigone model checker, then we report an error. We also make note of other information such as whether the variable is a parameter in a process instantiation. The key to a global variable in the state vector is the simple name of the variable *variable_name* whereas the key to a local variable in the state vector is the qualified name *process_name.variable_name*. Next, we generate the byte code instructions to initialize the variable; by default, a Promela variable of a primitive data type is initialized to 0. Finally, we write to the automaton output an entry in the symbol table for this variable. (Note that there is no corresponding *Typename* procedure because we have already obtained the type information of a variable during semantic analysis.)

The exact specification of a symbol table entry is important only to the Erigone model checker which must be able to read in the symbol table. The following is a trivial Promela program that is mostly used to declare a few variables:

```
bool rejected = false;

active proctype FA()
{
  byte h;
  byte i[5];
  i[0] = 'a';
  i[2] = 'b';
  i[4] = '.';
  printf("Accepted!\n");
}
```

The symbol table that results from the Promela program as listed above is as follows:

```
number=,offset=0,value=0,
symbol=,type=bit_type,name=rejected,offset=0,length=1,size=1,
  scope=global_scope, byte code={iconst 0 0,bit_store 0 0,},
symbol=,type=byte_type,name=FA.h,offset=1,length=1,size=1,
  scope=local_scope, byte code={iconst 0 0,byte_store 1 0,},
symbol=,type=array_type,element_type=byte_type,name=FA.i,
  offset=2,length=5,size=1,scope=local_scope,
  byte code={iconst 0 0,byte_store 2 0,},
number=,offset=1,value=97,
number=,offset=3,value=98,
number=,offset=4,value=2,
number=,offset=6,value=46,
number=,offset=7,value=4,
string=,offset=0,value="Accepted!\n",
```

6.4 Generating code for the Promela program

After generating the symbol table for a Promela program, we know how the global and local variables of the program are mapped in the model checker memory. Hence, we can turn our attention to the subproblem of generating the nondeterministic finite automaton of the program.

The problem of generating the NFA of the Promela program can be further reduced into two minor subproblems: the generation of correct byte code instructions for a Promela statement and the generation of correct state transitions between Promela statements. We will inspect the details of each of these two subproblems of NFA generation in the following sections.

The resulting nondeterministic finite automaton is written to a text file that is also known as an *automaton object* file. As a historical sidenote, the Pomegranate compiler used to belong in the runtime of the Erigone model checker. It was later decided that separating the runtimes of the compiler and the model checker will be good for the modularization of the overall software. However, there was considerable debate as to the precise nature of the modularization. One line of argument was that the compiler and the model checker should be completely independent of each other, to the point of sharing no code whatsoever, even at the cost of redundant code. The argument was that this is in line with the traditional philosophy of UNIX tools. The opposing vein of argument was that it might be harmful to the development of the overall software should the compiler and model checker share no code whatsoever. In the end, it was agreed that the compiler and the model checker should at least share the instruction set of the PVM: this way, the compiler and the model checker do not run the risk of "speaking" different languages.

6.4.1 Generation of byte code instructions for a Promela statement

The task of generating byte code instructions for most Promela statements is quite simple. We are following standard techniques[1][Chapters 5-6, 8] for generating byte code instructions for a stack-based machine.

For example, the generation of byte code instructions for Promela arithmetic expressions is trivial. For any in-order Promela binary arithmetic expression $expr_1 \ \phi \ expr_2$, we would first generate byte code instructions for the expression $expr_1$, followed by the expression $expr_2$ and finally for the operator ϕ. Intuitively, this means that we first evaluate $expr_1$ and push the answer onto the stack. We next evaluate $expr_2$ and push the answer onto the stack. Finally, with both operands on the top of the stack, we evaluate the binary operator by popping both of the operands off the stack, applying the binary operation on these operands and pushing the final answer onto the top of the stack.

As another example, there are shared routines for generating instructions for storing into and loading from variables of primitive types. These procedures mostly involve the computation of the appropriate instruction together with the index into the variable in the state vector memory of the Erigone model checker. Arrays of primitive data types are also supported and the procedures for loading from and storing into them mostly involve the computation of the appropriate load or store instruction, an instruction to check at runtime the index into the array, the index into the array and the size, in bytes, of each element of the array. Curiously, Promela also allows for scalar variables to be treated as 1-element arrays; we do generate the proper code to handle these edge cases.

Other examples of implemented Promela statements include the *printf* statement (which is similar if not identical to its counterpart in the C language), the *printm* statement (which allows one to print an enumerated constant) and the *atomic, assert, break, do, else, goto, if*, labeled and assignment statements.

At the time of writing, we have not yet implemented the compilation of channel operations, which can be fairly said to consist of most of the Promela-specific expressions. We plan to do so in the near future.

6.4.2 Generation of transitions between Promela statements

After generating the byte code instructions for a Promela statement, we must now form a transition from this statement to others.

Transitions in a Promela sequence of statements

The following pseudocode explain the formation of transitions between most Promela statements, with the exceptions of *if, do, atomic, d_step, goto*, nested and labeled statements, within a given sequence.

This is the pseudocode for the *Sequence* procedure. For a given Sequence of Statements:

1. Prepend and append the Sequence with *noop* or no-operation statements if necessary. (See the *Prepend_And_Append_Sequence_With_Noop* procedure.)

2. For every Statement in a Sequence:

   (a) Assign a unique-per-process state number to this Statement.

   (b) Generate byte code instructions for this Statement.

   (c) If the *previous* Statement is not a *if, do, atomic, d_step, goto*, nested or a labeled statement, then form a transition from the *previous* Statement to *this* Statement.

3. Form the transitions for the *if, do, atomic, d_step, goto*, nested or labeled statements within this Sequence.

The following is the pseudocode for the *Prepend_And_Append_Sequence_With_Noop* procedure. This procedure is used to guarantee that certain statements, notably the *if, do, atomic, d_step, goto*, nested or labeled statements, are ultimately connected by transitions (see Section 6.5.1). For a given Sequence of Statements:

1. If this Sequence is not an optional sequence of statements within an *if* or a *do* statement, then:

   (a) Assume that all declarations of variables are at the top of this Sequence[2].

[2] Regrettably, this has not been enforced by the semantic analyzer due to a desire for backwards compatibility with the C SPIN model checker.

(b) Find the first Statement Step in this Sequence that is not a variable declaration.

(c) Insert a *noop* statement *before* this Statement Step.

2. If the *last* Statement in this Sequence is a *if, do, atomic, d_step, goto*, nested or a labeled statement, then insert a *noop* statement *after* the last Statement.

Transitions for IF statements

The process for forming transitions for an *if* statement is pictured in Figure 6.4.

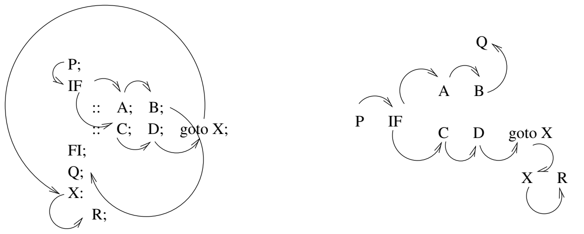

Figure 6.4: The processing of Promela IF statements

1. Form a transition between the first statement before the *if* statement and the *if* statement itself. (P to IF).

2. For every guarded command statement within the *if* statement, form a transition between the *if* statement and the guarded command statement. (IF to A, IF to C.)

3. For every exiting statement that follows a guarded command statement:

   (a) If it is a *goto* statement, then form a transition between the *goto* statement and the corresponding labeled statement. (GOTO to X.)

   (b) Otherwise, form a transition between the exiting statement and the first statement after the *if* statement. (B to Q.)

Transitions for DO statements

The process for forming transitions for an *do* statement is pictured in Figure 6.5.

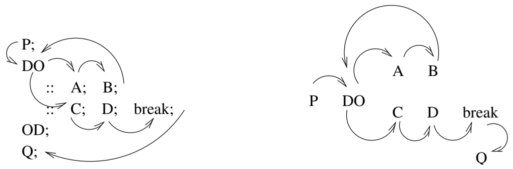

Figure 6.5: The processing of Promela DO statements

1. Form a transition between the first statement before the *do* statement and the *do* statement itself. (P to DO).
2. For every guarded command statement within the *do* statement, form a transition between the *do* statement and the guarded command statement. (DO to A, DO to C.)
3. For every exiting statement that follows a guarded command statement:
   (a) If it is not a *break* statement, form a transition between the exiting statement and the *do* statement. (B to DO.)
   (b) Otherwise, form a transition between the *break* statement and the first statement after the *do* statement. (BREAK to Q.)

