Master's Thesis
in Computer Science

The SpinJ Model Checker

A fast, extensible, object-oriented model checker

Author:
Marc de Jonge
m.dejonge-1@student.utwente.nl

Supervisor:
Theo Ruys
ruys@cs.utwente.nl

Committee:
dr. ir. Theo Ruys
dr. ir. Arend Rensink
dr. ir. Marieke Huisman

Research group:
Formal Methods and Tools
Faculty of EEMSC
University of Twente

Abstract

Model checking has grown to be a practical addition to the field of formal verification. One model checker that has proven itself very useful in practice is SPIN, which is built to validate models that are written in PROMELA. It can be used to search for deadlocks, assertions, liveness properties and even LTL properties.

This thesis describes the design and implementation of SPINJ, a reimplementation of SPIN in Java. SPINJ is designed to be behave similarly to SPIN, but to be more *extendible* and *reusable*. To achieve this the conceptual framework of Kattenbelt is used as the basis for the design of the SPINJ library, using the three layers that he describes.

Firstly, the *generic layer* is the lowest layer which uses a basic model with states and transitions. On this layer all storage methods, search algorithms and simulation techniques are implemented. The *abstract layer* describes a concurrent model with processes that is an extension of the model in the generic layer. This knowledge of processes within the model makes it possible to implement partial order reduction here. Finally the *tool layer* is implemented for the PROMELA language support.

SPINJ also contains a PROMELA compiler that generates Java code to represent the given PROMELA model. This Java code can be compiled and then verified using the SPINJ library. Since this library contains all the actual algorithms, the generated code can be relatively small, only describing the model itself. Also all algorithms that are available can be used with any model and can be selected at runtime.

Despite the fact that SPINJ is designed to be extendible and reusable, it is not slow; using the BEEM benchmark, this thesis has shows that SPINJ is on average only 3.5 times slower than SPIN and it uses less memory in most of the cases.

Contents

List of Figures

List of Tables

Preface

First I would like to thank Theo Ruys for his support over the course of this project. Without his constructive comments and discussions during our regular sessions, I would not have been able to complete this assignment. Furthermore I would like to thank Marieke Huisman for her last minute review.

I would also like to thank my girlfriend Sandra for supporting and helping me all this time, as well as my parents for their support and encouragment. Finally, I would like to thank Natascha Agricola, my manager at TNO, for her support and understanding my situation.

Chapter 1

Introduction

In any software lifecycle process *testing* plays an important role [4], taking up many resources. It helps to see if the system meets its requirements, by finding faults in the implementation. A limitation of most testing techniques is that it only covers a few predefined runs, therefore not finding all possible errors. This is especially true for concurrent systems, where different threads can interfere with each other.

To complement standard testing techniques, it is possible to use *model checking* [29] [7]. This is a formal method that rather than executing a couple of selected runs, tries to verify every possible run in the system. Typically a model checker does not run on the implemented system itself, but on a *model* of the system, with specified *properties* that represent the requirements. Such a formal verification can then be used to verify that the intended model is correct.

Three computer scientists - Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis - pioneered model checking in the 1980s, for which they received the ACM Turing Award [3] of 2007. From [3]: "[They have received the award] for their original and continuing research in a quality assurance process known as Model Checking. Their innovations transformed this approach from a theoretical technique to a highly effective verification technology that enables computer hardware and software engineers to find errors efficiently in complex system designs. This transformation has resulted in increased assurance that the systems perform as intended by the designers."

One of these model checkers that has proved to be very useful in practice is SPIN [1]. The creator of SPIN, Gerard Holzmann, even received the ACM Software System Award, because (from [2]) "(ACM) has recognized Dr. Gerard Holzmann for his contribution to a widely used software package called SPIN that quickly detects defects in networked computers, making them more reliable".

The models that can be checked by SPIN are written in the PROMELA language, which is specially designed to allow for dynamic creation of concurrent processes. This allows a user to model a distributed system or a protocol. In PROMELA models, communication between different processes can be defined through *channels*, which can be synchronous (i.e. rendez-vous) or asynchronous (i.e. buffered).

SPIN can validate models written in PROMELA for deadlocks, assertion violations, lifeness properties and even properties expressed in LTL (linear temporal logic, see [17]). It is designed to scale well and to handle even very large problem sizes. To achieve this, it is implemented is plain C, using a monolithic design that is fully optimised for PROMELA models. Unfortunately, this design makes it more difficult to implement new algorithms or reuse parts of the implementation.

1.1 Problem statement

Is it possible to re-implement the core of SPIN *in Java whilst being competitive in terms of memory consumption and runtime behaviour?*

This question comes from two underlying questions. First to see if it is possible to create a program like SPIN that is created using modern, object-oriented programming techniques. This would make it easier to reuse parts of the program and to extend it. Secondly to see if Java has the capability to execute such a program in acceptable time and memory usage.

The program that has been created during this thesis is called SPINJ. The goal is to implement it such that the following conditions hold:

- The design and implementation of SPINJ should be object-oriented, *reusable* and well documented.
- The design and implementation of SPINJ should be *extendable*, such that other algorithms are easily added. It should be possible to extend the modelling language.
- The performance of SPINJ should be *comparable* to SPIN (both in time and space), where one order of magnitude of difference is acceptable.
- SPINJ's simulator and verifier should use the same Java code.
- SPINJ should support several of SPIN's optimisation algorithms, e.g. partial order reduction, bitstate hashing and hash compaction.
- The output and parameters of SPINJ should resemble the ones of SPIN as much as possible.

1.2 Overview

The rest of this document describes the development of SPINJ, a new model checker that can verify models written in PROMELA. We will start with some background information in Chapter 2, where we will discuss some other model checkers like SPIN.

After that, Chapter 3 will give an overview of the architecture of SPINJ, where the conceptual framework of Mark Kattenbelt will be used as reference. In his thesis [32] he describes how a model checker should be designed such that it is reusable and extendable.

In the chapters 4, 5 and 6 the implementation of the 3 different layers is explained. Each of these layers describes more details of the model, where the lowest layer only knows about states and transitions, the highest layer knows all specific PROMELA features.

To see how SPINJ performs, a benchmark is used to test it against SPIN in Chapter 7. Finally the conclusion in Chapter 8 discusses whether all the goals are reached and proposes some future work.

Chapter 2

Related work

This chapter discusses model checking in general and different existing explicit model checkers. Then it describes the design and implementations of Spin and the conceptual framework of Kattenbelt. This will be kept brief, a more extensive overview is written in [32].

2.1 Model checking

Model checking [35] is the formal *verification* of a *model* against a *specification*. To do this it uses a verification algorithm to check whether model ($\mathcal{M}$) models the specification (p), which can be written as $\mathcal{M} \models p$. When an error is found, a model checker shows an error trace on which p does not hold.

2.1.1 Model Specification

The model is a formal description of a system. This model can be derived from a system, automatically or manually, but can also be created before the system is built. Validating a model before the implementation is used to validate the design or protocol. Models usually abstract from irrelevant aspects of the system, to make the validation easier. Using the system itself as a model is also possible.

A model can be represented in many different forms. Most models are specified in a language and are first transformed to a mathematical model. This mathematical model can then be used by the verification algorithm to prove that all properties of the specification hold.

There are some differences between mathematical models, although they are mostly based on a *Labelled Transition System*. For example, some types add the notion of clocks (e.g. Timed Automata [5] or Continuous-Time Markov Chains [6]).

A Labelled Transition System (LTS) is one of the most elemental ways to describe a model. It can be defined as a 4-tuple $\mathcal{M} = (S, s_0, \Sigma, T)$, where S is the set of states, $s_0 \in S$ the initial state, Σ is the set of labels (the alphabet) and $T \subseteq S \times \Sigma \times S$ is the set of transitions.

There are other variants based on an LTS system. For example, a finite

state machine adds a set of final (or ending) states $F \subseteq S$ to the LTS. Such a state machine accepts a (regular) language with words of finite length.

Another variation is a labelled Büchi automaton, which adds a set of accepting states ($A \in S$). A Büchi automaton accepts a language with infinite words. For a Büchi automaton to accept such a word, it must pass an infinite number of accepting states. In model checking Büchi automata are used to define "bad" behaviour, as we can define a language that specifies that "bad" behaviour.

2.1.2 Property Specification

There are different ways to define properties that should be checked over an LTS. Three of those are *invariants*, *Linear Temporal Logic* (LTL) and *Computation Tree Logic* (CTL). All these properties are based on the use of *atomic propositions*.

Atomic propositions These are the most basic elements of any proposition. They simply describe a condition about one part of the model, in a certain state. For example, "$x > 0$" or "process A is blocked" are such atomic propositions.

Invariants The atomic propositions can be combined with logic operators to form formulae. For example, if p and q are valid propositions, $\neg p$, $p \vee q$ and $p \wedge q$ are also propositions. The invariants describe *safety properties* of a single state in the model. This could be defined as *assertions* on one of the labels (checking the last state) or as global *invariants* that should hold for every state. These safety properties can be verified by an exhaustive search over all the possible states. When a violation was found, it can then return a *finite trace* to the error.

Linear Temporal Logic Invariants can be used to specify a property of one state, but it can not be used to define properties over *paths* in the model. LTL [27] expressions are designed to specify properties over a path in the model (e.g. lifeness properties). For example, it can be used to define that a certain atomic property holds until some other property becomes true.

Computational Temporal Logic As LTL properties are defined over paths, it can not distinguish between paths. CTL [27] can be used in cases where we want to make such a distinction. For example, it is possible with CTL to define that from a certain state there is at least one path where some property p always holds.

2.2 Model checkers

We can split model checkers into two different groups:

- *Explicit state model checkers* generate each possible state of the system and use those states to validate different properties. It usually consists

of an exhaustive search over all the states and an on-the-fly validation. Examples of explicit model checkers are SPIN [1], NIPS [39] and JPF [9].

- *Symbolic model checkers* do not represent individual states, but they store sets of states. These sets can be represented symbolically (e.g. in a binary decision diagram). This approach is more effective for checking CTL propositions. One example of a symbolic model checker is SMV [34].

There are other variations (e.g. bounded model checkers like NuSMV [10]), but in this thesis we are going to focus on explicit model checking. The rest of this section is going to discuss a couple of model checkers, showing the design of each and how they are implemented.

2.2.1 SPIN

SPIN [1] is arguably the most used model checker, which has proven itself on industrial size problems [23]. Gerard Holzmann, the creator of SPIN, even received an ACM Software System award [2]. SPIN is still further improved. Recently version 5 has been released, which adds support for multicore systems [22].

The model specification language for SPIN is called PROMELA. PROMELA can be used to model several processes that can be started dynamically. More details about this language will be discussed in Chapter 6.

Architecture

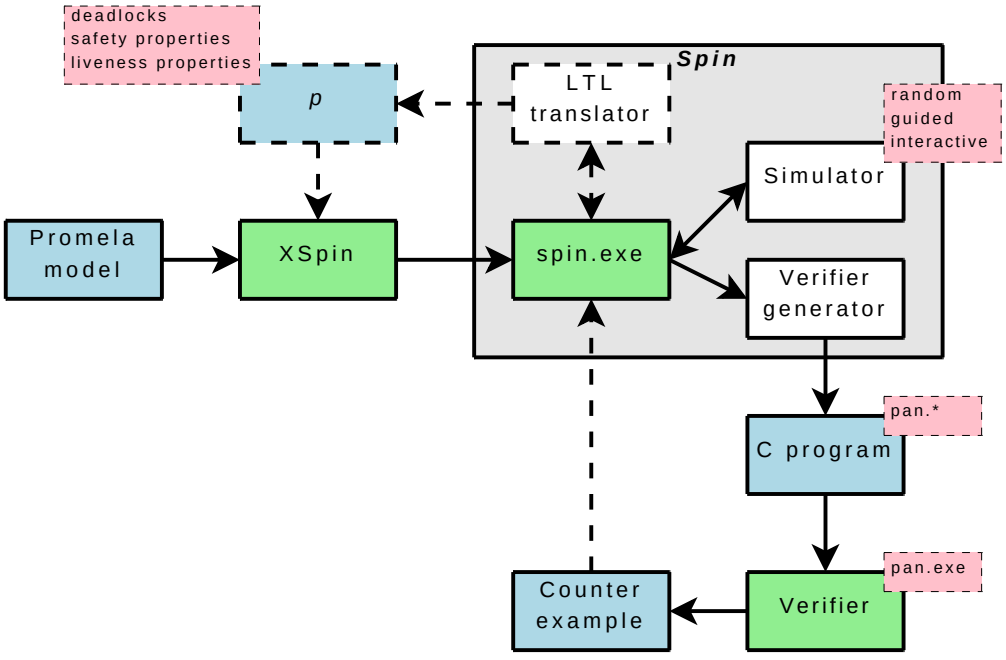

Figure 2.1: An overview of the architecture of SPIN.

In [19] the basic structure of SPIN is discussed, which is also shown in Figure 2.1. SPIN itself is the PROMELA parser that can read models that are written in PROMELA and do two things with it:

When the `-a` option is not set when calling SPIN, a *simulation* is started. This simulation can either be guided by the user (interactive), by a generated trace-file (guided) or at random.

When the `-a` option is set, SPIN will use the *verifier generator* to produce the *verifier*. This verifier is highly optimised and written in ANSI C. It can be compiled using any C compiler. When the verifier is run and it finds an error, it can produce a counter-example in the form of a trace-file. This file can be used to guide the simulation.

Implementation

SPIN is completely written in ANSI C, with speed as its main goal. This has a disadvantage in terms of reusability and extensibility. SPIN is written as a "black box", meaning that the whole application is tightly coupled and it is difficult to identify the different parts.

Also, to keep SPIN as fast as possible, all of the options regarding to the search algorithm, storage techniques etc. are implemented using pre-processor instructions in the generated C-code. For example, if we want a model to use hash compaction [40], we need to compile the generated C-code with the preprocessor option `-DHC`.

2.2.2 Kattenbelt's conceptual framework

In his thesis [32], Mark Kattenbelt describes a conceptual framework that unifies the model checking domain. Kattenbelt also created an implementation based on that design that can handle models that are written in `Prom+`, a simpler variation based on PROMELA that adds the notion of dynamic storage and pointers.

Architecture

The conceptual framework contains three layers, also shown in Figure 2.2. From [32]:

- "The *generic layer* provides algorithms for certain types of models. [...] It is not feasible to define just one generic layer for all models, this is due to the diversity of models in the model checking domain. For instance, the fields of explicit-state, symbolic, bounded and probabilistic model checking are too different to be encapsulated within the same generic layer, and should probably be defined in separate generic layers."

  "The most important requirement of such a generic layer is that the algorithms in this layer are oblivious to the abstract layer. Also, the generic layer should provide a means in which a model can be defined such that they can use the generic functionality. In Figure 2.2 this is the abstract base class `StateSpace`."

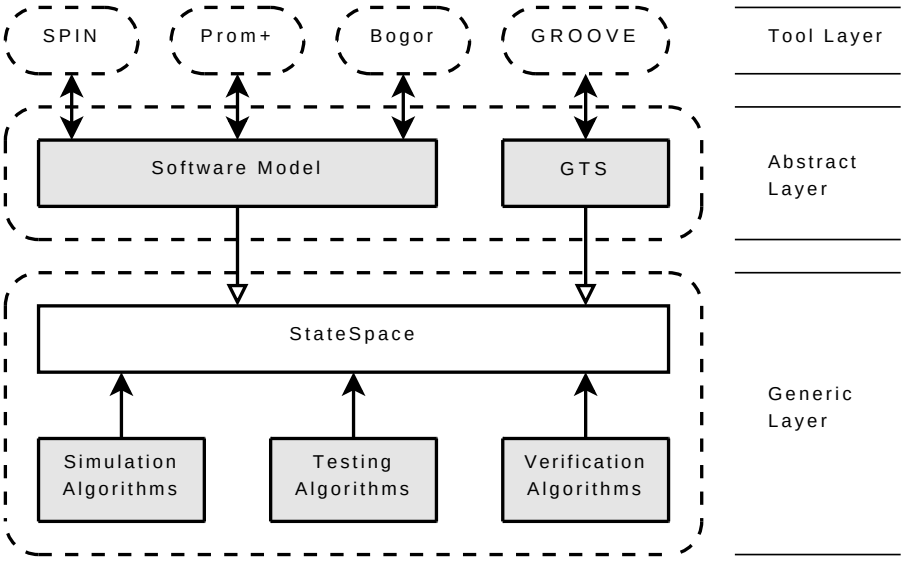

Figure 2.2: An overview of three layers in the framework that is designed by Mark Kattenbelt.