Transitions for *atomic, d_step* and nested statements

The process for forming transitions for an *atomic, d_step* or nested statement is pictured in Figure 6.6.

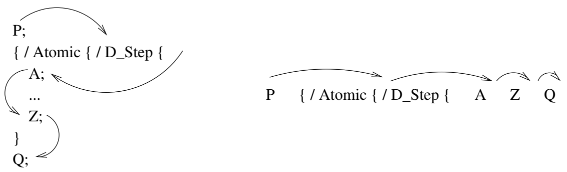

Figure 6.6: The processing of Promela nested, atomic and d_step statements

1. Form a transition between the *atomic, d_step* or nested statement and the first statement within the *atomic, d_step* or nested statement. ({ to A).
2. Form a transition between the last statement within the *atomic, d_step* or nested statement and the last statement after the *atomic, d_step* or nested statement. (Z to Q).

Transitions for GOTO and labeled statements

The process for forming transitions for a *goto* or labeled statement is pictured in Figure 6.7.

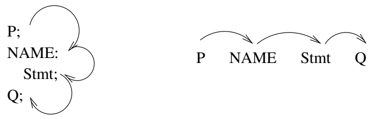

Figure 6.7: The processing of Promela named statements

1. For every *goto* statement:

(a) If the target of the *goto* statement was seen before, then form a transition between the *goto* statement and the corresponding labeled statement.

(b) Otherwise, remember to form the transition for this *goto* statement when the labeled statement is seen.

2. For every labeled statement:

   (a) Form a transition between the labeled statement and its child statement. (NAME to Stmt.)

   (b) Additionally, form transitions between outstanding *goto* statements with the same target as this label and this labeled statement. (GOTO to NAME.)

   Only a *noop* transition is made for a transition between a statement and the labeled statement (NAME) itself. Such *noop* transitions will be eliminated later.

6.5 Postprocessing the Promela program

After the code generator produces a nondeterministic finite automaton for a process, we perform certain postprocessing tasks that optimize the automaton or correct certain transitions of the automaton.

6.5.1 Elimination of no operation transitions

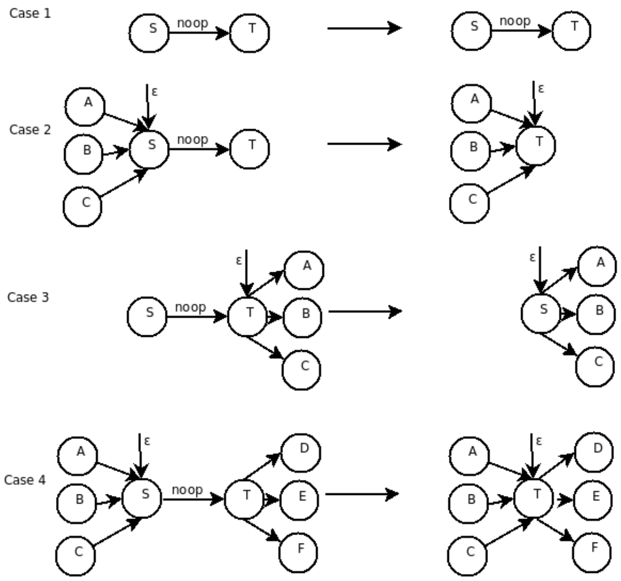

Figure 6.8: The 4 possible cases for eliminating a *noop* transition

Eliminate_Noop: This procedure eliminates transitions that represent only *noop* statements in order to produce a more optimized automaton, at least in terms of transitions. Earlier, we explained that the present approach to code generation uses *noop* or no-operation statements in order to guarantee that certain transitions can be formed even without obvious targets. In this sense, the *noop* statements are really necessary but implicit statements. It is then clear that our next task is to eliminate transitions that only represent *noop* statements so that we may produce a smaller, more optimized automaton. The basic idea of the algorithm is pictured in Figure 6.8.

6.5.2 Correction of exiting atomic transitions

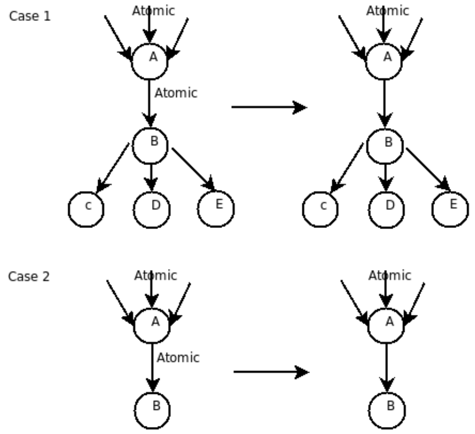

Figure 6.9: The 2 possible cases for correcting an exiting atomic transition

Correct_Atomic_Transitions: The code generator writes transitions correctly for atomic sequences save for one exception. The Promela manual page for the atomic sequence[6][Chapter 16] dictates that all transitions except for the last one within an atomic sequence must be atomic. The reason for this specification is that the atomic flag within a transition is used to control whether or not a process is allowed exclusive execution during the simulation of the automaton. Indeed, it is this exclusive execution of a process that enforces the semantics of the atomic sequence. While the entry transition and its subsequent transitions within an atomic sequence is correctly marked as being atomic, the last transition within the atomic sequence must necessarily be non-atomic in order to enforce the minimum number of atomic transitions necessary to provide the correct semantics of an atomic sequence. We use a straightforward depth-first search algorithm in order to find the last atomic transition — this is the case if it is atomic and it has no outgoing transitions or if it is atomic and all of its outgoing transitions are non-atomic — so that we may retrospectively mark it as being non-atomic. Similar depth-first search algorithms are used to correct transitions between labeled statements that are named with the special *accept, progress* and *end* labels that are used to specify liveness properties[6][Chapter 16]. The basic idea of the algorithm is pictured in Figure 6.9.

6.5.3 Writing the nondeterministic finite automaton for a process

End_New_Process: In the end, we write the NFA for a process by the following steps. First, we call *Eliminate_Noop* (Section 6.5.1) to eliminate *noop* transitions and *Correct_Atomic_Transitions* (Section 6.5.2) to correct exiting atomic transitions. Next, we write the nondeterministic finite automaton for the process in question by writing the process header followed by the optimized, corrected set of transitions. Finally, we perform some accounting on certain global variables, such as clearing some of them for the next process, incrementing the number of process types and ensuring that there are no *goto* statements that correspond to unknown labeled statements in this process.

6.6 Data structures, exceptions and LTL expressions

6.6.1 Data structures

Natural numbers and strings from a Promela program are stored in hashed containers in the Types.Lexer package during the compiler runtime. Consequently, AST nodes may contain only the cursors to these numbers or strings. This was a design decision to keep the size of certain attributes in the AST nodes constant no matter how large the actual values may be.

This optimization was, again, inspired by the JVM specification. One of the goals of the JVM specification is to produce byte code that is as compact as possible. One method for achieving this is to maintain a runtime constant pool where numbers and strings are stored in tables. A byte code instruction would then refer to an index into a table of numbers or strings instead of the literal itself.

The PVM does not strictly need such byte code compactness since the Erigone model checker values clarity over efficiency. In fact, there are technically a few minor inconstencies in the PVM byte code specification where some instructions may refer to some small literals. (For example, the *printf* instruction contains the number of arguments to be printed and the array load and store instructions contain the number of elements of the array in question.) One can then see that it is indeed possible for a byte code instruction to be able to refer to large literals. Although such a design helps to keep the byte code compact, we plan to replace this design in the near future with the simple and redundant use of constant literals in the abstract syntax tree nodes and byte code instructions. This simpler and redundant design would also help to render some of the compiler code more amenable to maintenance.

There are also a number of global variables in the code generator that helps us to keep track of the state of the code generation and act conditionally. For example, if a parent node has set a global variable to indicate that we are currently inside an atomic sequence, then we may write state transitions slightly differently.