- "The *abstract layer* is the layer on top of the generic layer. In particular it gives an internal structure to the generic layer. [...] The idea is that it is possible to have multiple abstract layers on the same generic layer such that the algorithms of generic layers are reused."
- "The tool layer is not provided by the framework, it is included in the figure to show how tools could use the framework. [...] The nature of the tool layer is that it is not reusable, however, the idea is that a well-defined abstract layer could be used by multiple tools."

Implementation

Mark Kattenbelt has implemented his framework in `C++`, and added in the tool layer the support for the `Prom+` language. It does not use the complete potential of its design, because of the simplicity of the `Prom+` language. Also his benchmarks show that it is much slower than SPIN (by a factor 1000) while using a comparable amount of memory.

Chapter 3

Design of SpinJ

For the design of SpinJ, the main objectives of this thesis must be taken into account. First of all, SpinJ should be *extensible* and *reusable*. This means for example that adding a new search algorithm should be easy and such a algorithm should be readily available for all implementations based on SpinJ. Secondly, SpinJ should remain as *fast* as possible. A magnitude slower than Spin is acceptable, but it must still be practical for model checking.

The first section of this chapter will explain the overall architecture that was chosen for SpinJ. The following sections will then explain the design of the two basic parts of SpinJ: the library (Section 3.2), with the PROMELA implementation on top of it, and the PROMELA compiler (Section 3.3).

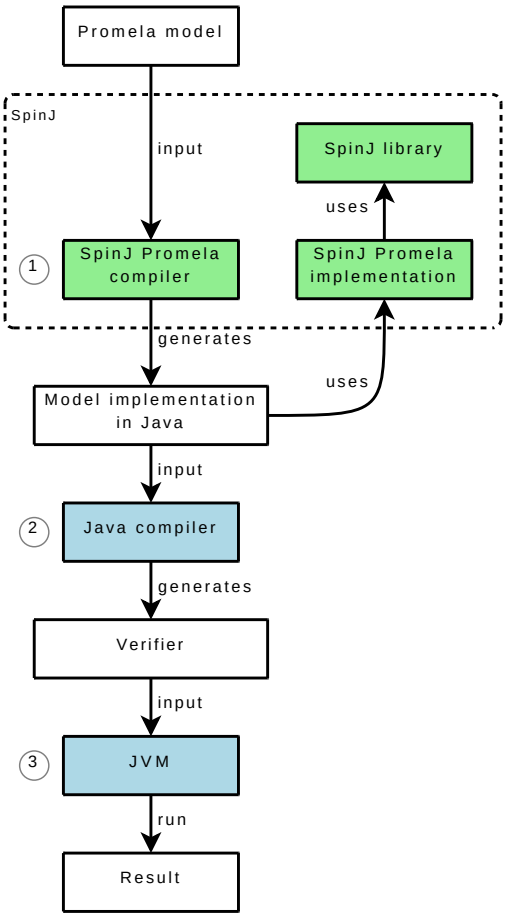

Figure 3.1: The architecture of SpinJ

3.1 Architecture

To verify a model that is written in the PROMELA language, SpinJ follows Spins compilation architecture. Spin first compiles the model to C-code (e.g. `pan.c`, `pan.m`, etc.), which is then compiled to a `pan` executable. This executable file is the verification program.

An alternative solution to verify a PROMELA model would be using an interpreter, although pure interpretation would be slower [38]. Though there are midways between an interpreter and a compiler, in most cases the fastest solution would be the compiler approach.

There is a downside to the architecture of SPIN: all code is generated by the compiler and no code of the verifier can be reused. This is great for optimising the code, but makes changes more complicated and the algorithms that are provided can not be used directly in other projects.

SPINJ tries to take advantage of using generated code like SPIN, while still making use of a shared library. How this works is shown in Figure 3.1. First the compiler generates a Java representation of the PROMELA specification (step 1).

Then the Java compiler is used to compile the code (step 2) using the SPINJ PROMELA implementation, which in turn uses the general SPINJ library. The SPINJ library contains much of the shared code, for example search algorithms and storage methods. This is the main advantage of SPINJ, because this library is designed in such a way that it can be reused. This makes it easier to support other languages: only a basic implementation and a compiler have to be added.

After compilation the verifier can be run on any Java Virtual Machine (step 3), to produce the result. On the command line, the following commands would have to be executed to check the model `Foo`:

```
1 java -jar SpinJ.jar Foo.prom
2 javac -cp SpinJ.jar spinj/generated/FooModel.java
3 java -cp SpinJ.jar spinj.generated.FooModel
```

`SpinJ.jar` contains both the layered library as well as the PROMELA compiler. On the first line that PROMELA compiler from `SpinJ.jar` is called, which does not use the library at all. The second and third line don't use the compiler anymore, but do use the library.

3.2 The layered model

For the design and implementation of SPINJ we have adopted the conceptual framework developed by Mark Kattenbelt et al. [32]. Mark Kattenbelt showed in his Master Thesis [31] that his framework is more flexible than the "black box" approach of many other model checkers. In Figure 3.2 our implementation follows this layered model.

The idea behind these three layers is that each layer extends the layer below it and adds more functionality. This is achieved through inheritance (indicated by the open arrow in Figure 3.2). For example, the concurrent model on the Abstract layer extends the model from the Generic layer. This means that all algorithms that can use a model, can also use any concurrent model. On the abstract layer, however, there are algorithms implemented that can use the fact that a concurrent model contains one or more processes.

This way the different parts of the model checker are loosely coupled. This increases the maintainability and reusability of the whole system.

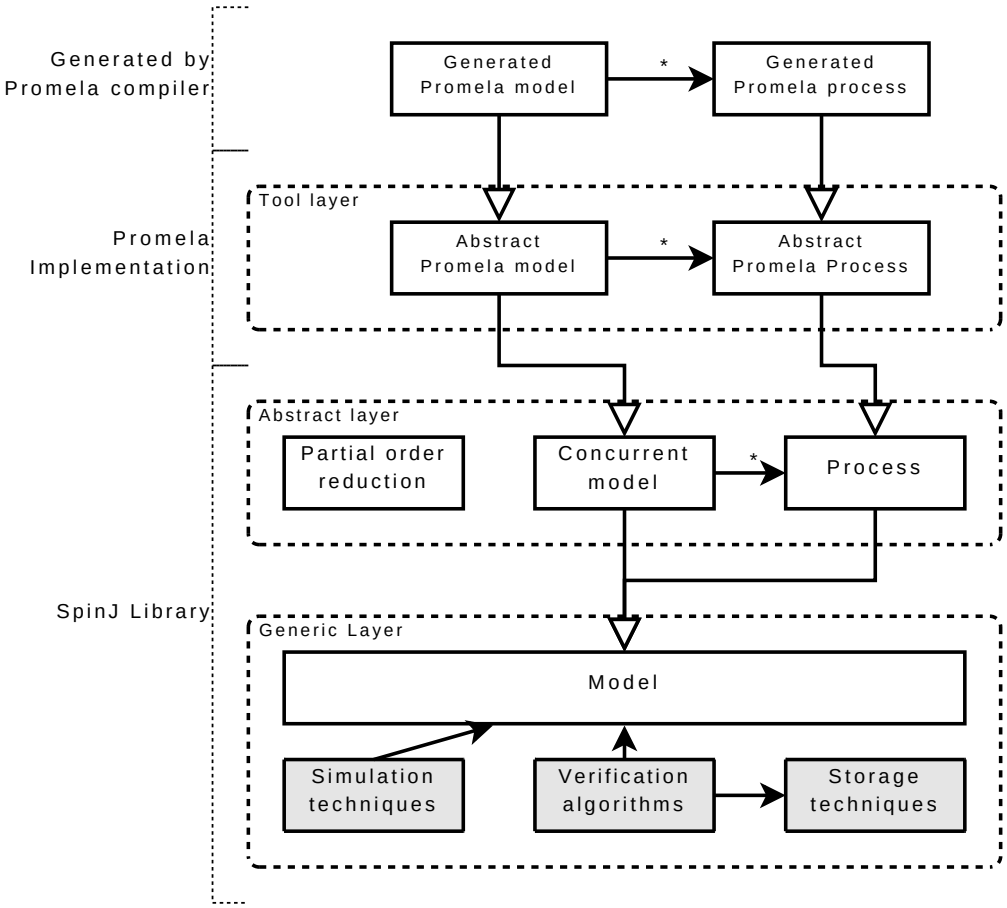

Figure 3.2: The design of the SpinJ library, with on top the PROMELA implementation

3.2.1 Generic layer

The generic layer is the lowest layer, which has a very basic concept of the model. Here we only know that a model consists of states and transitions, not how the model works internally. Despite the fact that we do not know much about the model, we have enough information to implement several important algorithms in this layer. E.g. search algorithms, storage methods and hashing methods.

For each of the different type of algorithm there is an abstract base class that must be implemented (e.g. the `StateStore` and `SearchAlgorithm` classes). So when a new search algorithm will be implemented, it should be done in this layer. In the current version there are enough algorithms already implemented to build a basic state space explorer. Also there are a couple of simulation techniques implemented that can be used. The implementation of this layer is described in more depth in Chapter 4.

3.2.2 Abstract layer

The abstract layer provides us with an extension of the model that was described on the generic layer. It adds the notion of concurrency to the model, where it contains several processes (indicated with the arrow and star in Figure 3.2). These processes can be viewed as models by themselves and the concurrent model generates a Cartesian product of these.

On this level there is also method implemented to optimize the model: Partial Order Reduction. This optimizers can reduce the state space that is generated by the model, by using the fact that the processes are not always depending on each other. For more information see Section 5.2.

The implementation of the abstract layer is described in Chapter 5.

3.2.3 The PROMELA Tool layer

Within the tool layer an abstract model is defined that makes it easier to generate Java code from the PROMELA compiler. The abstract PROMELA model is again an extension of the concurrent model from the abstract layer. Also some extra helper objects are implemented here (e.g. the `Channel` object, for easy simulation of channels).

On top of this layer the generated PROMELA models are placed, as an extension of the abstract PROMELA model. These generated models are the product of the PROMELA compiler as described in Section 6.3. The implementation of the tool layer is described in Section 6.2.

It is possible to adapt SpinJ to support different languages by creating a new implementation of this layer and of the compiler.

3.3 The PROMELA Compiler

The SpinJ PROMELA compiler is responsible for translating the PROMELA specification to the generated PROMELA Java model from Figure 3.2, which can be used with the SpinJ library to verify the model. In Figure 3.3 we can see this process in three steps.

The first step is translating the original PROMELA model to an intermediate representation. This is done using a PROMELA parser. This parser is generated using the JavaCC (Java Compiler Compiler) tool. This tool reads a grammar file and converts it to the Java code that makes up the parser.

When this first step is done, we have an intermediate representation where the `Specification` object holds all the information. This object contains one or more `Proctype` objects (the PROMELA processes), which in turn holds one `Automaton`. An `Automaton` object holds all information concerning the behaviour of the process, e.g. the possible states and transitions.

The second step optimises the `Automaton` objects to generate a more efficient automaton, which usually contains less states. This is done using one or more `Automaton` optimizers.

Finally the Java code is generated that makes up the final generated PROMELA model. This code can be compiled and together with the SpinJ

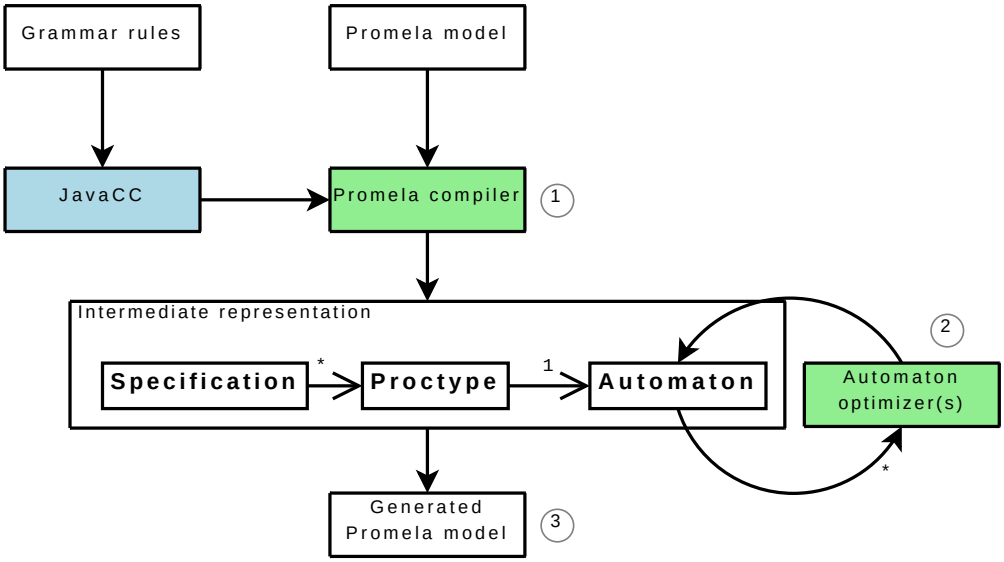

Figure 3.3: The design of the SpinJ PROMELA compiler

library run as a complete model checker.

See Chapter 6 for more details on the PROMELA Compiler.

Chapter 4

Generic layer

This chapter will explain how the Generic layer of the SpinJ library is designed and implemented. Recall that the Generic Layer has a very basic concept of the model (see Figure 3.2). The different parts that will be discussed here are displayed in the UML class diagram of Figure 4.1. As can be seen, this layer contains several methods and algorithms. We can split these into the following **categories**, with corresponding *package names*:

- The **model description** (*spinj.model*). This provides some abstract classes that have to be implemented by any model, to be verifiable by SpinJ. A `Model` object represents here a means to generate a state space on-the-fly. On this layer not much more is known than that it consists of states and transitions. This is further explained in Section 4.1.
- The **simulation methods** and **search algorithms** (*spinj.search*). There are several methods available to simulate a given model, which are described in Section 4.2.1. Also a couple of search algorithms are implemented to search trough the state space that can be generated by the model. These are described in Section 4.2.
- The **storage methods** (*spinj.store*). These are used to store the states that have been found during the execution of one of the search algorithms. This is needed, because it is more efficient when encountering duplicate states. Now when a state is reached that was stored before, the search algorithm can skip it. The different storage methods are described in Section 4.3.
- The **hashing algorithms** (*spinj.store.hash*). Many of the storage methods use a hashing algorithm to efficiently store and find states. The implementation of one of these methods is described in Section 4.4.

4.1 Model description

As described before, a `Model` object is responsible of generating all the possible states at runtime. This means that it is not known which states may be reached, or even how many there exist.

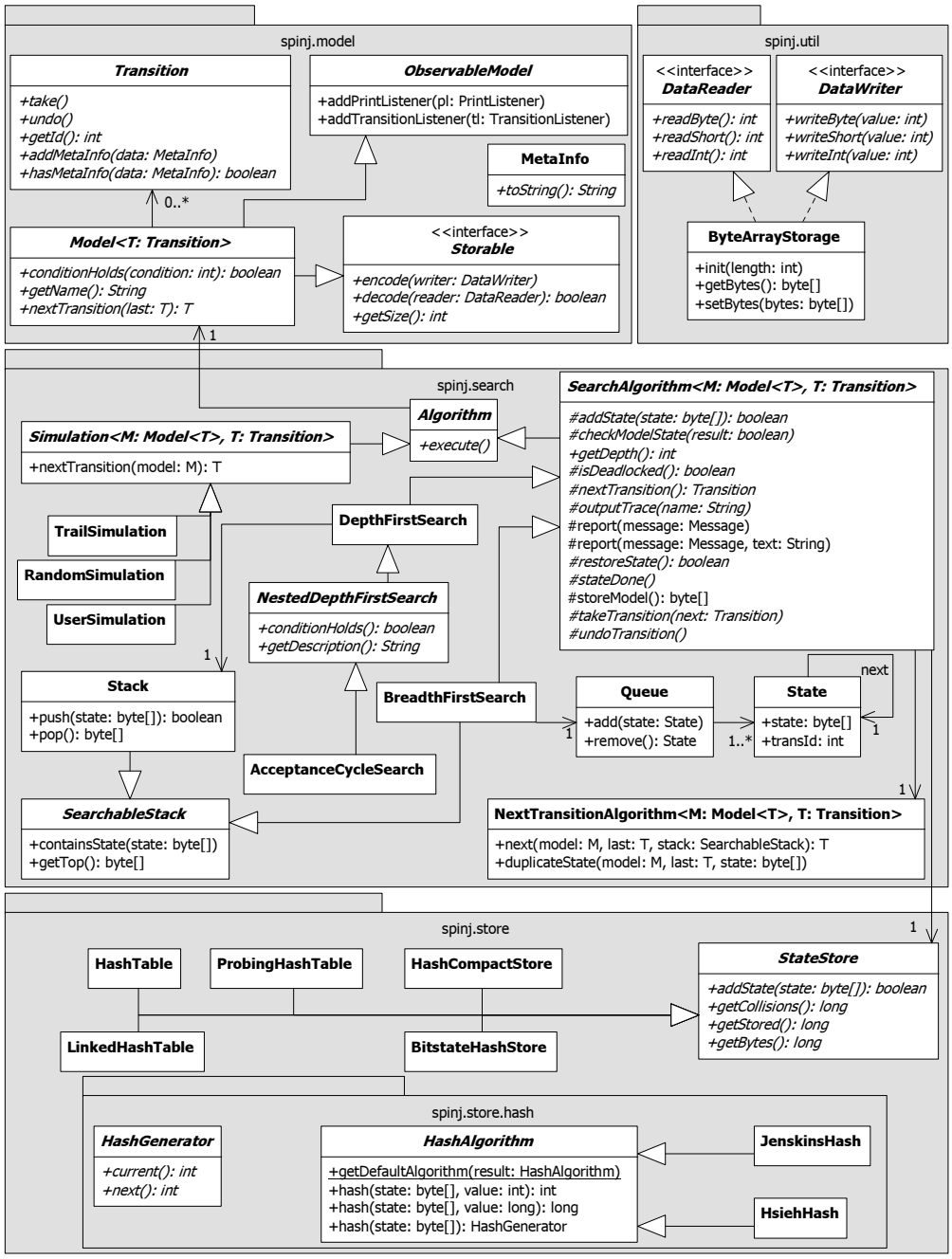

Figure 4.1: An UML diagram that describes the Generic layer of SpinJ

To create the states at runtime, a starting state should be known and it should be known how to take a transition from one state to the next. When that transition is taken, the current state of the model is changed. To achieve this functionality, the following function should be implemented by each `Model` object:

- The `nextTransition` method: Given the current (implicit) state of the model and a previous taken transition from that state, it can generate the next transition. This method can be used to compute all the transitions that are enabled in the current state. More details on transitions will be explained in Section 4.1.2.
- The `encode` method: Encodes the current state of the model in an efficient way. More details will be explained in Section 4.1.1.
- The `decode` method: Decodes the encoded data to restore a state that was stored using the `encode` method.
- The `conditionHolds` method: Checks whether a given condition holds. The conditions that are provided by default are:
  - Whether the current state should be *stored*, because there are cases when a state should not be stored (e.g. in a atomic sequence in Promela, see Section 6.2.5).
  - Whether the current state is an *end state*. This is used when checking for a deadlock. When encountering a state where there are no more transitions that can be taken, we are either in an end state (all activity has ended normally) or in a deadlock situation.
  - Whether the current state is in an *acceptance state*. This is used when searching for acceptance cycles [25]. See Section 4.2.4.
- The `getName` method: This method is used to produce a meaningful name for this model.
- The methods from the `ObservableModel` can be used to add listeners to the model. These listeners can listen for two types of events: print events or taking/undoing a transition. This is further explained in Section 4.1.3.

4.1.1 Encoding states

To store the states that have been visited already, there are two options: storing the state vector in a specialized object or encoding all data to an array of primitive data (e.g. a byte array). The advantage of the first is that it is relatively simple to implement and it makes returning to a previous state easy.

The advantage of encoding the data to a primitive array is that it uses less memory, because of the overhead that objects have in Java. For example, when we have a `Model` object with 5 processes in it, at least 68 extra bytes are used for each state (4 bytes per reference to a Process object, plus 8 bytes extra for

each Java object). Since the state vector of most models is less than 100 bytes, this is a significant overhead.

Furthermore implementing a hash function on a simple array of numbers is easier, than to create an approximate hash function for each generated `Model`.

The `Storable` interface To encode the states, the `Model` implements the `Storable` interface (see Figure 4.1). This interface makes it possible to encode the current state of the model into a more primitive format (e.g. a byte array). The simplest solution to this would be to have a `store` function that returns a new byte array that represents the current state, but this solution is not perfect. The problem lies in the performance, because many optimizations are not possible.

Having a `encode` function that can encode the state using the `DataWriter`, is more flexible. To make this work a `getSize` method is also available, to indicate how many bytes need to be available to store it.

To see what exactly the advantage of this method is, let us evaluate the next example:

```
1  public int getSize() {
2    // The model needs to store the number of processes
3    int size = 1;
4    for(Process proc : processes) {
5      // Add the size of each process
6      size += proc.getSize();
7    }
8    return size;
9  }
10
11 public void encode(DataWriter writer) {
12   // First encode the number of processes
13   writer.writeByte(nrProcs);
14   for(Process proc : processes) {
15     // Encode each process in the buffer
16     proc.encode(writer);
17   }
18 }
```

This example shows the (simplified) implementation of the `encode` function of a model with a few processes in it as defined in SpinJ. To encode all the processes, the `Process` class also implements the Storable interface. Now we can simply call the `encode` function for each process in the model (and the same for the `getSize` method).

If we had used a `store` function that simply returns a byte array, each call to the `store` function of each process would have returned a temporary object. This puts more pressure on the garbage collector. The current design handles this more efficiently.

Furthermore, the `Storable` object itself does not have to know exactly how the data is stored. The `DataWriter` object could write it to a byte array, an integer array or even a file.

4.1.2 Transitions

The set of transitions that can be executed from a certain state is called the *enabled set*. To retrieve the enabled set from the model, the `nextTransition` function is defined in the model.

This function does not return the enabled set directly, as one might expect, because of efficiency reasons. To store such a set, a temporary storage object is needed. Furthermore, not all enabled transitions are needed directly when using Depth First Search (which is the default search algorithm in SPIN and in SPINJ).

Instead, only one transition is returned each time the `nextTransition` function is called. To retrieve any further transitions that may be executed, the previous transition is provided as a parameter, until the function returns null, indicating that there are no more transitions. This resembles the iterator and enumeration functionality found in the Java class library.

The `Transition` class All transitions that are returned from the model, implement the `Transition` class. This class describes the following functionality:

- The `take` function executes the transition, changing the state of the model to the next state.
- The `undo` function undoes all changes that are made by the `take` function. This function is called when we want to return to a previous state.
- The `getId` function returns a unique identifier that can be used to create a trail file, which stores the path to a certain state (see Section 4.2.6).

As can be seen, each transition has a mandatory `undo` action. Because many transitions change only a very small part of the state vector, it is often more efficient to undo the transition than to decode the complete state vector.

4.1.3 Observable Model

There are two aspects of the model for which the observer pattern [15] is used: changing the state of the model and performing a print action. For both actions a separate interface is defined: the `TransitionListener` and the `PrintListener`.

To ease the usage of this pattern, each `Model` extends the `ObservableModel` class. This class contains all the functionality to add listeners, notify them and disable or enable them (also described in Figure 4.2).

Now each time a transition is taken in a model, the `sendTransactionTakeEvent` is called to send a `TransitionEvent` to all the `TransitionListener`s. The same goes for all the `PrintListener`s when the `sendPrintEvent` function is called, which is only used to send text from the model to the console.

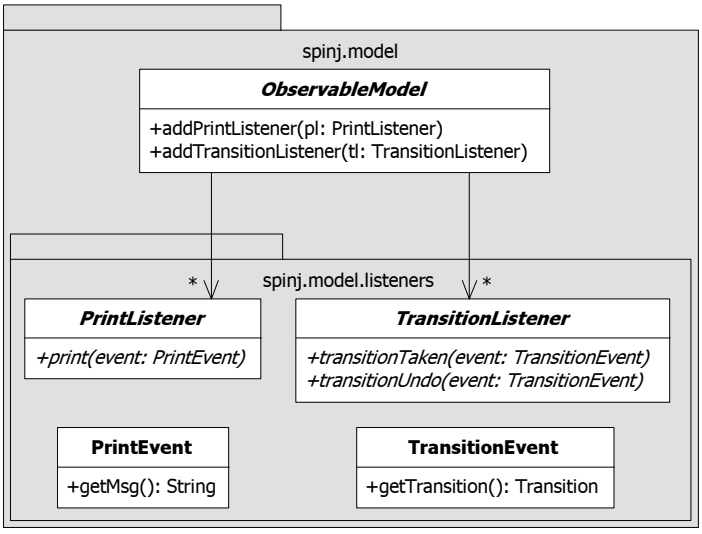

Figure 4.2: An UML diagram that describes the `ObservableModel`

4.2 Exploration algorithms

A crucial part of a model checker are the exploration algorithm(s) that are supported. In SpinJ the goal is to make this part extendable, such that other exploration algorithms can be added in the future. In the current version there are several different implementations, which can be split in two categories: search algorithms, which explore the complete state space, and simulation algorithms, which executes a single path through the state space.

4.2.1 Simulation

To see if a given model behaves as expected, the author of that model will usually perform a couple of test runs. This is done in the simulation mode. In simulation mode there is a single path through the state space chosen to be executed.

The `Simulation` class uses generics to keep track of the type of `Model` (`M`) and `Transition` (`T`) it is executing. This type safety is needed for calculating the next transition, otherwise a (unsafe) cast must be used. Simulation of a model is a simple procedure, as can be seen in the following code example from the `Simulation.execute` method:

```
1  public void execute() {
2    while (true) {
3      try {
4        // Choose the next transition that we are going to execute
5        final T t = nextTransition(model);
6        if (t == null) { // No next transition, quit
7          break;
8        }
9        out.println("Taking transition: " + t.toString());
10       t.take(); // Discard the backup object, we don't need it
```

```
11          states++; // Increase the number of states that were visited
12        } catch (final ValidationException ex) {
13          out.println(ex.getMessage());
14          break;
15        }
16      }
17    }
```

When a simulation is executed, it chooses one transition to execute for each step. The choice is done in the `nextTransition` method. Then it takes that transition.

This loop is executed until either the `nextTransition` function returns `null` (indicating to end the run), or a `ValidationException` is thrown while taking the next transition (indicating an error in the model).

The current version provides three different implementations of `Simulation`:

- `UserSimulation`, which asks the user for choosing the transition to be taken.
- `RandomSimulation`, which randomly chooses the transition from the enabled set, until the enabled set is empty.
- `TrailSimulation`, which reads a *trail file* (see Section 4.2.6) and chooses the transition with the same identifier (returned by the `Transition.getId` method).

4.2.2 Abstract Search algorithm

The implementation of all the search algorithms currently in SpinJ are based on an abstract search algorithm, which contains some basic code that is the same for every search algorithm.

The difference between different search algorithms is the order in which the states are explored (e.g. a stack or queue of waiting states) and if some states may or may not be explored further (e.g. with dynamic partial order reduction [14] we can optimize the state space). The common denominator between all the possible search algorithms can be described in the following three basic steps:

- Retrieving a not yet fully expanded state to expand the search area.
- Checking the retrieved state for errors (e.g. deadlocks).
- Executing one transition from the current state that has not been executed yet, resulting in a new state. When this new state was already visited before, it is discarded. Otherwise it is stored and put in the collection of states that need to be expanded further.

In the implementation of the steps in the abstract search algorithm, checking for errors and executing a transition will be done at the same time. This is done because many checks will be executed automatically by the transition

itself (e.g. assertion checks). A simplified version of the `execute` method in the `SearchAlgorithm` class is given below.

```
1  public void execute() {
2    byte[] state = storeModel();
3    addState(state); // Adds the initial state to the search algorithm
4    if(model.conditionHolds(SHOULD_STORE_STATE)) {
5      store.addState(state);
6    }
7
8    while(nrErrors < maxErrors && restoreState()) {
9      // Make sure that the state matches the internal state of the model
10     assert checkModelState(model)
11
12     // Retrieve the next transition
13     final Transition next = nextTransition();
14
15     if(next == null) {
16       if (checkForDeadlocks && !model.conditionHolds(Condition.END_STATE)
17           && getLastTransition() == null) {
18         report(DEADLOCK);
19       } else {
20         report(NO_MORE_TRANSITIONS);
21       }
22       stateDone(); // Marks this state as fully explored
23       continue;
24     }
25
26     try { // Take the next transition and check for errors
27       takeTransition(next);
28     } catch(final SpinJException ex) {
29       report(TRANS_ERROR, ex.getMessage());
30       continue;
31     }
32
33     state = storeModel(); // Private method that creates a new byte[]
34                           // from the current state of the model.
35
36     // If the state should be stored, try to store it
37     if(model.conditionHolds(SHOULD_STORE_STATE)) {
38       if(!store.addState(state)) {
39         report(DUPLICATE_STATE);
40         undoTransition();
41         continue;
42       }
43     }
44
45     // Add the state to the stack/queue/...
46     if(!addState(state)) {
47       report(EXCEED_DEPTH_ERROR);
48       undoTransition();
49       continue;
50     }
```

```
51    }
52  }
```

As can be seen in the previous code example, each iteration first restores a state (using the `restoreState` function on line 8). This makes sure that the current internal state of the `Model` object is one that needs to be expanded further. Then it tries to execute a transition, to generate a new state. In the meantime some conditions are checked to see if anything is going wrong with the model.

Many of the function calls that are made in the example code, are to abstract functions of the `SearchAlgorithm` class (e.g. see table 4.1, 4.2 and Figure 4.1). These functions must be implemented by the search algorithm to define the behaviour.

4.2.3 Depth First Search

Depth First Search (DFS) is the first search algorithm that is implemented in SpinJ, which is also the default search algorithm that is used in Spin. This algorithm is implemented using a stack to store all states that still need to be expanded further.

The `Stack` class that is implemented for the DFS stores the following information for each state that is pushed onto the stack:

- The encoded *state*. This is needed to check if a state is already on the stack (e.g. while performing Partial Order Reduction or Nested Depth First search).
- The *last transition* that has been executed from the state with which it is stored. This information is needed to retrieve the next transition that may be executable from this state and to undo this transition.

Table 4.1 shows how the abstract functions of the `SearchAlgorithm` are implemented for DFS.

4.2.4 Nested Depth First Search

For the cycle detection algorithm that is implemented in SpinJ (acceptance cycle detection), a nested depth first search is implemented. This algorithm is implemented as an extension of `DepthFirstSearch`. It is based on the algorithm as described on page 180 of [21].

There are two extra variables that need to be tracked during a nested depth first search:

- `toggle`; to indicate that we are in the first or second search.
- `seed`; that holds the state that initiated the nested search.

To make the nested search work, three parts have changed. First the `toggle` is encoded (and decoded) with the state. This is done in the `storeModel` and

`addState`	Adds the given state on top of the stack.
`checkModelState`	Checks if the internal state of the model is the same as the encoded state that is on top of the stack.
`nextTransition`	Calculates the next transition that is to be executed using the `NextTransition` object and the last executed transition (that is on top of the stack).
`restoreState`	Since restoring of a state is implicit in Depth First Search, it does not restore any state. It does return true when there are states left to be searched (e.g. the stack is not empty).
`stateDone`	Removes the top of the stack (the current state) and undoes the transition that was stored one below the top of the stack. This method returns the model to a previous state.
`takeTransition`	Executes the given transition and stores the transition on top of the stack, next to the encoded state.
`undoTransition`	Undoes the transition that was stored on top of the stack, returning the model into the previous state.