6.6.2 Exceptions

The code generator may raise one of the following exceptions:

- File_Unopened: This exception is raised when the code generator is unable to open an automaton output file to write to.
- Unsupported_Construct: This exception is raised to warn the user that the code generator is presently unequipped to handle a particular Promela construct.
- Invalid_Operator: This exception is raised when an unexpected unary or binary operator is seen.
- Generic_Code_Generation_Error: This exception is raised when any unexpected error is seen, especially when a boolean condition that ensures correctness is not met.

6.6.3 Compiling Linear Temporal Logic expressions

As was briefly mentioned in Chapter 2, the user of the Erigone model checker may specify a Linear Temporal Logic (LTL) expression to check the correctness properties of the global state of the system of a running Promela program. The C SPIN model checker translates an LTL expression into an equivalent *never* claim which is used for verification of the system. The Erigone model checker translates an LTL expression into a Büchi automaton which is then used to form a never claim.

These details need not concern us. What is important is that an LTL expression contains Promela expressions which need compilation into byte code instructions. There is a specialized procedure or function in each of the major phases of the Pomegranate compiler that is dedicated to the task of translating a Promela expression into byte code instructions:

- Lexer.Fill_Buffer_With_String(F : String): This procedure copies a string to the buffer of the lexical analyzer and sets it up properly for further analysis.
- Parser.Parse_Any_Expr: This function returns an *Any_Expr* AST node after parsing the expression with the aid of the lexical analyzer.
- AST.Analyze_Any_Expr(T : Types.AST.Any_Expr_Node_Ptr): This procedure checks the *Any_Expr* node for semantic correctness.
- CodeGen.Generate_Any_Expr(T : Types.AST.Any_Expr_Node_Ptr): Finally, this function returns byte code instructions, which are encoded as an Ada string, for the given *Any_Expr* node with the aid of the global scope in runtime memory.

LTL is based on propositional calculus[3][Chapter 5]. Formulas of the propositional calculus are composed from atomic propositions (denoted by letters such as p and q) and the operators are given in Table 6.1.

An LTL formula is built from atomic propositions, operators of the propositional calculus and temporal operators. The temporal operators are given in Table 6.2.

Operator	Math	SPIN
not	$\neg$	`!`
and	$\wedge$	`&&`
or	$\vee$	`||`
implies	$\rightarrow$	`->`
equivalent	$\leftrightarrow$	`<->`

Table 6.1: Propositional operators of an LTL expression

Operator	Math	SPIN
always	$\Box$	`[]`
eventually	$\Diamond$	`<>`
until	U	`U`

Table 6.2: Temporal operators of an LTL expression

The $\Box$ and $\Diamond$ operators are unary whereas the U operators is binary. Temporal and propositional operators may be combined freely, so the following formula, given in both mathematical and Promela notation, is syntactically correct:

$\Box((p \wedge q) \rightarrow rU(p \vee r))$, `[] ( (p && q) -> r U (p || r) ).`

It should be read as: Always, (p and q) implies that r holds until (p or r) holds.

Promela expressions may be written as the atomic propositions of an LTL formula. It is important to keep in mind that Promela expressions within an LTL expression may be compiled only after a Promela program has been analyzed and compiled because such a Promela expression may refer to a global variable in this Promela program. As long as the semantic analyzer and the code generator keep enough information (for example, the global scope) after the compilation of the Promela program, any number of Promela expressions within LTL expressions may be translated afterwards.

The user specifies the LTL expression in a file which is passed as an optional argument, along with the required filename of the corresponding Promela program, at the command line to the Erigone model checker. Erigone then calls the Pomegranate compiler to translate the given Promela program and *then* the Promela expressions within the LTL expression.

6.7 Comments on code generation

The Promela grammar allows for an arbitrary order between variable declarations, channel assertions and other Promela statements in a sequence of statements. At the moment, no attempt is made to reorder these statements in a sequence in such a manner as to render certain tasks of code generation simpler. For example, in the process of generating byte code instructions and transitions for Promela statements, the code generator has to be careful to skip variable declarations and channel assertions. It would be a good task in the future to make the semantic analyzer reorder a sequence of a statements.

A crucial set of Promela constructs that is lacking implementation by the

Pomegranate compiler at the time of writing is channel operations. We plan to close this gap in the near future.

Chapter 7

Conclusions

> "For we can get some idea of a whole from a part, but never knowledge or exact opinion."
> — Polybius, ca. 150 B.C., Histories, Book I:4

Although we have succeeded in translating a functional, basic subset of the Promela language with the Pomegranate compiler, there remains much work to be done. The categories for future directions include translating the complete Promela grammar, optimizations and code review.

7.1 Future directions

7.1.1 Compiling the complete Promela grammar

The following Promela constructs are presently recognized by the Pomegranate compiler:

- Primitive data types
  - bit/bool, byte, mtype
- Arrays
  - Primitive data types
- Variable declarations
  - Primitive data types
  - Mtype declarations
- Statements
  - Assignments
  - Sequences
  - Atomic statements
  - Selection
  - Repetition

  - Jump statement
  - Labeled statements
  - Break statement
  - Else statement
  - Print statements
  - Assert statement
  - Expressions

- Expressions
  - Parenthesized expressions
  - Binary expressions
  - Unary expressions
  - Logical expressions
  - Variable references
  - Constants

- Processes
  - Active processes (limited to 1)
  - Init process

The following Promela constructs should be recognized by the Pomegranate compiler in the future:

- Primitive data types
  - short, int, unsigned

- Channel declarations

- Statements
  - Deterministic atomic statements
  - Channel send operations
  - Channel receive operations
  - Channel assertions
  - Unless statements

- Expressions
  - Predefined global variables
  - Predefined functions
  - Channel poll operations
  - Channel boolean functions
  - Tertiary conditions
  - Predefined unary operator for creating new processes

  - Remote references

- Processes
  - Active processes (up to 255 for each process type)
  - Parameters
  - Priority
  - Execution constraint

- Variable visibility (hidden, show)

- User-defined type declarations

- Never claims

- Event traces

In particular, we plan to introduce channels to the Pomegranate compiler in the near future. Constructs for specifying C code is not expected to be ever recognized by either Pomegranate or Erigone.

The lexical analyzer is sufficiently mature and efficient. After the addition of a C-like preprocessor, further development of the lexical analyzer will be halted.

The parser is also sufficiently mature and efficient. It only needs work in verifying the accuracy of its parsing, the addition of certain minor grammatical constructs, the complete left-factoring of the grammar and the tagging of abstract syntax tree nodes with an enumerated type for efficient discrimination. After that, further development of the parser will be halted.

The semantic analyzer needs more rigorous analysis for supported Promela constructs as well as equally strong analysis for Promela constructs to be supported in the future. Compile-time type checking should also be introduced to the semantic analyzer to make Promela a safer and more user-friendly language. As described earlier in the chapter on semantic analysis, the semantics of every Promela construct needs to be tightened by adhering to the rules as defined by the Promela manual pages.

Finally, the code generator also needs further work on both supported and unsupported Promela constructs. Most of the remaining work on the code generator involves the implementation of the rest of the Promela language.

7.1.2 Optimization

Since most Promela program models are quite small and the actual complexity of the program occurs during model checking, optimizing the compiler is a smaller concern. Nevertheless, there is much that we can do to make the compiler as efficient as possible. For example, the semantic analyzer and code generator presently use the Ada membership test in order to determine the precise type an AST node. Clearly, such membership tests must be replaced with the far more efficient discrimination of nodes by tagging them with the literals of an Ada enumerated type.

An example of a different kind of optimization, as described earlier in the chapter on code generation, would be the replacement of cursors into the number and string tables with the literals of the numbers and strings themselves. This substitution of design would make the compiler code simpler to understand and easier to maintain.

Another class of optimizations would be to emulate the same kind of optimizations performed by the C SPIN compiler on Promela programs[6][Chapter 9].

A final example of optimization would be the refactoring of the compiler code to group related or repeated code together in procedures or functions.

7.1.3 Code review

Finally, there is the matter of model checking the Pomegranate compiler itself. It will be good to produce proofs of correctness for certain key algorithms of the compiler, such as the algorithm in the semantic analyzer that rewrites subtrees of binary arithmetic expressions in order to adhere to binary operator precedence rules. The compiler will also need a code review in order to basically ensure that its translation of the Promela language is sound and complete.