Table 4.1: The implementation description of the Depth First Search

`restoreModel` methods. This is not the most efficient method and a better implementation is described on page 189 of [21]. There it is explained how each state needs only 1 extra bit to indicate if the nested search has encountered this state. This method is very hard to implement in the current design, because this would require some extra functionality of a `StateStore` to store extra metadata with each state.

Secondly the `nextTransition` can return an extra transition when `conditionHolds` returns true. When the model is in such a state where a nested DFS needs to be started, this method will first return all normal transitions, and then the transition that starts the nested search. This special transition sets the `toggle` to `true` and encodes the current state of the model into the `seed`.

The third change is to the `takeTransition` function, which is responsible for executing the transition. If we are in the nested search (`toggle` is set to true), then some extra conditions are checked to see if a cycle has been detected. This occurs when we are in a state that is equal to the `seed` or if the state was already on the stack before the `seed`.

Acceptance cycle detection

Acceptance states are used to look for acceptance cycles in a state space. This is useful when checking for lifeness properties, which deals with infinite runs.

"An acceptance cycle in the reachability graph of automaton A exists if and only if two conditions are met. First, at least one accepting state is reachable from the initial state of the automaton $A.s_0$. Second, at least one of those accepting states is reachable from itself." (page 178 of [21])

The acceptance cycle detection is implemented by the `AcceptanceCycleSearch`, which extends the `NestedDepthFirstSearch`. It implements the `conditionHolds` method by checking if the current state of the model is an acceptance state.

4.2.5 Breadth First Search

Breadth First Search (BFS) is an alternative search algorithm that is implemented, which expands each state fully and per level of depth. This method is implemented using a `Queue` (see Figure 4.1) to store the states that have to be expanded further. The `Queue` is implemented using a linked list of `State` objects. Each of these `State` objects remembers the following values:

- The *next state*, which is needed for the single-linked list, which is how these `State`s are stored in the `Queue`.
- The *previous state*, which references to the previous state in the executing path (not a reference for the linked list).
- The *encoded state*, to be able to implement the `SearchableStack` interface (implementing the `containsState` method).
- The *Transition identifier*, to be able to write a trail-file when necessary.

The abstract functions of the `SearchAlgorithm` that are implemented for BFS are shown in table 4.2. The Breadth First Search keeps track of the current state that is to be expanded, named `fromState`.

`addState`	Adds the given state at the end of the queue.
`checkModelState`	Checks if the internal state of the model is the same as the encoded state that is currently executing.
`nextTransition`	Calculates the next transition that is to be executed using the `NextTransition` object and the last executed transition.
`restoreState`	When the current `fromState` is null, it retrieves a new state from the head of the queue. When that queue is already empty, it returns false. When there still was a `fromState` it undoes the last transition, to return to that state.
`stateDone`	Sets the `fromState` to `null`, indicating that the current state is done and that a new one should be retrieved from the queue.
`takeTransition`	Executes the given transition and stores the transition as the last executed one.
`undoTransition`	Does nothing. If there is a transition to be undone, the `restoreState` will undo it.

Table 4.2: The implementation description of the Breadth First Search

4.2.6 The trail file

The trail file is created when the search algorithm finds an error. The file contains a list of identifier numbers to indicate which transition has been executed in which order. With these number the `TrailSimulation` can easily execute that run again, to see in detail what went wrong.

Unfortunately, this trail file is not compatible with the trail files that are generated by SPIN because the numbering of the transitions is done differently.

4.3 Storage methods

To avoid expanding states that were already found during the search, states are stored using a `StateStore`. The `StateStore`'s only responsibility : *Has this state been visited before?* To answer this question, any implementation of the `StateStore` implements the following methods:

- `addState` tries to add the state to the store, but can return false when the state has already been added.
- `getStored` returns the number of states that have been stored successfully.
- `getBytes` returns the number of bytes that is estimated to be used by this storage facility.

The different storage methods can be split up in two categories: the exhaustive storage and approximate storage. Exhaustive storage will always provide the correct answer to the question stated above, but approximate can sometimes give a wrong answer. The can either be a false positive or a false negative.

A false negative occurs when a state was said to be visited before, while it has not been. When a storage method returns false negatives, some parts of the state space may never be reached. And while the state space was not completely checked, it can never be guaranteed that there is no error in the model.

A false positive occurs when a state was not visited before, while it has been. When a storage method returns false positive it possibly visits more states than necessary, thus slowing down the search algorithm.

The following sections will describe the different storage methods that are implemented in SPINJ.

4.3.1 Hashtable

A hash table [18] is an exhaustive storage method that uses a set of "buckets", which each can hold one or more states. A hash function is used - given a state as a key - to determine the identifier of the bucket where the state will be stored in.

Hashtables have the advantage of being very fast in finding and adding states (average complexity is $O(1)$), but the difficulty and overhead comes when dealing with collisions. Three different methods are implemented in SPINJ.

Linked lists are used in the `LinkedHashTable`. Here every bucket is essentially a linked list (similar to the `LinkedList` provided by the Java API), which is used to resolve any collisions. When two states both fall into the same bucket, they are stored in that linked list.

A disadvantage is that each item in the linked list requires an extra object, which costs 16 bytes of memory each. Also some CPU time is wasted, for creating the extra objects and traversing a linked list.

Arrays are used in the `HashTable` class. In this implementation each bucket is an array of states (similar to the `ArrayList` provided by the Java API). Here we still have the advantage that handling collisions is simply adding a new state to that list, plus that the memory overhead is a lot smaller (only one extra object for each bucket).

When the array becomes too small we need to create a new array and copy all the items from the old array. This is still faster while we are adding states to the end of the list (as shown in [30]) than using a linked list.

Probing is an alternative solution to dealing with collisions. This method is implemented in the `ProbingHashTable`. Every bucket in this hashtable contains simply one state and collisions are solved by finding a new bucket for that state. This removes almost all memory overhead of using lists (no extra objects used to store the bucket).

To find a next bucket, a new hash value must be calculated. This is done by rehashing, which is explained in more detail in Section 4.4.

The problem with a probing hash table is when the table becomes full, because then there will be a lot of collisions. In the current implementation this is solved having an overflow table, which gets used when an empty bucket could not be found.

4.3.2 Bitstate hashing

Bitstate storage is one of the first approximations for state storage that was added to SPIN [20]. It works by storing only a variable number (k) of bits per state instead of storing the complete state. To determine which bits need to be set, k (preferably independent) hash functions are used to calculate k values. Each of these values then corresponds to an address in a table of bits.

To check if a state was already stored, all the values are calculated and those bits in the table are checked. Only when all bits are set, the model checker can conclude that the given state was visited before. This method can return a false negative when all the bits were already set by one or more other visited states that returned these hash values.

The implementation in SPINJ provides a `BitstateHashStore` class, which implements this algorithm using an array of long values (64-bits each). This way we can create a maximum array of 16 Gigabytes (64 bits * 2^{31} entries). Also, instead of using independent hash functions, rehashing is used (see Section 4.4). As shown in [12] this method seems to be almost as good and can be calculated much faster.

4.3.3 Hash compaction

Hash compaction [40] is a storage method that is also an approximation. It is based on the idea that bitstate storage needs a very large table to work effectively (see [33]), so it simulates a very large table by storing the addresses of the bits that would be set in a bitstate table. These addresses can be stored in a normal hash table for efficient handling.

The implementation of the improved hash compaction algorithm (from [33]) in SpinJ is very straightforward. It uses a probing hash table to store the hash values instead of the states. First the value that needs to be stored is calculated by the first hash function. Then a second hash function is used to calculate the index of the bucket where the value will be stored.

The limitation of the current implementation of hash compaction is that it only supports table sizes that are a power of 2. Some optimisations from [12] are yet to be implemented, to make it possible to use tables of any size.

4.4 Hashing methods

A good hash function is important for all the storage methods that are described above. As Bob Jenkins describes on his website [28]: "A good hash function distributes hash values uniformly."

A second requirement is that a good hash function must be as fast as possible. For example, MD4 is an excellent hash function with good distribution, but is much slower than some other functions.

Any class that implements the `HashAlgorithm` interface can be used as a hash function. The `HashAlgorithm` interface declares a couple of functions to achieve this:

- The `hash(byte[], int)` method takes a state (a `byte[]`) and an initial value (an `int`) as parameters and calculates a 32-bits hash value for it.
- The `hash(byte[], long)` method takes a state (a `byte[]`) and an initial value (an `long`) as parameters and calculates a 64-bits hash value for it.
- The `hash(byte[])` method takes a state (a `byte[]`) and returns a `HashGenerator` object. This object makes it possible to generate a series of 32-bits hash values (for example needed with a probing hash table).

The current version of SpinJ provides two implementations of the `HashAlgorithm`.

4.4.1 Jenkins hash

On his website [28], Bob Jenkins describes a very fast and good hash function, which is also used in Spin. The `JenkinsHash` class implements this algorithm. Also the rehashing optimisations from [12] are implemented here, but SpinJ still has the limitation of only handling tables with a power-of-two granularity.

4.4.2 Hsieh hash

On his website [26], Paul Hsieh describes a method that is even faster than the one of Bob Jenkins describes and distributes the hash values almost as well. The problem with this method is that it is optimised for C code, where it can be compiled to only 13 assembly instructions per cycle. For this reason this new hash function is the default in SPIN since version 5.

The `HsiehHash` class implements this algorithm. Some small tests with a real model have shown that this optimizations is not faster when implemented in Java than the Jenkins hash. What the reason for this is, is left for future research.

Chapter 5

Abstract layer

As discussed in Section 3.2.2 the abstract layer provides an extension of the generic layer with the notion of concurrency. The concurrency within the model is based on the definition of one or more processes (also see Section 3.2.2). Each of these processes can be seen as models by themselves, and the concurrent model as created by the Cartesian product of these.

In Figure 5.1 we can see the UML diagram that describes the relation between the different parts of this Abstract layer and how these are related to the Generic layer. The relation is mainly established through inheritance. Since the `ConcurrentModel` is an extension of the `Model` class, all algorithms that were defined in the Generic layer can just as easily be applied to any `ConcurrentModel`.

5.1 Concurrent model

The `ConcurrentModel` has all the functionality of the `Model` class. To accommodate the processes within a concurrent model, it defines two extra functions:

- `getNrProcesses()` returns the number of processes that are currently running inside the `ConcurrentModel`.
- `getProcess(index)` returns the process that is located on the given index (comparable with a list).

5.1.1 Processes

The processes that are running inside a `ConcurrentModel` are implementations of the `Process` class. Since every `Process` is an extension of the `Model` class, every process also can, e.g., determine the next transition (of that specific process) or check if a condition holds etc.

Next to this basic functionality, there are also two extra functions:

- `getId()` returns the unique identifier of the process within the model. This identifier is also the index that can be used in the `getProcess()` method of the `ConcurrentModel`.

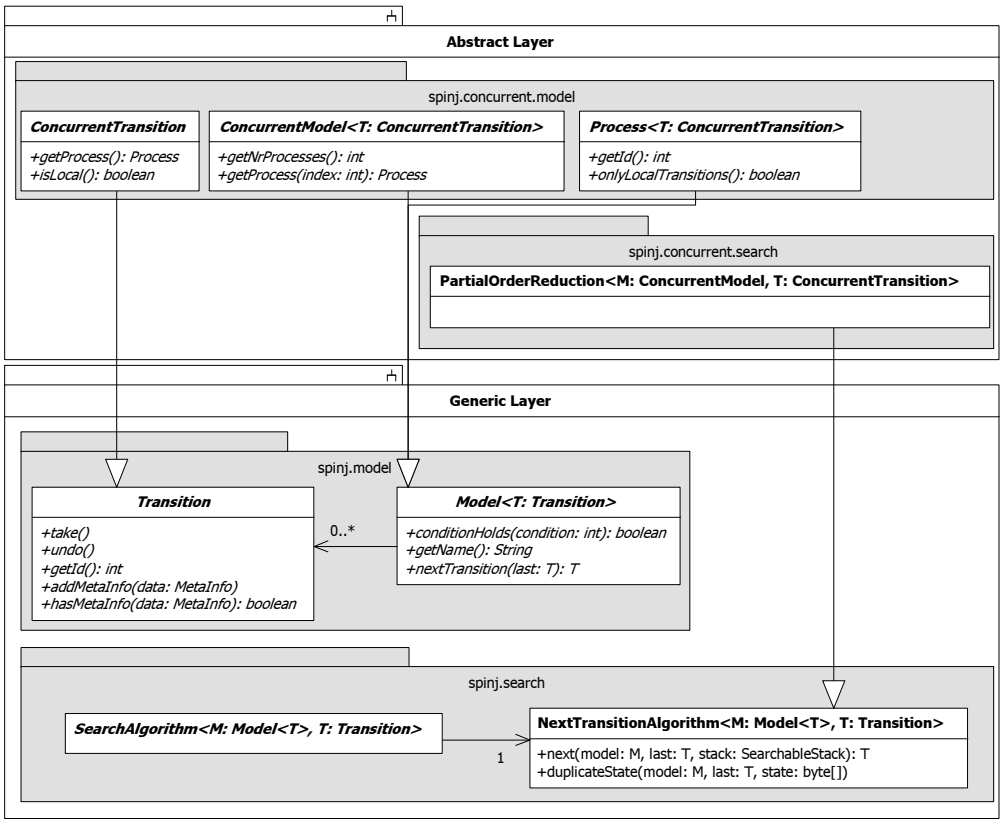

Figure 5.1: An UML diagram that describes the Abstract layer of SpinJ on top of (part of) the Generic layer

- `onlyLocalTransitions()` returns true when all the current transitions that may be executed from the current state of this process, only have local effects. This method can be used to limit the state space that has to be searched (see also Section 5.2).

5.1.2 Concurrent Transitions

The transitions that can be returned by `Process.nextTransition()` (and therefore also by the `ConcurrentModel.nextTransition()`) are always implementations of the `ConcurrentTransition`. This extension of the `Transition` class defines two extra functions:

- `getProcess()` returns the `Process` to which this transition belongs.
- `isLocal()` returns true when this transition only has local effects (e.g. not affecting other processes in the concurrent model).

These concurrent transitions can be used by optimisations techniques that require the knowledge of processes and their transitions (e.g. Partial Order Reduction, see Section 5.2).

5.2 Partial Order Reduction

Partial order reduction [16] [13] is aimed at reducing the size of the total state space that needs to be generated. This algorithm changes the way the search algorithm can select the transitions that can be executed from a certain state. Instead of selecting all transitions that are *enabled* from the current state, we can choose a subset of that. So from a certain state s, we calculate $ample(s) \subseteq enabled(s)$, which will result in a reduced state space.

To determine the $ample(s)$, the similar heuristic is implemented as the one in [16]. First it returns only transitions from processes, for which all possible transitions only have a local effect. The conditions for when a transition only has a local effect is listed in [16], chapter 5. If at least one transition was found and no state was generated that was already on the stack, the algorithm ends (not returning any more transitions from other processes).