7.2 Summary

In summary, the Pomegranate compiler for the Erigone model checker has been successful in translating basic Promela programs into corresponding nondeterministic finite automata. We expect to extend the capabilities of Pomegranate in the future so that it may efficiently and correctly handle the complete set of the Promela programming language constructs.

Appendix A

A deterministic finite automaton for lexical analysis

The lexer's deterministic finite automaton is pictured in the following figure:

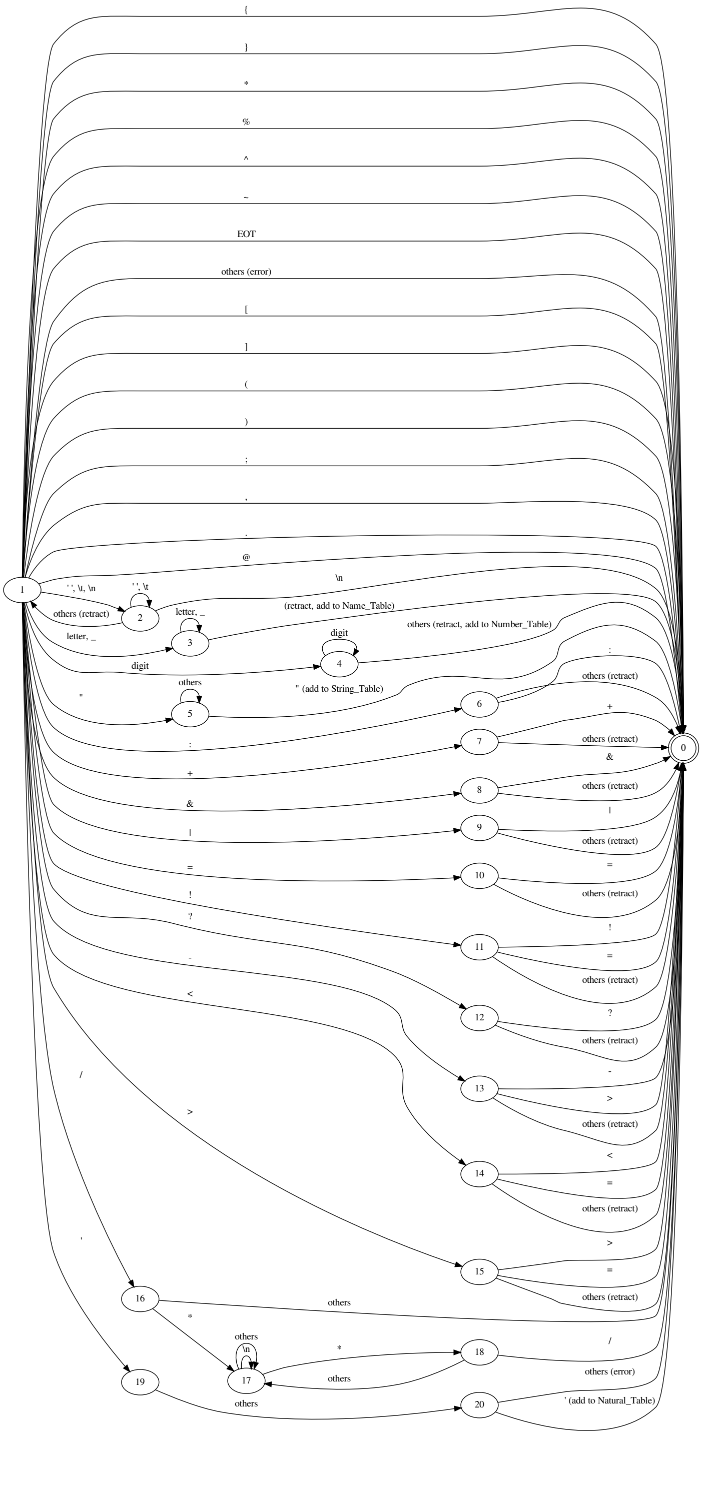

Figure A.1: The lexer automaton

Appendix B

A complete grammar for a Promela lexical analyzer

A Backus-Naur Form (BNF) or context-free grammar (CFG) of the lexer's deterministic finite automatonA is given by:

⟨*Token*⟩ ::= ⟨*Identifier*⟩
| ⟨*Name*⟩
| ⟨*String*⟩
| ⟨*Character*⟩
| ⟨*Whitespace*⟩
| ⟨*Special*⟩
| ‘`EOT`’

⟨*Identifier*⟩ ::= ⟨*Alpha*⟩ ⟨*Identifier*⟩ | ϵ

⟨*Alpha*⟩ ::= ‘`a`’
| ...
| ‘`z`’
| ‘`A`’
| ...
| ‘`Z`’
| ‘`_`’

⟨*Number*⟩ ::= ⟨*Digit*⟩ ⟨*Number*⟩
| ϵ

⟨*Digit*⟩ ::= ‘`0`’
| ...
| ‘`9`’

⟨*String*⟩ ::= ‘”’ ⟨*Many_Character*⟩ ‘”’

⟨*Character*⟩ ::= ‘’’ ⟨*ASCII*⟩ ‘’’

⟨*Many_Character*⟩ ::= ⟨*ASCII*⟩ ⟨*Many_Character*⟩
| ϵ

⟨*ASCII*⟩ ::= (any 8-bit ASCII character)

⟨*Whitespace*⟩ ::= space ⟨*Whitespace*⟩
| newline ⟨*Whitespace*⟩
| tab ⟨*Whitespace*⟩
| ϵ

⟨*Special*⟩ ::= {
| }
| (
|)
| [
|]
| %
| ˜
| ˆ
| ;
| ,
| .
| @
| *
| /
| +
| ++
| −
| −−
| − >
| :
| ::
| &
| &&
| |
| ||
| <
| <<
| <=
| >
| >>
| >=
| =
| ==
| !
| !!
| !=
| ?
| ??

Appendix C

A nearly complete Promela grammar

A nearly complete Promela grammar. See Chapter 4.

$\langle \mathit{spec} \rangle$ ::= $\langle \mathit{module} \rangle$ $\langle \mathit{spec\_prime} \rangle$

$\langle \mathit{spec\_prime} \rangle$ ::= $\langle \mathit{module} \rangle$ $\langle \mathit{spec\_prime} \rangle$
| ; $\langle \mathit{module} \rangle$ $\langle \mathit{spec\_prime} \rangle$
| ϵ

$\langle \mathit{module} \rangle$ ::= $\langle \mathit{proctype} \rangle$
| $\langle \mathit{init} \rangle$
| $\langle \mathit{never} \rangle$
| $\langle \mathit{trace} \rangle$
| $\langle \mathit{utype} \rangle$
| $\langle \mathit{mtype} \rangle$
| $\langle \mathit{decl\_lst} \rangle$

$\langle \mathit{proctype} \rangle$ ::= $\langle \mathit{proctype\_prime1} \rangle$ $\langle \mathit{proctype\_prime5} \rangle$ $\langle \mathit{name} \rangle$ ($\langle \mathit{proctype\_prime2} \rangle$) $\langle \mathit{proctype\_prime3} \rangle$ $\langle \mathit{proctype\_prime4} \rangle$ { $\langle \mathit{sequence} \rangle$ }

$\langle \mathit{proctype\_prime1} \rangle$ ::= $\langle \mathit{active} \rangle$
| ϵ

$\langle \mathit{proctype\_prime2} \rangle$ ::= $\langle \mathit{decl\_lst} \rangle$
| ϵ

$\langle \mathit{proctype\_prime3} \rangle$ ::= $\langle \mathit{priority} \rangle$
| ϵ

$\langle \mathit{proctype\_prime4} \rangle$ ::= $\langle \mathit{enabler} \rangle$
| ϵ

$\langle \mathit{proctype\_prime5} \rangle$ ::= PROCTYPE
| D_PROCTYPE

⟨*active*⟩ ::= ACTIVE ⟨*active_prime*⟩

⟨*active_prime*⟩ ::= [⟨*const*⟩]
  | ϵ

⟨*priority*⟩ ::= PRIORITY ⟨*const*⟩

⟨*enabler*⟩ ::= PROVIDED (⟨*any_expr*⟩)

⟨*const*⟩ ::= TRUE
  | FALSE
  | SKIP
  | ⟨*number*⟩

⟨*decl_lst*⟩ ::= ⟨*one_decl*⟩ ⟨*dec_lst_prime*⟩

⟨*decl_lst_prime*⟩ ::= ; ⟨*one_decl*⟩ ⟨*decl_lst_prime*⟩
  | ϵ

⟨*one_decl*⟩ ::= ⟨*one_decl_prime1*⟩ ⟨*typename*⟩ ⟨*ivar*⟩ ⟨*one_decl_prime2*⟩

⟨*one_decl_prime1*⟩ ::= ⟨*visible*⟩
  | ϵ

⟨*one_decl_prime2*⟩ ::= , ⟨*ivar*⟩ ⟨*one_decl_prime2*⟩
  | ϵ

⟨*visible*⟩ ::= HIDDEN
  | SHOW
  | LOCAL

⟨*typename*⟩ ::= BIT
  | BOOL
  | BYTE
  | SHORT
  | INT
  | MTYPE
  | CHAN
  | ⟨*uname*⟩
  | UNSIGNED
  | PID

⟨*uname*⟩ ::= ⟨*name*⟩

⟨*ivar*⟩ ::= ⟨*name*⟩ ⟨*ivar_prime1*⟩ ⟨*ivar_prime2*⟩

⟨*ivar_prime1*⟩ ::= [⟨*const*⟩]
  | ϵ

⟨*ivar_prime2*⟩ ::= = ⟨*ivar_prime3*⟩
  | ϵ

```
⟨ivar_prime3⟩ ::= ⟨any_expr⟩
  |  ⟨ch_init⟩

⟨ch_init⟩ ::= [ ⟨const⟩ ] OF { ⟨typename⟩ ⟨ch_init_prime⟩ }

⟨ch_init_prime⟩ ::= , ⟨typename⟩ ⟨ch_init_prime1⟩
  |  ϵ

⟨sequence⟩ ::= ⟨step⟩ ⟨sequence_prime⟩

⟨sequence_prime⟩ ::= ; ⟨sequence_prime_prime⟩
  |  -> ⟨step⟩ ⟨sequence_prime⟩
  |  ⟨step⟩ ⟨sequence_prime⟩
  |  ϵ

⟨sequence_prime_prime⟩ ::= ⟨step⟩ ⟨sequence_prime⟩
  |  ϵ

⟨step⟩ ::= ⟨stmt⟩ ⟨step_prime1⟩
  |  ⟨decl_lst⟩
  |  XR ⟨varref⟩ ⟨step_prime2⟩
  |  XS ⟨varref⟩ ⟨step_prime2⟩

⟨step_prime1⟩ ::= UNLESS ⟨stmt⟩
  |  ϵ

⟨step_prime2⟩ ::= , ⟨varref⟩ ⟨step_prime2⟩
  |  ϵ

⟨stmt⟩ ::= ⟨varref⟩ ⟨stmt_prime3⟩
  |  GOTO ⟨name⟩
  |  IF ⟨options⟩ FI
  |  DO ⟨options⟩ OD
  |  ATOMIC { ⟨sequence⟩ }
  |  D_STEP { ⟨sequence⟩ }
  |  { ⟨sequence⟩ }
  |  BREAK
  |  ELSE
  |  ⟨name⟩ : ⟨stmt⟩
  |  ⟨stmt_prime2⟩ ( ⟨string⟩ ⟨stmt_prime1⟩ )
  |  PRINTM ( ⟨name⟩ )
  |  ASSERT ⟨any_expr⟩
  |  ⟨any_expr⟩
  |  C_CODE { ... }
  |  C_EXPR { ... }
  |  C_DECL { ... }
  |  C_TRACK { ... }
  |  C_STATE { ... }

⟨stmt_prime1⟩ ::= , ⟨arg_lst⟩
  |  ϵ
```

```
⟨stmt_prime2⟩ ::= PRINT
  | PRINTF

⟨stmt_prime3⟩ ::= ⟨assign_prime⟩
  | ⟨send_prime⟩
  | ⟨receive_prime⟩

⟨varref⟩ ::= ⟨name⟩ ⟨varref_prime1⟩ ⟨varref_prime2⟩

⟨varref_prime1⟩ ::= [ ⟨any_expr⟩ ]
  | ϵ

⟨varref_prime2⟩ ::= . ⟨varref⟩
  | ϵ

⟨options⟩ ::= :: ⟨sequence options_prime⟩

⟨options_prime⟩ ::= :: ⟨sequence⟩ ⟨options_prime⟩
  | ϵ

⟨assign⟩ ::= ⟨varref⟩ ⟨assign_prime⟩

⟨assign_prime⟩ ::= = ⟨any_expr⟩
  | ++
  | --

⟨send⟩ ::= ⟨varref⟩ ⟨send_prime⟩

⟨send_prime⟩ ::= ! ⟨send_args⟩
  | !! ⟨send_args⟩

⟨receive⟩ ::= ⟨varref⟩ ⟨receive_prime⟩

⟨receive_prime⟩ ::= ? ⟨receive_prime_prime⟩
  | ?? ⟨receive_prime_prime⟩

⟨receive_prime_prime⟩ ::= ⟨recv_args⟩
  | < ⟨recv_args⟩ >

⟨poll⟩ ::= ⟨varref⟩ ⟨poll_prime⟩

⟨poll_prime⟩ ::= ? [ ⟨recv_args⟩ ]
  | ?? [ ⟨recv_args⟩ ]

⟨send_args⟩ ::= ⟨any_expr⟩ ⟨send_args_prime⟩

⟨send_args_prime⟩ ::= ( ⟨arg_lst⟩ )
  | ⟨arg_lst_prime⟩

⟨arg_lst⟩ ::= ⟨any_expr⟩ ⟨arg_lst_prime⟩
```

```
⟨arg_lst_prime⟩ ::= , ⟨any_expr⟩ ⟨arg_lst_prime⟩
  |  ϵ

⟨recv_args⟩ ::= ⟨recv_arg⟩ ⟨recv_args_prime⟩

⟨recv_args_prime⟩ ::= ⟨recv_args_prime_prime⟩
  |  ( ⟨recv_args⟩ )

⟨recv_args_prime_prime⟩ ::= , ⟨recv_arg⟩ ⟨recv_args_prime_prime⟩
  |  ϵ

⟨recv_arg⟩ ::= ⟨varref⟩
  |  EVAL ( ⟨varref⟩ )
  |  ⟨recv_arg_prime⟩ ⟨const⟩

⟨recv_arg_prime⟩ ::= -
  |  ϵ

⟨any_expr⟩ ::= ( ⟨any_expr⟩ ⟨any_expr_prime4⟩ ) ⟨any_expr_prime3⟩
  |  ⟨unarop⟩ ⟨any_expr⟩ ⟨any_expr_prime3⟩
  |  LEN ( ⟨varref⟩ ) ⟨any_expr_prime3⟩
  |  ⟨varref⟩ ⟨any_expr_prime5⟩ ⟨any_expr_prime3⟩
  |  ⟨const⟩ ⟨any_expr_prime3⟩
  |  TIMEOUT ⟨any_expr_prime3⟩
  |  NP_⟨any_expr_prime3⟩
  |  ENABLED ( ⟨any_expr⟩ ) ⟨any_expr_prime3⟩
  |  PC_VALUE ( ⟨any_expr⟩ ) ⟨any_expr_prime3⟩
  |  RUN ⟨name⟩ ( ⟨any_expr_prime1⟩ ) ⟨any_expr_prime2⟩ ⟨any_expr_prime3⟩
  |  ⟨chanpoll⟩ ( ⟨varref⟩ ) ⟨any_expr_prime3⟩

⟨chanpoll⟩ ::= FULL
  |  EMPTY
  |  NFULL
  |  NEMPTY

⟨any_expr_prime1⟩ ::= ⟨arg_lst⟩
  |  ϵ

⟨any_expr_prime2⟩ ::= ⟨priority⟩
  |  ϵ

⟨any_expr_prime3⟩ ::= ⟨binarop⟩ ⟨any_expr⟩
  |  ϵ

⟨any_expr_prime4⟩ ::= -> ⟨any_expr⟩ : ⟨any_expr⟩
  |  ϵ

⟨any_expr_prime5⟩ ::= @ ⟨name⟩
  |  : ⟨name⟩
  |  ⟨poll_prime⟩
  |  ϵ
```

```
⟨binarop⟩ ::= +
  |  −
  |  *
  |  /
  |  %
  |  &
  |  ^
  |  |
  |  >
  |  <
  |  >=
  |  <=
  |  ==
  |  !=
  |  <<
  |  >>
  |  ⟨andor⟩

⟨andor⟩ ::= &&
  |  ||

⟨unarop⟩ ::= ~
  |  -
  |  !

⟨init⟩ ::= INIT ⟨init_prime⟩ { ⟨sequence⟩ }

⟨init_prime⟩ ::= ⟨priority⟩
  |  ϵ

⟨never⟩ ::= NEVER { ⟨sequence⟩ }

⟨trace⟩ ::= ⟨trace_prime⟩ { ⟨sequence⟩ }

⟨trace_prime⟩ ::= TRACE
  |  NOTRACE

⟨utype⟩ ::= TYPEDEF ⟨name⟩ { ⟨decl_lst⟩ }

⟨mtype⟩ ::= MTYPE ⟨mtype_prime1⟩ { ⟨name⟩ ⟨mtype_prime2⟩ }

⟨mtype_prime1⟩ ::= =
  |  ϵ

⟨mtype_prime2⟩ ::= , ⟨name⟩ ⟨mtype_prime2⟩
  |  ϵ
```