Partial order reduction is implemented in SpinJ as an extention of the `NextTransition` class, which is shown in the following listing.

```
1  public class PartialOrderReduction<M extends ConcurrentModel<T>, T
       extends ConcurrentTransition<T>>
2    extends TransitionCalculator<M, T> {
3
4    public T next(final M model, T last) {
5      int i = model.getNrProcesses() - 1;
6      if (last != null) {
7        Process<T> lastProc = last.getProcess();
8        if (lastProc == null) {
9          return null;
10       }
```

```
11
12        T next = lastProc.nextTransition(last);
13        if (next != null) {
14          next.copyMetaInfo(last);
15          return next;
16        } else if (last.hasMetaInfo(MetaInfo.PartialOrderReduction)
17            && !last.hasMetaInfo(MetaInfo.StateOnStack)) {
18          return null;
19        }
20
21        i = last.getProcess().getId() - 1;
22      }
23
24      if (last == null || last.getProcess().onlyLocalTransitions()) {
25        while (i >= 0) {
26          final Process<T> proc = model.getProcess(i--);
27          if (proc.onlyLocalTransitions()) {
28            T next = proc.nextTransition(null);
29            if (next != null) {
30              next.setMetaInfo(MetaInfo.PartialOrderReduction);
31              return next;
32            }
33          }
34        }
35        i = model.getNrProcesses() - 1;
36      }
37
38      while (i >= 0) {
39        final Process<T> proc = model.getProcess(i--);
40        if (!proc.onlyLocalTransitions()) {
41          T next = proc.nextTransition(null);
42          if (next != null) {
43            return next;
44          }
45        }
46      }
47
48      return null;
49    }
50
51    public void duplicateState(final M model, final T last, final byte[]
        state,
52      final SearchableStack stack) {
53      if (last.hasMetaInfo(MetaInfo.PartialOrderReduction)) {
54        if (stack.containsState(state)) {
55          last.setMetaInfo(MetaInfo.StateOnStack);
56        }
57      }
58    }
59  }
```

As we can see, the `next` function tries to first to return transitions from processes with only local transitions. These transitions are marked with the

`PartialOrderReduction` meta info. When a state gets found that was visited before, then the `duplicateState` function is called. This function checks if the last transition was a local one (by seeing if it has the `PartialOrderReduction` meta info) and if the state was also on the stack. If this is the case, then the last transition also gets marked with the `StateOnStack` meta info.

The implementation of the `next` function to find the local processes consist of three parts. The first part on lines 6 to 22 determines if there was a previous transition, in which case we can try to find more transitions from that process. When there are no more transitions, then it checks if POR was used and there was no duplicate state on the stack. In that case it can return prematurely.

The second part (on lines 24 to 36) is when a next process must be found that is local. To see if a process is local, its `onlyLocalTransitions` function is called. The processes are scanned from the last on to the first, to mirror exactly how Spin executes its transitions.

The third part (on lines 38 to 47) is the same as the second, only it is looking for non-local processes.

Chapter 6

Implementation of the PROMELA language

As shown earlier in figures 3.2 and 3.3, the implementation of the PROMELA language consists of two parts: the PROMELA compiler and the abstract PROMELA model. The PROMELA compiler is responsible for generating the PROMELA models that are implementations of the abstract PROMELA model.

This chapter is going to explain these different parts. First it discussed the PROMELA language itself and how the compiler parses this language into an intermediate representation.

Then the second part will explain the tool layer, which contains abstract classes for a PROMELA model, processes and transitions. Also there it will be explained what Java code is generated for the intermediate representation and how this relates to the tool layer.

The final section of the chapter will discuss some problems that were encountered while implementing the PROMELA language in Java. These problems are mostly caused by the differences between the C programming language and Java.

6.1 PROMELA

This section explains the PROMELA language and how the compiler parses that language into the intermediate representation.

6.1.1 The language

From [1]: "PROMELA is a verification modelling language. It provides a vehicle for making abstractions of protocols (or distributed systems in general) that suppress details that are unrelated to process interaction. The intended use of SPIN is to verify fractions of process behaviour, that for one reason or another are considered suspect. The relevant behaviour is modelled in PROMELA and verified. A complete verification is therefore typically performed in a series of steps, with the construction of increasingly detailed PROMELA models. Each model can be verified with SPIN under different types of assumptions about

Listing 6.1: An example specification in the PROMELA language.

```
1  chan com = [0] of {byte};
2
3  int total;
4
5  active proctype counter() {
6    byte curr;
7
8    do
9    :: com ? curr ->
10     total = total + curr;
11   od;
12 }
13
14 active proctype generator() {
15   byte cnt = 0;
16
17   do
18   :: cnt < 127 ->
19     cnt++;
20       if
21       :: com ! cnt;
22       :: com ! (cnt * 2);
23       fi;
24   :: else -> break;
25   od;
26 }
27
28 active proctype monitor() {
29   do
30   :: assert(total < 16256);
31   od;
32 }
```

the environment (e.g., message loss, message duplications etc). Once the correctness of a model has been established with SPIN, that fact can be used in the construction and verification of all subsequent models.

PROMELA programs consist of *processes*, message *channels*, and *variables*. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behaviour, channels and global variables define the environment in which the processes run."

For a list of features that is supported by SPINJ, see appendix C.

To illustrate the PROMELA language, we will first explain the example model of Listing 6.1. At the top of this example, there are two global variables declared: a channel `com` and an integer `total`. Then there are three processes defined:

1. The `counter` process, which reads numbers from the `com` channel and adds those numbers to the `total` counter.

2. The `generator` process, which generates numbers that are put into the `com` channel non-deterministically.

3. The `monitor` process, which monitors if the `total` variable never exceeds 16256. When it does, the assertion fails and the model checker reports an error.

All three processes are declared `active`, meaning that one instance of the process is started when the model is initalized.

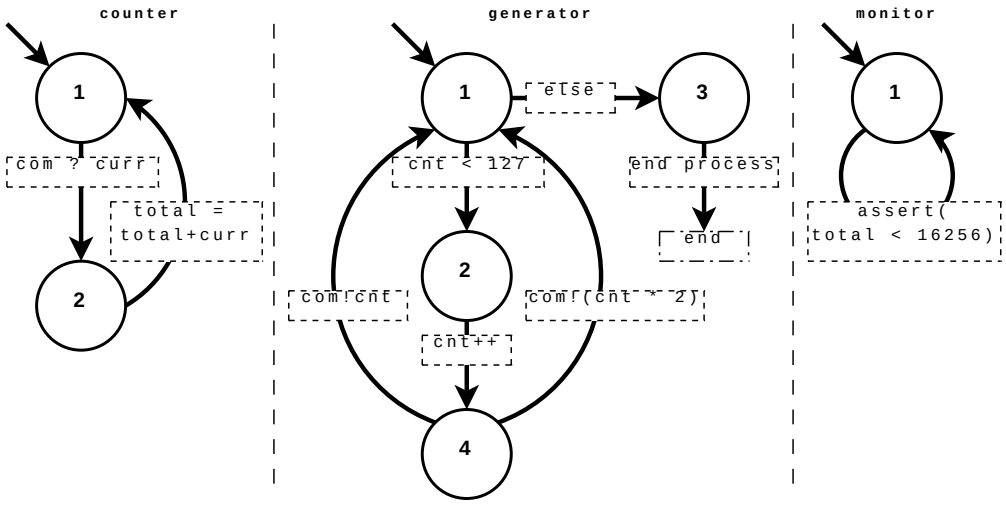

Figure 6.1: A graphical representation of the state in the three processes of the example from Listing 6.1.

The *semantics* of a Promela process is defined as a finite state automaton (see 2.1.1). In Figure 6.1 the automata for the three processes are shown.

Let us look at the `generator` process. State 1 of this process is the starting state (as with all processes), and there are two transitions possible from this state. This choice is a deterministic one, either the `cnt` variable is less than 127 and we go to state 2, or it is not and we go to state 3.

From state 2 there is only one transition executable, increasing `cnt` variable by 1. After state 4 there is again a choice, but this time a non-deterministic one. We can either send the value of `cnt` over the channel, or we can send twice that value over the channel. Also both end up back in the starting state, because of the do-loop.

The last transition that is shown, is the `end process` transition. This one deletes the running process from the model and therefore does not end in any state. The other two processes are self-explanatory and will not be discussed further.

For more details on the PROMELA language, see [21] and [1]. The rest of this section will explain how the compiler transforms a PROMELA model (like the one given in the Listing 6.1) to automata (like the one in Figure 6.1).

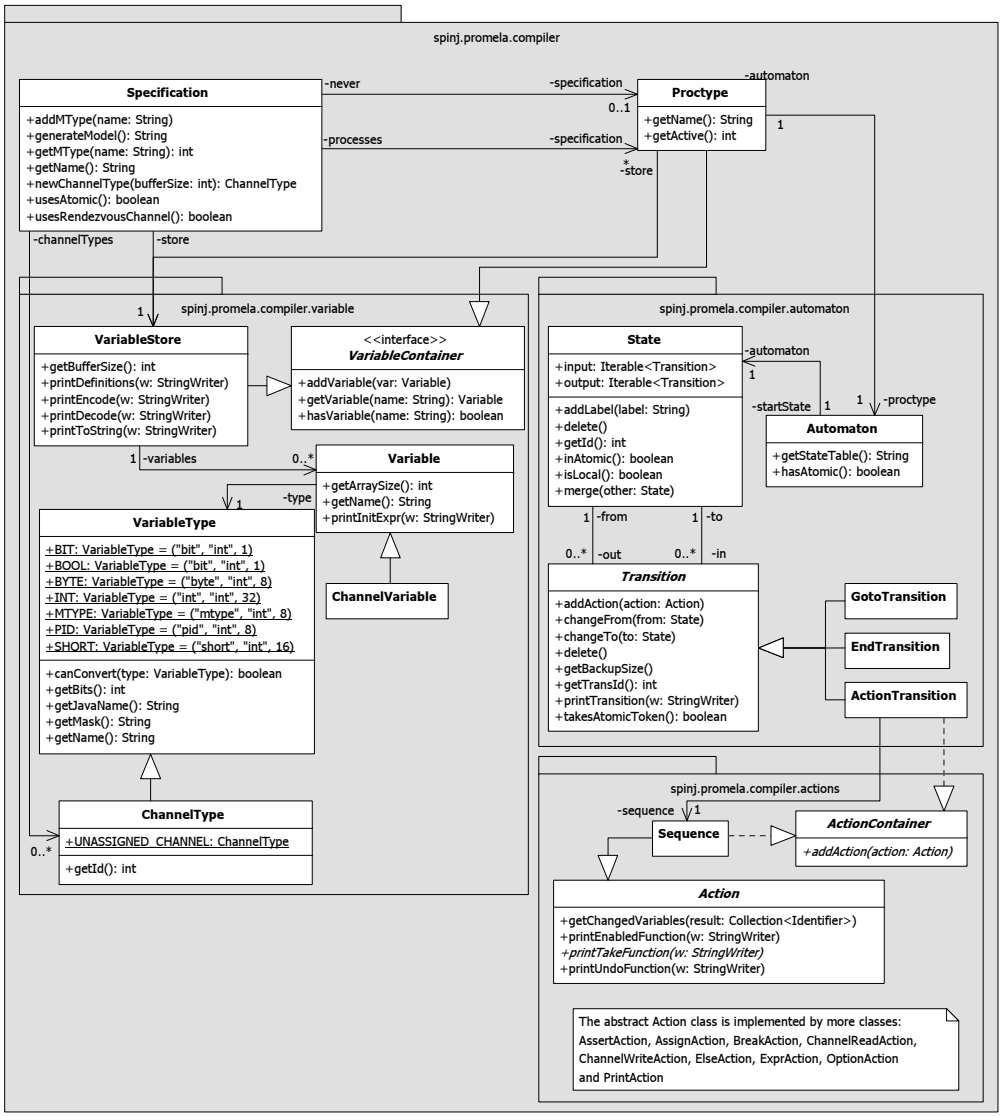

Figure 6.2: An UML diagram that describes the intermediate representation of the PROMELA compiler of SPINJ.

6.1.2 The compiler

The design of the PROMELA compiler was described earlier in Section 3.3. The parser for this compiler was written in JavaCC and transforms the PROMELA code into the intermediate representation as described in the following section.

The intermediate representation is a collection of Java objects describing the finite state automaton of the processes of the PROMELA model. These automata are optimised using the automaton optimizers, which will be described further on.

6.1.3 Intermediate representation

The intermediate representation of the PROMELA code is a collection of Java objects describing the finite state automata of the processes of the PROMELA model. These different types of objects are shown in Figure 6.2. The main object that represents the complete PROMELA specification is the `Specification` object. This object holds the global variables as well as the processes that are described here.

The processes are represented by `Proctype` objects. These objects are responsible for storing the local variables that are declared within the process, as well as the automaton that describes the states and transitions (i.e. the statements).

All the global variables from the `Specification` as well as the local variables from the `Proctype` are stored in the `VariableStore` object. This store keeps track of all the variables and their declaration (type and initialising expression).

Automata

The automata are described by the `Automaton` object, which holds all the states and the transitions in the object. It has a reference to the starting state, from which the complete graph can be reached by following the transition to next states.

States The `State` objects that are stored in the `Automaton` each represent a single state, as the ones in Figure 6.1. Each state has a list of input transitiona and output transitions.

Transition Each `Transition` object represents a single transition in the automaton, always going from one state to the next. Therefore the `Transition` object has two references: the `to` state and the `from` state. There are different types of `Transition` objects defined in SpinJ.

- The `GotoTransition` object is a `Transition` that can not store any actions, but represents a goto-statement or a break-statement. These transitions can often be optimised, and therefore are not seen in Figure 6.1.

Listing 6.2: Example PROMELA code to show why we need an empty transition for each option

```
1 do
2 :: do
3    :: x < 50 -> x++;
4    od
5 :: x < 100 -> x = x + 2;
6 od
7 }
```

- The `EndTransition` object is a special `Transition` that is used to terminate the process. E.g. this type of transition can be seen in the `generator` process in Figure 6.1, pointing to the end block.
- The `ActionTransition` object is the most common `Transition`, which stores one or more `Action` objects within it. These actions represent PROMELA statements that have to be executed by that transition.

Actions An `Action` object represents a single PROMELA statement from a process, e.g. `curr < 127` or `com!curr`. Therefore there are a number of different implementations of the `Action` object. This internal representation is straightforward and will not be described fully in this thesis.

Automaton Optimizers

The automatons that are generated by the parser may not be minimal automatons and can be optimised. For example, each option in a do-loop starts with an empty transition to make it to possible to compile the example from Listing 6.2.

When performing a do-loop directly inside an other do-loop, we need the extra state just before the inner do-loop. Otherwise we do not have the right place to return after the `x++` statement. Therefore there is always an extra transition and state created for each option in the do-loop.

Many of these empty transitions can be optimised though. The approach to optimise such an automaton is based on the implementation of SPIN. An example of this is shown in Figure 6.3, showing what the original states were as generated by the parser, and how the following 4 steps optimises the automaton. The code that we use is the `generate` process from Listing 6.1.

1. *The initial representation* that is generated by the parser. Here we can see that there a couple of empty transitions, due to the options in the do-loop and if-statement. Also an explicit *break* statement is shown.
2. *Statement merging*, which merges sequences of safe or atomic steps. Figure 6.4 shows the two situation where this can be applied. In both cases there are three states, which are connected through 2 single transitions.