Appendix D

The abstract syntax tree node type hierarchy

The abstract syntax tree node type hierarchy for the Promela grammar defined here is given by:

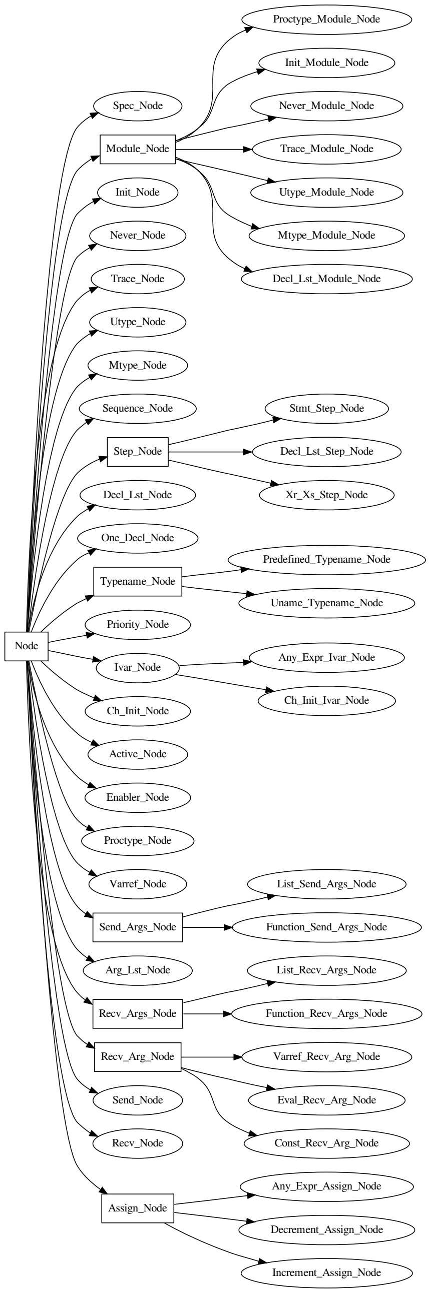

Figure D.1: The abstract syntax tree node type hierarchy

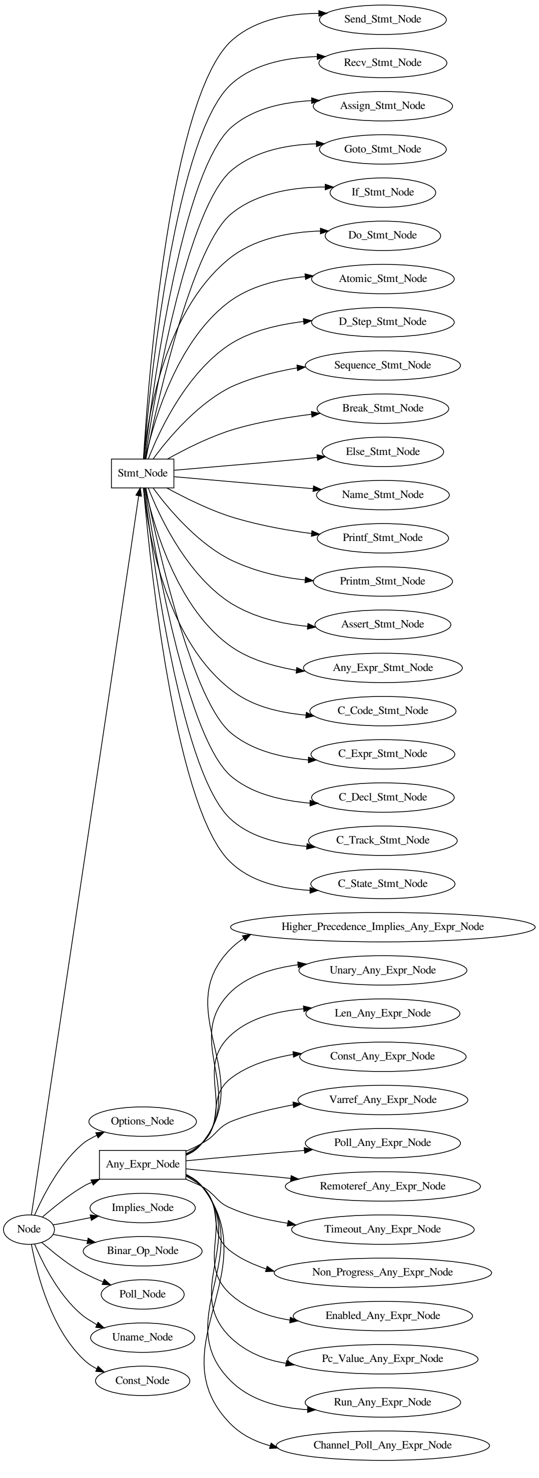

Figure D.2: The abstract syntax tree node type hierarchy

Appendix E

The Pomegranate virtual machine specification

Pomegranate Virtual Machine (PVM) Specification

0. Header

0.1 Abstract

This document tentatively specifies stack-based virtual machine and its instruction set for the Pomegranate compiler and the Erigone model checker.

0.2 Notes

This document is still under revision.

Among other things, the semantics of certain instructions should be specified to better fit the semantics of Promela rather than the Java programming language.

Some instructions may have semantics that are specified within quotation marks. The quotation marks are used to indicate that the semantics for an instruction are not radically different from the Java semantics for the corresponding instruction, although this does not mean that the semantics are identical.

Instructions for channel operations are also expected to be specified soon.

Revision: April 30 2009.

0.3 References

1. Tim Lindhold & Frank Yellin. “The Java Virtual Machine Specification.” Second Edition. Sun Microsystems, Inc., 1999.
    1. URL: http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

1. Data Types

1.1 Primitive Types and Values

- bit, whose values are 1-bit unsigned integers
- byte, whose values are 8-bit unsigned integers
- short, whose values are 16-bit signed two's-complement integers
- unsigned int, whose values are 32-bit unsigned integers
- int, whose values are 32-bit signed two's-complement integers

1.2 Integral Types and Values

- bit, from 0 to 1 (0 to 2^0), inclusive
- byte, from 0 to 255 (0 to 2^8-1), inclusive
- short, from -32,768 to 32,767 (-2^{15} to 2^{15}-1), inclusive
- unsigned int, from 0 to 2,147,483,647 (0 to 2^{31}-1), inclusive
- int, from -2,147,483,648 to 2,147,483,647 (-2^{31} to 2^{31}-1), inclusive

2. Runtime Data Areas

2.1 Heaps & Stacks

It is assumed that the Erigone interpreter will allocate and deallocate memory with the heap data structure for each instance of a process, since there may not be a way to tell in advance how many active process there will be (it is easy to imagine exponential process branching) and also because it may be easier to manipulate a heap than a stack. Nevertheless, the possibility of stacks is not excluded. How a frame is allocated for a process is up to the Erigone interpreter. (See Section 3.1.)