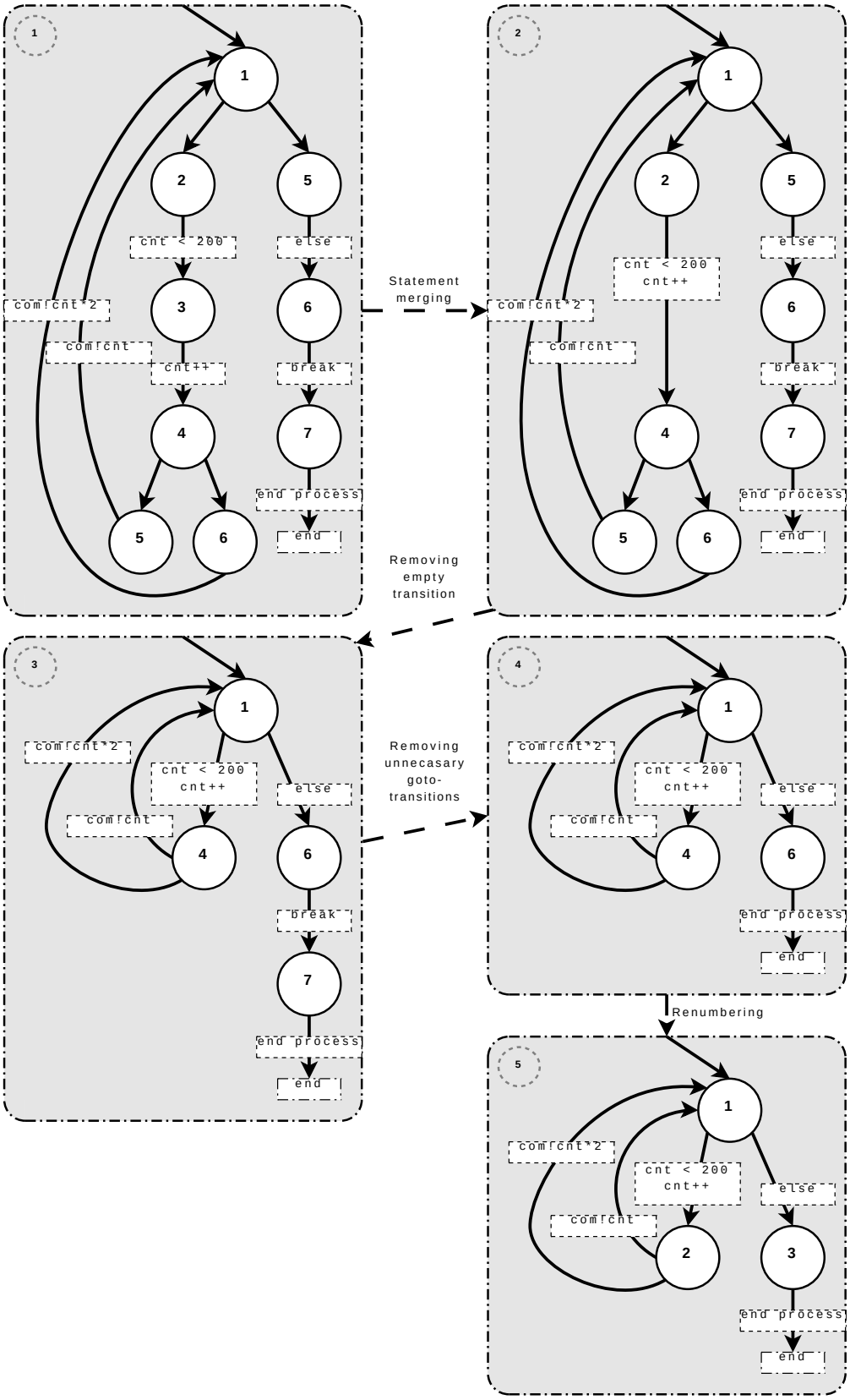

Figure 6.3: The four steps of optimising the intermediate representation of the states of the `generate` process

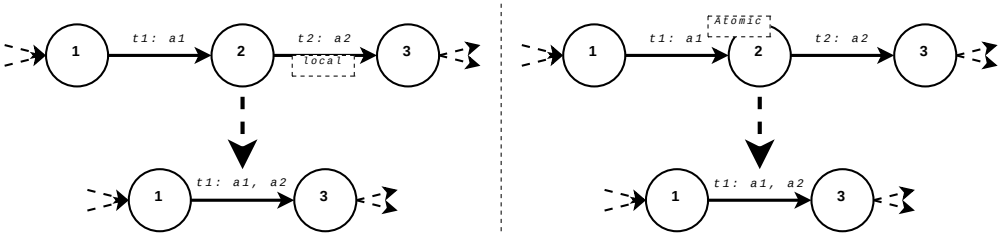

Figure 6.4: Two possible situations in an automaton where statement merging can reduce the number of states.

State 2 can be merged with state 3 when the following conditions hold for that state:

- The state has only 1 input transition.
- The state has only 1 output transition.
- The single output transition of the state is always executable (for example, a simple assignment).
- The single output transition of the state contains only local actions (changing only local variables).

  or

- The state is inside an atomic sequence.

After merging state 2 and 3, the actions of t_2 will be appended to t_1 and t_1 will go to state 3.

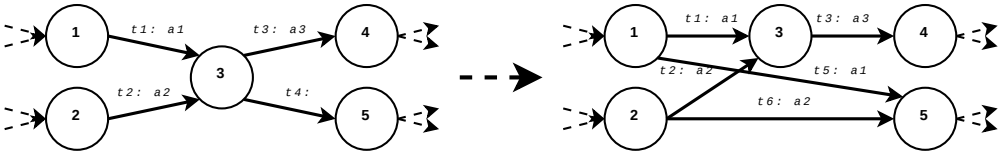

Figure 6.5: Removing an empty transition from the state space by duplicating the actions.

3. Removing *empty transitions*. Figure 6.5 shows the removal of such an empty transition, in this case t_4. This is done by duplicating all incoming transitions to the `from` state (in this case state 3). Then the new transition can point directly to the `to` state (state 5).

   In this example no state is removed, because there was another transition from state 3. But in most cases if one transition from a state is empty, then the rest also will be empty. And when all transition from a state are empty, the state can be removed.

4. Removing *`goto`-transitions*. `Goto`-transitions have no direct effect on the model, but they can only be removed safely if there or no *sibling* transitions (i.e. transitions going out from the same state) that have an effect.

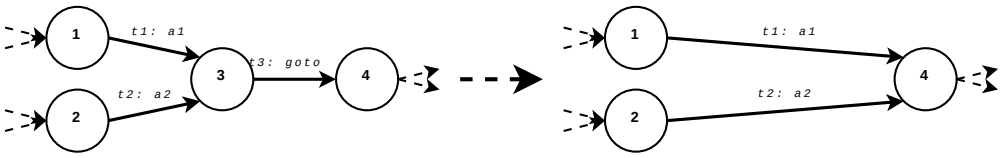

Figure 6.6: Removing a state that has only a single `goto`-transition going out from it.

Figure 6.6 shows an example where an `goto`-transition (t_3) can be removed, because from state 3 it is the only outgoing transition. This state can be removed by simply deleting t_3 and pointing the input transitions (t_1 and t_2) to state 4. Then state 3 has no more incoming transitions and can be deleted

5. *Renumbering* all the states, needed for efficiency in the generated code. It simply finds all the states and renumbers them. Naturally this will not change the number of states and transitions in the intermediate representation.

6.2 The PROMELA Tool layer

The tool layer for the PROMELA language is built on top of the abstract layer for concurrent processes, which was described in Chapter 5. How these layers are related can be seen in Figure 6.7. As between the abstract layer and the generic layer, the tool layer relates to the abstract layer through inheritance. We have a `PromelaModel` that extends the `ConcurrentModel`, as with the `PromelaProcess` and `PromelaTransition`.

Also shown in Figure 6.7 is an example of the generated code. This code was generated from the PROMELA example from Listing 6.1. Here we can see that this implementation is again based on inheritance. Furthermore, the generated processes are all inner classes of the generated model. And the generated `PromelaTransitionHolder` objects are inner classes of the generated processes.

The rest of this section is going to explain the different parts of the tool layer.

6.2.1 The `PromelaModel`

A `PromelaModel` is an abstract object that is usually implemented by the code that will be generated by the compiler (see also appendix A). It holds a number of member variables and functions:

- The `addProcess()` function can be used to start a new process inside the current state of the model. This can be used when executing a `run` statement from PROMELA. There is a maximum of 255 processes that can be running simultaneously. This is because the reference number that is returned must be storable in a single byte.

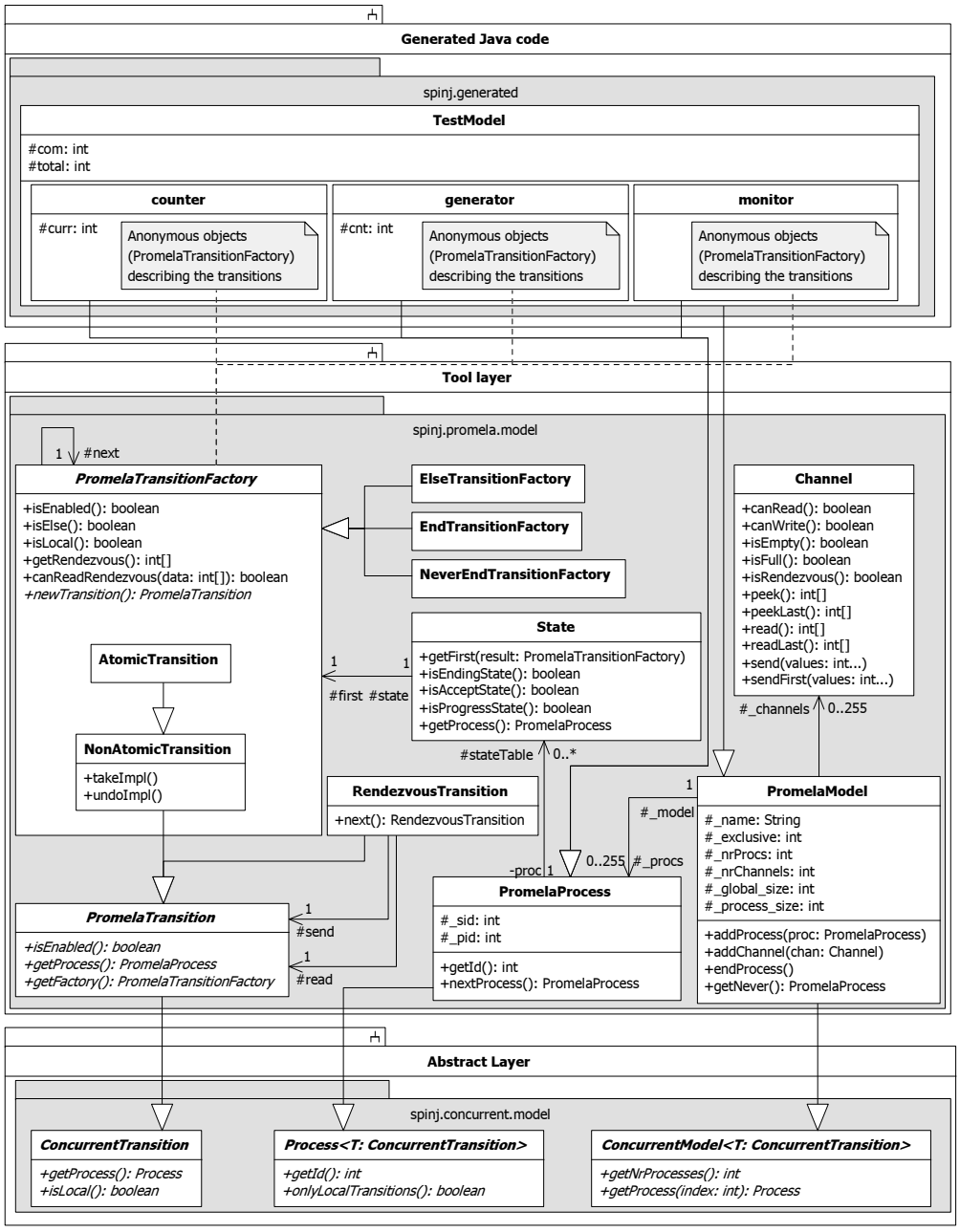

Figure 6.7: UML diagram describing the Tool layer that implements the PROMELA language, on top of the abstract layer that was described before.

- The `addChannel()` function adds a new channel to the current state of the model, which can be used through the reference that is returned (also a single byte). Channels are further explained in Section 6.2.4.
- The `endProcess()` function kills the last process. This function must be called when the ending transition is executed.
- The `getNever()` function can be overridden to return a new instance of the never-claim. By default it returns `null` to indicate that there is no never-claim available. For more information about never-claims, see Section 6.2.6.
- The `nextTransition()` function is already implemented by using the processes that are stored here. This is explained further in Section 6.2.5.

The only parts of a `PromelaModel` that have to be defined are the `encode` and `decode` functions and an explicit constructor.

6.2.2 The `PromelaProcess`

The `PromelaProcess` is a basic implementation of the `Process` from the abstract layer. It contains the following information:

- The *model* that it belongs to.
- The *state table*, which is an array of `State` objects. The `State` object from this package contains a list of `PromelaTransitionFactory` objects, and some meta information (e.g. whether this particular state is an ending state). The factory objects can be used to generate new instances of the `PromelaTransition`s.
- The *state identifier*, which is the identifier of the current state. This can be used to look in the table for the transitions that can be executed from this state.
- The *process identifier*, which is the unique identifier of that process.

Furthermore there are a few functions already implemented:

- The *constructor* accepts the model that it belongs to, the state table and the starting state.
- The `getId()` returns the unique process identifier.
- The `conditionHolds()` uses the meta information from the current state to determine if the condition holds (e.g. whether this particular state is an ending state).
- The `nextTransition()` is implemented by reading the state table for next transitions. This is explained in further detail in Section 6.2.5.

- The `onlyLocalTransitions()` checks from the current state if all transitions from that state are local.

The rest of the functions (e.g. `encode` and `decode`) have to be implemented by the generated code.

6.2.3 The `PromelaTransitions`

The state table of each of the processes contains a list of states, each holding a list of `PromelaTransitionFactory` objects. The `PromelaTransitionFactory` is responsible for creating the `PromelaTransion` objects when the next transition is being calculated (see Section 6.2.5).

Returning a new `PromelaTransition` each time is needed, because this object needs to remember what was changed when the `take` method is executed. This way, when calling the `undo` method, the `PromelaTransition` has the information it needs to undo the changes. Unfortunately this means that it is not possible to reuse the object, because otherwise the `backup` information will be overwritten.

6.2.4 Channels

In PROMELA channels can be used to communicate between processes. There are two types of channels: synchronous or asynchronous. Synchronous channels are also called rendez-vous channels, because they cause the read and send action to be executed simultaneously. This way a variable can be send from one process to an other without any other transition executed in between.

Asynchronous channel have a buffer, which gets filled by the send actions and are emptied by read actions. Buffered channels can cause the state space to expand, because the state after one send or read action will be stored (including the state of the buffer).

The implementation in the tool layer of channels is done by the `Channel` object. This object has an array of integer arrays that functions as a buffer. In this implementation, even with a rendez-vous channel there is a buffer of size 1. Since more than one variable can be send at a time, each item in the buffer is an integer array.

As can be seen in Figure 6.7, the `Channel` object has several functions that correspond to different PROMELA language constructs. For example, we can check if a channel is full or empty or we can send and read variables.

When defining a channel in PROMELA the size of the buffer and the types of the elements of the buffer have to specified. These are variables that will be stored when a new `Channel` is created.

6.2.5 Calculating the next transition

The `nextTransition` method that is defined by the `Model` object is implemented on two places in the tool layer: in the `PromelaModel` for calculating the transitions of the complete (concurrent) model, and in the `PromelaProcess` for calculating the next transition of that specified process. In

the PROMELA language there are two constructs that can effect which transition can be executed: atomic sequences and rendez-vous channels.

The atomic token

An atomic sequence in PROMELA is used to execute several statements in one indivisible step. This means that once the first statement of the atomic sequence is executed, the *atomic token* is taken. While the atomic token is in possession by one of the processes, that process is the only one that may execute statements until it releases the atomic token. This happens at the end of an atomic sequence or when the process is blocked on a statement within the atomic sequence.

When calculating which transition can be executed next, the atomic token must be taken into account. While the atomic token is not possessed by any process, the algorithm can be executed normally. But when the atomic token is in possesion by one of the processes, only that process may provide the next transition.

Rendez-vous channels

As explained in Section 6.2.4, there are channels that have no buffer: rendez-vous channels. These type of channels also need a special type of handling, because the send and read action must be performed simultaneously. When a match between a send and a read action has been found, the `nextTransition` function returns a special `RendezvousTransition` object that holds both transitions. Then these can be executed after one another.

The algorithm

In the `PromelaModel`, the `nextTransition` mostly makes use of the `nextTransition` function of the `PromelaProcess`. As can be seen on line 257 of Listing 6.3, this algorithm already takes the atomic token into account. If there is an exclusive process that has the atomic token, it only checks that specific process for transitions. When there were no transitions enabled in that process (line 269), it returns a new `EndAtomic` transition. This transition is a special type that removes the token, such that the next time other transitions can be executed.

When the atomic token is not held by any process, the algorithm starts looking from the last process that it encountered (line 258). Then it finds any next transition that can be executed until either one is found, or there are no more processes meaning that there are no further transitions to take.

The `nextTransition` function in the `PromelaProcess` class can be seen in Listing 6.4. It first checks if this process may even return a transition, by checking the atomic token. This check is performed to make this method valid, even when calling it directly.

After this check, the lines 79 - 90 try to find the first factory that can be used to generate the next transition. On lines 96 - 106 of this method the first match between a send action and a receive action can be found, returning a `RendezvousTransition`. On line 82 the last transition is checked whether it is

such a `RendezvousTransition`, because then it needs to check if there are other receive actions that can be coupled with the current send action. When calling the `nextRendezvousTransition()` function of the `RendezvousTransition`, it tries to find such a match.

After the first factory was selected, we can see the while-loop that checks each factory from the last transition if it can be used to generate a next transition. The first option (line 93) is when the transition in the factory is enabled, in which case the next transition is simply returned.

The second option (line 95) is when the factory returns rendez-vous data. In that case, a matching receive action is sought. A read action matches when the `canReadRendezvous` function of the `PromelaTransitionFactory` returns true when given the rendez-vous data. When such a match is found, a new `RendezvousTransition` is returned with those send and read actions. As said before, to find more matching receive actions for the current send action, the `nextRendezvousTransition` method is called on line 82.

The third option (line 107) is checking for an else-statement. An else-statement is only executable when there was no last transition and there is no other transition that is executable. This else statement will be returned on line 114 if that is the case.

6.2.6 Never claim

"A `never` claim can be used to define system behaviour that, for whatever reason, is of special interest. It is most commonly used to specify behaviour that should *never* happen. The claim is defined as a series of propositions, or boolean expressions, on the system state that must become *true* in the sequence specified for the behaviour of interest to be matched.

A `never` claim can be used to match either finite or infinite behaviours. Finite behaviour is matched if the claim can reach its final state (that is, its closing curly brace). Infinite behaviour is matched if the claim permits an ω-acceptance cycle. `Never` claims, therefore, can be used to verify both safety and liveness properties of a system." (from [21], page 441)

In SpinJ the never claim is retrieved from a `PromelaModel` using the `getNever()` method. This returns a `PromelaProcess`, because a never-claim can be parsed like a normal process (with some limitations). The only difference is that it is not possible to run the never-claim directly. Instead a `NeverClaimModel` can be created, with the `PromelaModel` and its `never` claim.

The `NeverClaimModel` is a special type of the `ConcurrentModel` that executes the given `never` process and the rest of the `PromelaModel` alternating. It does this by keeping track of a variable `turn`. When the `turn` is `true`, the `never` process may execute one step and when it is `false` the `PromelaModel` may execute one.

By implementing the `TransitionListener` it can flip the `turn` variable each time a transition is taken or undone. This variable is also encoded and decoded in the relative functions.

Listing 6.3: The `nextTransition` method of `PromelaModel`.

```
254  public PromelaTransition nextTransition(PromelaTransition last) {
255    PromelaTransition next = null;
256    if (_nrProcs > 0) {
257      if (_exclusive == _NO_PROCESS) {
258        PromelaProcess proc = last==null ? _procs[0] : last.getProcess();
259        next = proc.nextTransition(last);
260        while (next == null) {
261          proc = proc.nextProcess();
262          if (proc == null) {
263            break;
264          }
265          next = proc.nextTransition(null);
266        }
267      } else {
268        next = _procs[_exclusive].nextTransition(last);
269        if (last == null && next == null) {
270          next = newEndAtomic();
271        }
272      }
273    }
274
275    return next;
276  }
```

Listing 6.4: The `nextTransition` method of `PromelaProcess`.

```
71  public PromelaTransition nextTransition(PromelaTransition last) {
72    if (_model._exclusive != PromelaModel._NO_PROCESS && _model.
          _exclusive != _pid) {
73      return null;
74    }
75    int[] msg = null;
76    PromelaTransitionFactory elseFactory = null;
77    PromelaTransitionFactory factory = null;
78
79    if (last == null) {
80      factory = getCurrentState().getFirst();
81    } else if (last instanceof RendezvousTransition) {
82      RendezvousTransition next =
83        ((RendezvousTransition) last).nextRendezvousTransition();
84      if (next != null) {
85        return next;
86      }
87      factory = last.getFactory().getNext();
88    } else {
89      factory = last.getFactory() == null ? null : last.getFactory().
            getNext();
90    }
91
92    while (factory != null) {
93      if (factory.isEnabled()) {
94        return factory.newTransition();
95      } else if ((msg = factory.getRendezvous()) != null) {
96        for (int i = 0; i < _model._nrProcs; i++) {
97          PromelaProcess proc = _model._procs[i];
98          if (proc != this) {
99            for (PromelaTransitionFactory read = proc.getCurrentState().
                  getFirst(); read != null; read = read.getNext()) {
100             if (read.canReadRendezvous(msg)) {
101               return new RendezvousTransition(_model, factory.
                      newTransition(),
102                   read.newTransition());
103             }
104           }
105         }
106       }
107     } else if (last == null && factory.isElse()) {
108       elseFactory = factory;
109     }
110     factory = factory.getNext();
111   }
112
113   if (elseFactory != null) {
114     return elseFactory.newTransition();
115   }
116   return null;
117 }
```

6.3 The generated `TestModel`

When compiling the PROMELA example from Listing 6.1, the result is the `TestModel` object. This section will explain the different parts of the generated code from `TestModel.java`. The complete file is shown in appendix A.

6.3.1 The static header

```
1  package spinj.generated;
2
3  import spinj.util.DataReader;
4  import spinj.util.DataWriter;
5  import spinj.promela.Run;
6  import spinj.promela.model.*;
7  import spinj.exceptions.*;
8
9  public class TestModel extends PromelaModel {
```

This piece of code shows the default header of the generated model. As we can see, it is stored in the `spinj.generated` package and imports the basic packages that are used. The name of the class that is generated comes directly from the filename that was compiled.

```
11   public static void main(String[] args) {
12     Run run = new Run();
13     run.parseArguments(args);
14     run.search(TestModel.class);
15   }
```

After the header, the `main` function is defined to make it easy to start model checking this model. It uses a `Run` class from the `spinj.promela` package, which parses all the command line options and then starts the search algorithm.

6.3.2 The generated `Channel`

In the generated Java code of the example from Listing 6.1, there is a class defined that describes the global channel defined by the `com` variable.

```
17   public static class Channel0 extends Channel {
18     public Channel0() {
19       super(new int[] {0xff}, 0);
20     }
21
22     public void encode(DataWriter _writer) {
23       _writer.writeByte(0);
24     }
25
26     public int getSize() {
27       return 1;
28     }
29
30     public boolean decode(DataReader _reader) {
```

```
31        if(_reader.readByte() != 0) return false;
32        return true;
33      }
34    }
```

This channel stores one variable at the time and has a buffer size of zero, meaning this channel is a rendez-vous channel. Since the `Channel` object implements the `Storable` interface (see Section 4.1.1), it also must implement the `encode()`, `getSize()` and `decode()` functions. These three make sure that the unique identifier (0 in this case) can be encoded and decoded, to identify the type of the channel. Since there is no buffer, this does not have to be stored.

This is in contast to SPIN where the buffer is always stored in the state vector. By dynamically encoding the buffer of a channel, SPINJ automatically compacts the state vector.

6.3.3 Global variable definition

```
36    int com;
37    int total;
```

The two global variables from the example have been made available here. All PROMELA types are stored in Java as integers, to make sure that the smaller types (`short`, `byte`, etc.) can be stored as if they were unsigned. The com variable is a reference to the channel, which is always a number between 0 and 255. Also see Section 6.4 for more explanation.

6.3.4 The Constructor

```
39    public TestModel() throws SpinJException {
40      super("Test", 7);
41
42      // Initialize the default values
43      com = addChannel(new Channel0());
44
45      // Initialize the starting processes
46      addProcess(new counter());
47      addProcess(new generator());
48      addProcess(new monitor());
```

The first line in the constructor calls the constructor of the `PromelaModel` with the name of this model and the initial buffer size. The initial buffer size of 7 refers to the number of bytes that is needed to store the *global vector*. In this case, it should store the number of processes that are running (1 byte), the number of channels that are initialised (1 byte) and the two global variables (5 = 4 bytes for `total` + 1 byte for `com`). This can easily be seen in the next part.

On line 43 there is room for initialisation of global variables. In this case we initialise the `com` variable with a new channel. After that, it initialises the three processes that are activated, by calling the `addProcess` function.

6.3.5 The implementation of the `Storable` interface

```
52  public void encode(DataWriter _writer) {
53    _writer.writeByte(_nrProcs);
54    _writer.writeByte(_nrChannels);
55    _writer.writeByte(com);
56    _writer.writeInt(total);
57    for(int _i = 0; _i < _nrChannels; _i++) {
58      _channels[_i].encode(_writer);
59    }
60    for(int _i = 0; _i < _nrProcs; _i++) {
61      _procs[_i].encode(_writer);
62    }
63  }
```

This implementation of the `encode` function first stores the two variables that have been explained before: the number of processes and the number of channels. Then it writes the global variables that were defined before. Remember that storing these 5 values cost exactly 7 bytes, which was defined in the constructor.

After writing the global vector, the channels are written to the `DataWriter` and then the processes. Both the channels and the processes implement the `Storable` interface, which makes it possible to encode them by simply calling the `encode` function.

```
65  public boolean decode(DataReader _reader) {
66    _nrProcs = _reader.readByte();
67    _nrChannels = _reader.readByte();
68    com = _reader.readByte();
69    total = _reader.readInt();
```

Decoding the state from the `DataReader` starts by simply reading the global vector, similar to what was done with encoding.

```
71    for(int _i = 0; _i < _nrChannels; _i++) {
72      _reader.storeMark();
73      if(_channels[_i] == null || !_channels[_i].decode(_reader)) {
74        _reader.resetMark();
75        switch(_reader.peekByte()) {
76          case 0: _channels[_i] = new Channel0(); break;
77          default: return false;
78        }
79        if(!_channels[_i].decode(_reader)) return false;
80      }
81    }
```

Decoding the channels is a bit more difficult. When the channels in the current state are the same as the state we are decoding, then the `Channel.decode()` function on line 73 will succeed and it can continue. It is possible that decoding a channel fails, because there was a different type of channel on that index. This can also be seen in Section 6.3.2, on line 31. In that case the correct type must be instantiated and decoded again.

This starts by restoring the mark in the `DataReader` back to the original place. Then the first byte determines the type of channel that we need to store here. In this example there is only one type of channel defined, called `Channel0`. When this succeeds, the channel state is decoded again.

```
83    int _start = _reader.getMark();
84    for(int _i = 0; _i < _nrProcs; _i++) {
85      _reader.storeMark();
86      if(_procs[_i] == null || !_procs[_i].decode(_reader)) {
87        _reader.resetMark();
88        switch(_reader.peekByte()) {
89          case 0: _procs[_i] = new counter(true); break;
90          case 1: _procs[_i] = new generator(true); break;
91          case 2: _procs[_i] = new monitor(true); break;
92          default: return false;
93        }
94        if(!_procs[_i].decode(_reader)) return false;
95      }
96    }
97    _process_size = _reader.getMark() - _start;
98    return true;
99  }
```

Decoding the processes is very similar to the channels. In this case, there are three types of processes, the `counter` with identifier 0, the `generator` with identifier 1 and the `monitor` with identifier 2.

In the `PromelaModel` there is also an extra variable that caches the number of bytes that is needed to store all the processes: `_process_size`. This variable is recalculated on line 96 to match the loaded processes again.

The rest of the `TestModel` contains the definitions of the processes and channels, which will be explained later on in this chapter.

6.3.6 The generated `PromelaProcesses`

In the example from appendix A, there are three implementations of the `PromelaProcess` generated. This part is going to explain how the code of the `counter` process relates to this `PromelaProcess`.

```
115 public class counter extends PromelaProcess {
116   protected int curr;
```

The first part of the `counter` defines the local variables. In PROMELA `curr` was defined as `byte`, but it is stored again as an integer (see Section 6.4).

```
118 public counter(boolean decoding) {
119   super(TestModel.this, new State[2], 0);
120
121   /*... state table definition ...*/
```

```
178 }
179
180 public counter() throws ValidationException {
```

```
181     this(false);
182
183   }
```

Each implementation of the `PromelaProcess` has two constructors. The first is to initialise the state table, which is also used in the `decode` function that was shown earlier. The second constructor calls the first, but also initialises the local variables using any possible parameters of the process.

The `encode` and `decode` functions are implemented to write or read the local variables. This is similar to what was done with the global variables. Also the other two processes look very similar to this one.

6.3.7 The generated state table

In the constructor of each of the processes of the example, there is a part that constructs the state table. Always the same type of statements are used:

```
factory = /* Create new factory, with its own Transition type */;
factory.append(/* another factory describing the second transition
    possible from the current state*/);
_stateTable[x] = new State(process, factory, false, false, false);
```

This example creates two instances of the `PromelaTransitionFactory`, links them through the `next` reference and adds them to a new `State`. This state is then stored in the state table for further use. When creating a new `State` object, the constructor takes three booleans as a value; if this state is an ending state, a progress state or an accept state.

In appendix A there are several examples of this. The following code shows one of these:

```
218     PromelaTransitionFactory factory;
219     factory =
220       new PromelaTransitionFactory(true, 19, 0, 1, "(cnt < 127); cnt++"
            ) {
221         public final boolean isEnabled() {
222           return (cnt < 127);
223         }
224
225         public final PromelaTransition newTransition() {
226           return new NonAtomicTransition() {
227             private int _backup_cnt;
228
229             public final void takeImpl() throws ValidationException {
230               _backup_cnt = cnt;
231               cnt = (cnt + 1) & 0xff;
232             }
233
234             public final void undoImpl() {
235               cnt = _backup_cnt;
236             }
237           };
238         }
```

```
239         };
240       factory.append(
241         new ElseTransitionFactory(20, 0, 2, false));
242       _stateTable[0] = new State(generator.this, factory, false, false,
              false);
```

This code defines the two transitions that are possible from the state 1 of the `generator` process. These can also be seen after step 5 of the graphical representation of this process in Figure 6.3. The first factory creates a `PromelaTransition` that executes two actions: $cnt < 127$ and $cnt++$. The first action determines the `isEnabled()` method.

The second action changes a variable and therefore a new anonymous `PromelaTransition` must be defined to back up the current value. Then the `take` function can increase the number by 1 and the `undo` function restores the backup.

The second factory that is appended to the linked list is an `ElseTransitionFactory`. This type is specially made for the *else* statement. It reads the other factories that it is linked to, to determine its behaviour. Also while calculating the next transition it has a special function (see Section 6.2.5).

Line 242 finally defines that these two factories create the transitions from the first state. It creates a new `State`, which is not an ending state, progress state or an accept state, and stores it as the first state.

6.4 Problems with Java

Using Java to implement PROMELA processes did have its problems. A few of those are described in this section.

6.4.1 Storing variables

Local and global variables in PROMELA can be of many types. Most of these types are unsigned (e.g. `byte`, `short`, etc.). Since Java does not support unsigned bytes or shorts, a different solution had to be found.

The solution chosen for SPINJ is to store all variables as integer. Since the `int` type in PROMELA is the only signed type, all types can be represented that way. But when assigning a value to a integer, it must take the number of bits into account. For example, when increasing a `byte` value by 1, the following Java code needs to be executed:

```
x = (x + 1) & 0xff;
```

This is done to make sure that only the lower 8 bits are written.

6.4.2 No goto-statements

In PROMELA there are goto-statements. These can be simulated by creating a transition from one state to another. But when dealing with `d_step` sequences, goto-statements must be written directly in the Java language. Since this is not possible in Java, the choice has been made not to support goto-statements

within `d_step` sequences. The `break` statement, despite the fact of being a type of goto-statement, is supported by SpinJ.

Chapter 7

Benchmarks

In this chapter, SpinJ is compared to the original Spin. First the benchmark suite is explained, which is used for the comparison. The second section will discuss the results from that benchmark.