2.2 Runtime Constant Pool

Among other data, Promela strings and natural numbers are stored here in respective tables. An instruction may refer to a string in this table by referring to its index. This is especially useful for the "printf" instruction. A similar cases applies to natural number constants.

3.Frames

3.1 Frames

A frame may be allocated for each instance of a process, possibly indexed by the key “process_name.process_id”.

3.2 Local Variables

Local variables are indexed by their offsets (in terms of their size in bytes). It is assumed that the interpreter will be able to locate a local variable in the global state vector by using its index (or possibly the process name or number, _pid, in case of multiple processes; in which case a local variable may be indexed by, for example, the key “process_name.process_id.variable_name”.).

3.3 Operand Stacks

Each frame is assumed to have an operand stack that will be used in executing a sequence of byte code instructions for each Promela statement.

4. Instruction Set Summary

NOTE: The terms “operator” and “instruction” are interchangeable here.

4.1 Arithmetic

Mnemonic:

iadd

Operation:

Add int

Operand Stack:

...,value1,value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a runtime exception.”[1]

Mnemonic:

idec <index> <const>

Operation:

Decrement local variable by constant

Operand Stack:

No change

Description:

“The *index* is an unsigned byte that must be an index into the local variable array of the current frame. The *const* is an immediate unsigned byte. The local variable at *index* must contain an `int`. The value *const* is first sign-extended to an `int`, and then the local variable at *index* is decremented by that amount.”[1]

Mnemonic:

idiv

Operation:

Divide int

Operand Stack:

...,value1,value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is the value of the Promela programming language expression *value1* / *value2*. The *result* is pushed onto the operand stack.

An `int` division rounds towards 0; that is, the quotient produced for `int` values in *n/d* is an `int` value *q* whose magnitude is as large as possible while satisfying $|d \cdot q| \leq |n|$. Moreover, *q* is positive when $|n| \geq |d|$ and *n* and *d* have the same sign, but *q* is negative when $|n| \geq |d|$ and *n* and *d* have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for the `int` type, and the divisor is *-1*, then overflow occurs, and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case. ”[1]

Runtime Exception:

“If the value of the divisor in an `int` division is 0, *idiv* throws an `ArithmeticException`.”[1]

Mnemonic:

iinc <index> <const>

Operation:

Increment local variable by constant

Operand Stack:

No change

Description:

“The *index* is an unsigned byte that must be an index into the local variable array of the current frame. The *const* is an immediate unsigned byte. The local variable at *index* must contain an `int`. The value *const* is first sign-extended to an `int`, and then the local variable at *index* is incremented by that amount.”[1]

Mnemonic:

imul

Operation:

Multiply int

Operand Stack:

...,value1,value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* * *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *imul* instruction never throws a runtime exception.”[1]

Mnemonic:

ineg

Operation:

Negate int

Operand Stack:

...,value => ...,result

Description:

“The *value* must be of type `int`. It is popped from the operand stack. The `int` *result* is the arithmetic negation of *value*, *-value*. The *result* is pushed onto the operand stack.

For `int` values, negation is the same as subtraction from zero. Because the virtual machine uses two's-complement representation for integers and the range of two's-complement values is not symmetric, the negation of the maximum negative `int` results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all `int` values `x`, -`x` equals (~`x`) + `1`.”[1]

Mnemonic:

irem

Operation:

Remainder int

Operand Stack:

...,value1, value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* - (*value1* / *value2*) * *value2*. The *result* is pushed onto the operand stack.

The result of the *irem* instruction is such that `(a/b)*b` + `(a%b)` is equal to `a`. This identity holds even in the special case in which the dividend is the negative `int` of largest possible magnitude for its type and the divisor is `-1` (the remainder is `0`). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative and can be positive only if the dividend is positive. Moreover, the magnitude of the result is always less than the magnitude of the divisor. ”[1]

Runtime Exception

“If the value of the divisor for an `int` remainder operator is 0, *irem* throws an `ArithmeticException`.”[1]

Mnemonic:

isub

Operation:

Subtract int

Operand Stack:

...,value1, value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1 - value2*. The *result* is pushed onto the operand stack.

For `int` subtraction, `a - b` produces the same result as `a + (-b)`. For `int` values, subtraction from zero is the same as negation.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *isub* instruction never throws a runtime exception.”[1]

4.2 Memory manipulation

Mnemonic:

bipush <byte>

Operation:

Push byte

Operand Stack:

... => ...,value

Description:

“The immediate *byte* is sign-extended to an `int` *value*. That *value* is pushed onto the operand stack.”[1]

Mnemonic:

bit_load <index>, byte_load <index>, short_load <index>, unsigned_load <index>, iload <index>

Operation:

Load int from a local variable

Operand Stack:

... => ...,value

Description:

“The *index* is an unsigned byte that must be an index into the local variable array of the current frame. The local variable at *index* need not contain an `int`, but it must be sign-extended to an `int` if it is of a type smaller than an `int`. This possibly sign-extended version of the value of the local variable at *index* is pushed onto the operand stack.”[1]

Mnemonic:

iconst <index>

Operation:

Push as an int constant a natural value at *index* into the natural number table of the runtime constant pool

Operand Stack:

... => ...,<value>

Description:

“Push the `int` constant <*i*> onto the operand stack. ”[1]

Mnemonic:

bit_store <index>, byte_store <index>, short_store <index>, unsigned_store <index>, istore <index>

Operation:

Store int into a variable

Operand Stack:

...,value => ...

Description:

“The *index* is an unsigned byte that must be an index into the local variable array of the current frame. The *value* on the top of the operand stack must be of type `int`, but the target that is the local variable at *index* need not be; in case it is of a type smaller than an `int`, it must be truncated to fit the type of the local variable. It is up to the interpreter to decide whether or not the truncation is a runtime error for any reason. *value* is popped from the operand stack, and the value of the local variable at *index* is set to *value* or a truncated version of it.”[1]

Mnemonic:

check_index <length>

Operation:

Assert that the index on stack is less than the specified length

Operand Stack:

… , index => ...

Description:

The *length* is an unsigned integer that must specify the length of an array. The *index* on top of the operand stack must be of type `int`. This instruction will then assert that 0 ≤ *index* < *length*. This instruction is to be followed with either an array load or store.

Mnemonic:

bit_aload <offset> <size>, byte_aload <offset> <size>, short_aload <offset> <size>, unsigned_aload <offset> <size>, iaload <offset> <size>

Operation:

Load int from an array

Operand Stack:

…,index => ...,value

Description:

"The *index* is an unsigned integer that must be an index into an array in the local variable array of the current frame. *offset* and *size* are unsigned integers. The local variable at (*offset* + *index* * *size*) need not contain an `int`, but it must be sign-extended to an `int` if it is of a type smaller than an `int`. This possibly sign-extended version of the value of the local variable at (*offset* + *index* * *size*) is pushed onto the operand stack."[1]

Mnemonic:

bit_astore <offset> <size>, byte_astore <offset> <size>, short_astore <offset> <size>, unsigned_astore <offset> <size>, iastore <offset> <size>

Operation:

Store int into an array

Operand Stack:

…,value,index => ...

Description:

“The *index* is an unsigned integer that must be an index into an array in the local variable array of the current frame. *offset* and *size* are unsigned integers. The *value* on the top of the operand stack must be of type `int`, but the target that is the local variable at (*offset* + *index* * *size*) need not be; in case it is of a type smaller than an `int`, it must be truncated to fit the type of the local variable. It is up to the interpreter to decide whether or not the truncation is a runtime error for any reason. *value* is popped from the operand stack, and the value of the local variable at (*offset* + *index* * *size*) is set to *value* or a truncated version of it.”[1]

4.3 I/O

Mnemonic:

printf <index> <n>

Operation:

Print a string with n parameters

Operand Stack:

...,parameter_n,...,parameter_1 => ...

Description:

The *index* is an unsigned byte that must be an index into the string table of the runtime constant pool. The *n* is an immediate unsigned byte. The entry at *index* must contain a string. *parameter_n* must be of type `int`. The string, formatted with the parameters, will be printed to the console.

Mnemonic:

printm

Operation:

Print a byte

Operand Stack:

..., value => ...

Description:

value will be of type `int` on the stack but it must be within the range of type `byte`. It will be printed to the console.

4.4 Bitwise & Logical Operations and Boolean Conditions

Mnemonic:

logic_and

Operation:

Logical NOT int

Operand Stack:

...,value1, value2 => ...,result

Description:

A convenient instruction. *value1* and *value2* must be of type `int`. It is popped from the operand stack. An `int` *result* is calculated by taking the logical AND of *value1* and *value2* (result == 1 if and only if *value1* == 1 and *value2* == 1; otherwise result == 0). The *result* is pushed onto the operand stack.

Mnemonic:

logic_else

Operation:

Distinguishing an else transition from other true transitions

Operand Stack:

... => ...,1

Description:

A convenient instruction. It is used solely to help the interpreter to distinguish the singular 'else' transition from other executable, 'true' transitions.

Mnemonic:

iand

Operation:

Bitwise AND int

Operand Stack:

...,value1,value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. They are popped from the operand stack. An `int` *result* is calculated by taking the bitwise AND (conjunction) of *value1* and *value2*. The *result* is pushed onto the operand stack.”[1]

Mnemonic:

icmpeq, icmpne, icmplt, icmple, icmpgt, icmpge

Operation:

Compare int

Operand Stack:

...,value1, value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparison are as follows:

- *icmpeq*: result == 1 if and only if *value1* = *value2*; otherwise result == 0
- *icmpne*: result == 1 if and only if *value1* ≠ value2; otherwise result == 0
- *icmplt*: result == 1 if and only if *value1* < value2; otherwise result == 0
- icmple: result == 1 if and only if *value1* ≤ *value2*; otherwise result == 0
- *icmpgt*: result == 1 if and only if *value1* > *value2*; otherwise result == 0
- *icmpge*: result == 1 if and only if *value1* ≥ *value2*; otherwise result == 0

”[1]

Mnemonic:

ifeq <branchbyte>, ifne <branchbyte>, iflt <branchbyte>, ifle <branchbyte>, ifgt <branchbyte>, ifge <branchbyte>

Operation:

Branch if int comparison with zero succeeds

Operand Stack:

...,value => ...,result

Description:

“An instruction used solely to support tertiary conditions. The *value* must be of type `int`. It is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows:

- *ifeq* succeeds if and only if *value* = 0
- *ifne* succeeds if and only if *value* ≠ 0
- *iflt* succeeds if and only if *value* < 0
- *ifle* succeeds if and only if *value* ≤ 0
- *ifgt* succeeds if and only if *value* > 0
- *ifge* succeeds if and only if *value* ≥ 0

If the comparison succeeds, *the* unsigned *branchbyte* is used as an unsigned 8-bit offset. Execution then proceeds at the index of the array of byte code for the statement that is the offset from the index of the opcode of this *if<cond>* instruction. The target index must be that of an instruction within the statement that contains this *if<cond>* instruction.

Otherwise, execution proceeds at the index of the instruction following this *if<cond>* instruction. result is the int value of the execution of either branch of instruction.”[1]

Mnemonic:

inot

Operation:

Bitwise NOT int

Operand Stack:

...,value => ...,result

Description:

value must be of type `int`. It is popped from the operand stack. An `int` *result* is calculated by taking the bitwise NOT of *value*. The *result* is pushed onto the operand stack.

Mnemonic:

ior

Operation:

Bitwise OR int

Operand Stack:

...,value1,value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. They are popped from the operand stack. An `int` *result* is calculated by taking the bitwise inclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.”[1]

Mnemonic:

ishl

Operation:

Arithmetic/Logical Shift left int

Operand Stack:

...,value1, value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting *value1* left by *s* bit positions, where *s* is the value of the low 5 bits of *value2*. The *result* is pushed onto the operand stack.”[1]

Notes:

“This is equivalent (even if overflow occurs) to multiplication by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.”[1]

Mnemonic:

ishr

Operation:

Arithmetic shift right int

Operand Stack:

...,value1, value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting *value1* right by s bit positions, with sign extension, where s is the value of the low 5 bits of *value2*. The *result* is pushed onto the operand stack.”[1]

Notes:

“The resulting value is $\lfloor value1/2^s \rfloor$, where s is *value2* & 0x1f. For nonnegative *value1*, this is equivalent to truncating `int` division by 2 to the power s. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.”[1]

Mnemonic:

iushr

Operation:

Logical shift right int

Operand Stack:

...,value1, value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting *value1* right by s bit positions, with zero extension, where s is the value of the low 5 bits of *value2*. The *result* is pushed onto the operand stack.”[1]

Notes:

“If *value1* is positive and s is *value2* & 0x1f, the result is the same as that of *value1* >> s; if *value1* is negative, the result is equal to the value of the expression (*value1* >> s) + (2 << ~s). The addition of the (2 << ~s) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 31, inclusive.”[1] (Where “>>” is Java's arithmetic shift right and “<<” is Java's arithmetic/logical shift left.)

Mnemonic:

ixor

Operation:

Boolean XOR int

Operand Stack:

...,value1, value2 => ...,result

Description:

“Both *value1* and *value2* must be of type `int`. They are popped from the operand stack. An `int` *result* is calculated by taking the bitwise exclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.”[1]

Mnemonic:

goto <branchbyte>

Operation:

Branch always

Operand Stack:

No change

Description:

“The unsigned byte *branchbyte* is an unsigned 8-bit offset. Execution proceeds at that offset from the index of this *goto* instruction. The target address must be an index of an instruction within the array of byte code for the statement that contains this *goto* instruction.”[1]

Mnemonic:

logic_not

Operation:

Logical NOT int

Operand Stack:

...,value => ...,result

Description:

A convenient instruction. *value* must be of type `int`. It is popped from the operand stack. An `int` *result* is calculated by taking the logical NOT of *value* (result == 1 if and only if *value* == 0; otherwise result == 0). The *result* is pushed onto the operand stack.

Mnemonic:

logic_or

Operation:

Logical OR int

Operand Stack:

...,value1, value2 => ...,result

Description:

A convenient instruction. *value1* and *value2* must be of type `int`. It is popped from the operand stack. An `int` *result* is calculated by taking the logical OR of *value1* and *value2* (result == 0 if and only if *value1* == 0 and *value2* == 0; otherwise result == 1). The *result* is pushed onto the operand stack.

4.5 Instructions to emulate built-in functions

Mnemonic:

assert

Operation:

Assertion of inequality of an int value to 0

Operand Stack:

...,value => ...

Description:

An instruction to emulate a built-in function. *value* must be of type `int`. It is popped from the operand stack. If *value* == 0, then no further execution of possible transitions is made for the process instance in question; the frame for this process instance must be deallocated and an assertion violation should be reported. Otherwise, the evolution of the finite state machine for this process is continued.

Mnemonic:

noop

Operation:

No operation; idle

Operand Stack:

No change

Description:

CPU idle operation.

Mnemonic:

halt

Operation:

Halt all execution for the process in execution.

Operand Stack:

No change

Description:

CPU halt operation.

5. Conventions

- **It is not strictly true that there may not be any additional value left on the operand stack after each execution of a statement.** Each executable statement will leave an integer result on the stack. It is assumed that after having executed the entire array of byte code for a statement, the interpreter will finally pop the stack (which should then be of size == 1) in order to obtain the value of the execution (a nonzero value means the statement is executable; 0 means the statement is not executable).
- **Once the executability of a statement has been established by executing the array of byte code for that statement, it must not be executed again**; instead, only the current state of the finite state machine for the process in question must be changed appropriately.

Bibliography

[1] Ravi Sethi Alfred V. Aho, Monica S. Lam and Jeffrey D. Ullman. *The SPIN Model Checker: Primer and Reference Manual.* Addison Wesley Professional, 2003.

[2] John Barnes. *Programming in Ada 2005.* Addison Wesley, 2006.

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

[4] Gregory J. Chaitin. *The Limits of Mathematics.* Springer-Verlag London, 2003.

[5] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[6] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison Wesley Professional, 2003.

[7] Tom Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Sun Microsystems, Inc., second edition, 1999.

[8] Nassim N. Taleb. *The Black Swan.* Random House, 2007.