7.1 BEEM: Benchmarks for explicit model checkers

In [37] Pelánek describes a benchmark suite that can be used for explicit model checkers, called BEEM. This benchmark was originally created for the Divine language, but Pelánek provides a translation to promela. Unfortunately this means that not all features of the language are used. For example, it does use rendez-vous channels, propositions and processes, but no dynamic process instantiation, never claims or buffered channels.

On the website [36] there are 57 different models available, each with one or more parameters, giving a total of 300 model instances. Unfortunately not all models have a (generated) promela version, which leaves only 43 models. Not all of these models are run while testing SpinJ. Of each different type of model only one instance is chosen that has a reasonably size. This is done to limit the amount of very small models (which do not produce a consistent time) or very large models (which will not run completely within the available memory).

The result is a set of 43 instances of these models that are used to compare SpinJ to Spin. The machine that has performed the benchmark is a Windows XP machine with 2Gb of RAM and an AMD Athlon X2 4200+ processor. For Spin the latest version at the time writing was chosen: `5.1.6`.

7.1.1 Running Spin and SpinJ

To run the different models a couple of options for both Spin and SpinJ have been fixed. The following script is therefore used to run it:

```
spin -a -o2 <modelname>
cl /DSAFETY /DWIN32 /DMEMLIM=1400 <option> /Ox pan.c
pan -m<depth> -w<size> -c0 -n

java -jar SpinJ.jar -nTest <modelname>
javac -cp .;SpinJ.jar spinj/generated/TestModel.java
```

```
java -server -Xms1250M -Xmx1250M -cp .;SpinJ.jar spinj.generated.
    TestModel <option> -m<depth> -w<size> -c0
```

On the first line, SPIN version 5.1.6 is used to compile the model to `pan.c`. For this we use the `-a` option, which tells SPIN to compile the model instead of simulating it. The `-o2` option disables a data-flow optimizer in the compiler, which is not yet implemented in SPINJ and is disabled to make a fair comparison.

On the second line, the `C++`-compiler of Visual Studio 2008[1] is used to compile `pan.c`. Here we use the `/DSAFETY` option to speed up the verifier by disabling code that is needed for lifeness properties, which are not used in the BEEM benchmark. The `/DWIN32` makes sure that `pan.c` uses the Windows libraries. Furthermore a limit on the memory has been set to 1400 Megabyte by the `/DMEMLIM` option. The `/Ox` option enables all optimisations that are known to this compiler.

The `<option>` is a dynamically changing option that determines the type of run that we are going to perform. For SPIN this option is a compiler switch, but for SPINJ this option can be given at runtime. There are 4 different runs of the BEEM benchmark:

1. `Without Partial Order Reduction`. This uses the `/DNOREDUCE` option to disable the reduction algorithm. This benchmark shows the simple raw speed of SPIN and SPINJ. This benchmark is somewhat limited, because some of the models run out of memory.

2. `With Partial Order Reduction`. This uses no specified option and therefore enables the reduction algorithm. This benchmark shows the reduction that is reached and how much overhead the Partial Order Reduction algorithm has.

3. `With bitstate hashing`. This uses the `/DBITSTATE` option to enable bitstate hashing [20] instead of the normal hash table. This benchmark shows how well the hash algorithms are implemented in SPINJ compared to SPIN.

4. `With hash compaction`. This uses the `/DHC4` option to enable hash compaction [40] instead of the normal hash table. This benchmark also shows how well the hash algorithms and the hash compaction tables are implemented in SPINJ compared to SPIN.

When finally running `pan` on the third line, a couple of runtime options are given. First, the maximum search depth (`-m`*depth*) is a dynamic value that is determined based on the model that will be run. Secondly the `-w`*size* switch sets the size of the storing table to 2^{size} entries. This number depends on the type of benchmark that is run. For the normal (with or without Partial Order Reduction) benchmarks, 24 is chosen, giving 16.7 million entries. For bitstate hashing 31 and for hash compaction 26 is chosen.

[1]The gcc compiler was tried first, but on this Windows machine it could not allocate more than 825Mb. The `C++`-compiler of Visual Studio did work perfectly up to 1700Mb.

By default the verifier quits when the first error is found. The `-c0` switch for `pan` enables it to run no matter how many errors are found. Lastly the `-n` suppresses the default listing of all unreachable states.

For SpinJ three very similar lines are executed. First the SpinJ compiler is called to compile the given model, with the `-nTest` switch to rename the model to `TestModel`. This makes running the rest of the script a lot easier. On line 6 the java compiler is called to compile the generated `TestModel`.

The last line is running the compiled `TestModel` with options that are very similar to the ones of Spin. The main difference here is that the compiler options are given at runtime, making it no longer necessary to recompile the model when choosing a different storage technique.

7.2 Results

This section is going to summarize the results of the 4 different runs that have been performed on the BEEM benchmark. The complete tables with all the benchmark results can be found in appendix B. This section shows a couple of highlights and average differences.

7.2.1 Without Partial Order Reduction

Table B.1 shows the results of the benchmark with Partial Order Reduction disabled. Here we can see that the results that are produced by SpinJ are very similar to Spin. The number of errors is the same for all models. Also the number of states and transitions is the same for all models except for four. This difference will be explained in Section 7.3.

	Minimum	Average	Maximum
Speed	1.29×	3.53×	5.73×
Memory	0.37×	0.71×	0.97×

Table 7.1: The difference between Spin and SpinJ in speed and memory for the benchmark without Partial Order Reduction.

In table 7.1 we can see the results of this benchmark, which has used 37 of the 43 available models which did not run out of memory. The other 6 ran out of memory. The average results are good, where the SpinJ is only 3.53 times slower than Spin but uses 29% less memory. Also the minimum and maximum values are well within range.

That SpinJ uses less memory than Spin is mainly due to the compact representation that is used in SpinJ. Spin uses a different approach to encoding the states, which always reserves bytes to store all information (e.g. for the buffer of all the channels). In SpinJ this information is compacted by first storing the number of entries of such a buffer and then only writing the part of the buffer that has been filled.

This can be seen clearly in the `firewire_link.7` model, where SpinJ is almost 3 times more efficient with memory than Spin. This model has 44

global channels and only a few other variables to store. The maximum size of the state vector in SPIN is 540 bytes, while SPINJ only has a maximum of 179 bytes.

Also when looking at the `elevator.3` model, we can see that SPINJ can find much more states. It even checks the complete model with only 1100 Mb of memory usage, while SPIN runs out of memory after seeing only 63% of the states.

7.2.2 With Partial Order Reduction

Table B.2 shows the result of the benchmark with the Partial Order Reduction enabled. A summary of the differences can be seen in table 7.2.

	Minimum	Average	Maximum
Speed	1.52×	3.77×	6.11×
Memory	0.36×	0.68×	0.94×

Table 7.2: The difference between SPIN and SPINJ in speed and memory for the benchmark with Partial Order Reduction.

The difference in speed between SPINJ and SPIN is only slightly worse when Partial Order Reduction is enabled. This can be explained by the way SPIN stores if a state was still on the stack. SPINJ has to search the stack if the state is on it, but SPIN has optimised this by storing one extra bit in the state store. In a previous version of SPINJ searching the stack was a linear search that sometimes slowed it down by more than two order of magnitudes when searching a very large stack. The current version has implemented a hash table for the stack, which speeds things up considerably.

The memory requirements seems hardly to have changed with Partial Order Reduction, as we would expect.

7.2.3 With Bitstate hashing

When bitstate hashing is enabled, it is interesting to see how many states are missed. Therefore only larger models (with at least 2000000 states) were searched completely in the second benchmark. Table B.3 shows the result of this benchmark with the selected 15 models.

	Minimum	Average	Maximum
Speed	1.87×	2.79×	4.64×
Memory	0.25×	0.47×	0.50×

Table 7.3: The difference between SPIN and SPINJ in speed and memory for the benchmark with Bitstate hashing.

In table 7.3 we can see that the speed of SPINJ in relation to SPIN is slightly better. But the memory usage of SPINJ is much better than that of SPIN. In most cases SPINJ simply uses 256 Megabytes ($2^{31}/8 = 268435456$ bytes) for

the hash table, plus a variable amount for the stack. SPIN uses an extra 256 Megabytes for a "bit stack", which presumably is used to store meta-data for partial order reduction. SPINJ solves this by having a hash table with the stack, to speed up searching the stack. This does explain why SPINJ uses less than half the memory of SPIN for this benchmark.

When looking at the number of states that are missed, we can hardly spot any differences. They never perform exactly the same, but because of the unpredictable behaviour of hash algorithms this is to be expected.

7.2.4 With Hash compaction

When using hash compaction as an approximate storage method, we would expect similar results to bitstate hashing. In the test results from table B.4 however, we see completely different results.

	Minimum	Average	Maximum
Speed	2.41×	3.24×	5.00×
Memory	0.89×	1.31×	1.70×

Table 7.4: The difference between SPIN and SPINJ in speed and memory for the benchmark with Hash compaction.

The speed is a bit slower than using bitstate hashing, as can be seen in table 7.4. But the memory usage is very different here; SPINJ actually uses more memory than SPIN. This can be explained by the way the hash compact table was implemented. SPINJ uses a probing table that uses a array of long values to store the 64 bits hash values. This uses a fixed amount of memory: $2^{26} * 8bytes = 512Mb$.

SPIN on the other hand uses a linked list for each entry in the hash table. This reduces the size of the table, because only a pointer to the first entry is stored, using 4 bytes. This reduces the size of the table to only 256Mb, but for each hash value that is stored in the table, an extra 20 bytes of memory is used. This means that models that have more than $\frac{256*1024*1024}{20} = 13421773$ states will use more memory than the implementation of SPINJ

7.3 Difference in the results

In the results using the different setting there are some differences. This section is going to explain a couple of these.

7.3.1 Optimising `goto`-transitions

For some models that have less states in SPINJ than they do in SPIN, there are unnecessary `goto`-statements in the code that SPIN does not optimise. One example is shown in Listing 7.1 from the `krebs.4` model.

When starting with the optimisation there are two `goto`-transitions generated, between the two `d_step` statements. In between these `goto`-transitions there is initially also an empty transition for the option in the second

Listing 7.1: An outtake of the `krebs.4` model, showing an unnecessary `goto`-statement

```
1 oxalacetrat: if
2 :: d_step {acetyl_co_a>=1 && H20>=1;acetyl_co_a = acetyl_co_a-1;H20 =
      H20-1;} goto citrat;
3 fi;
4 citrat: if
5 :: goto isocitrat;
6 fi;
7 isocitrat: if
8 :: d_step {NADp>=1;NADp = NADp-1;NADH = NADH+1;Hp = Hp+1;CO2 = CO2+1;}
      goto oxoglutarat2;
9 fi;
```

`if`-statement. Spin seems to lack the ability to remove the `goto`-statement that follows such an empty statement.

When removing the (useless) `if`-block around that `goto`-statement, Spin does produce exactly the same result for `krebs.2`. The number of states is then reduced, because there are less states in the processes (the state just before that `goto`-statement is removed). Also the number of transitions is reduced, because the `goto`-transition does not need to be executed anymore.

7.3.2 Taking transitions twice

The implementation of Partial Order Reduction in Spin is the cause of taking the same transition from the state twice. This is the cause for Spin finding more transitions than SpinJ for some models, while the number of states is the same.

This effect can be seen when producing the verbose output, by compiling `pan.c` with the `/DVERBOSE` option. A part of this is shown in Listing 7.2, from the `lamport_nonatomic.3` model.

This listing shows that on the first line the fifth process is preselected by the Partial Order Reduction algorithm. This process then executes transition 16 (on line 4), which results in a state that was already on the stack (line 6). Then the algorithm reverses that transition and unselects process 5.

Now it can execute transitions from the same state, but for all processes. Spin now chooses to execute transition 16 from process 5 again on line 13, resulting in the same state that was found earlier on the stack. This unnecessary transition is counted, which explains the difference between Spin and SpinJ in the number of transitions for 11 out of the 63 models.

7.3.3 Differences in the depth

Many models have a different depth in SpinJ when compared to Spin. This is due to the difference in taking transitions. The verifier in SpinJ does not know anything of the inner workings of the model, so some transitions are wrapped

Listing 7.2: A part of the verbose output of `pan`, generated from the `lamport_nonatomic.3` model

```
49847: proc 5 PreSelected (tau=32)
Pr: 5 Tr: 15
Pr: 5 Tr: 16
49847: proc 5 exec 16, 57 to 3, ((v==0))  non-accepting [tau=32]
49848: Down - program non-accepting [pids 5-0]
  Stack state 22391
49848: Up - program
49848: proc 5 reverses 16, 57 to 3
  ((v==0)) [abit=0,adepth=0,tau=0,48]
49847: proc 5 UnSelected (_n=3, tau=48)
Pr: 5 Tr: 15
Pr: 5 Tr: 16
49847: proc 5 exec 16, 57 to 3, ((v==0))  non-accepting [tau=0]
49848: Down - program non-accepting [pids 5-0]
  Stack state 22391
49848: Up - program
49848: proc 5 reverses 16, 57 to 3
  ((v==0)) [abit=0,adepth=0,tau=0,16]
```

in a special `Transition` object that mimics the specified behaviour. This can be seen in two cases:

1. When executing a *rendez-vous* action, first one process writes to the rendez-vous channel and then an other process reads it. In SpinJ this behaviour is simulated in a special `RendezvousTransition`, which executes these two steps as if they were one. In Spin these two transitions are executed separately, which causes the stack to increase faster. This is the cause for a small depth reached for SpinJ.

2. When an atomic token was taken and the selected process can not execute any transition, the atomic token is lost. In Spin this is done explicitly by the verifier, but SpinJ can not do this. This is solved by a special `EndAtomicTransition` that removes the atomic token. This causes the depth to be increased faster in SpinJ than it is in Spin.

Taking into account these two differences, the depth can be less or more in SpinJ when compared to Spin. This all depends on the number of rendez-vous transitions and the number of atomic tokens that are lost.

Chapter 8

Conclusion

This chapter concludes this thesis. It starts by giving a summary and it then evaluates the result. Finally some possible future work is discussed.

8.1 Summary

Overall architecture In Chapter 3 the overall architecture of SpinJ is discussed. It uses a similar approach as in Spin, where it first compiles the PROMELA code into a Java source file that can be compiled and subsequently verified. To verify such a generated model, a library is used that is inspired by the conceptual framework of Mark Kattenbelt [32]. Here we use three layers (the *generic* layer, the *abstract* layer and the *tool* layer), where each layer describes more details of the model. Also the design of the compiler is discussed, which uses an intermediate representation of the automaton of each process. This intermediate representation can then be optimised (e.g. state merging) to produce a smaller automaton.

The generic layer In Chapter 4 the lowest layer is described. This generic layer has a very basic description of the model, which simply consists of states and transitions. Although not much is known about the inner workings of the model, most of the algorithms are implemented on this level. This includes simulation techniques (e.g. user simulation, random simulation), searching algorithms (e.g. Depth First Search, Breadth First Search), storage techniques (e.g. Hash tables, Bitstate hashing, Hash compaction) and hash functions. For each different type of algorithm there is an abstract base class that must be implemented (e.g. the `StateStore` or `SearchAlgorithm` class). This makes it easy to implement a new type of algorithm and test it directly with real models.

The abstract layer In Chapter 5 the abstract layer is described, which is an extension of the generic layer. This layer adds the notion of concurrency to the model, where it consists of one or more processes. Since these processes can be seen as models by themselves, the concurrent model is really a Cartesian product of these smaller models. This knowledge about the model makes it

possible to implement Partial Order Reduction (POR) on this layer, which can greatly reduce the number of states for the concurrent model.

The tool layer In Chapter 6 the tool layer is explained, in which the PROMELA models can be expressed. These PROMELA models have knowledge about specific PROMELA constructs (e.g. channels, the atomic token, timeouts). Also on this layer the never-claim is implemented, to make it possible to check LTL properties. All these classes can be used to make it easier to generate the model. The generated model is an extension of the PROMELA model. It only has to implement a limited set of methods.

The benchmarks In Chapter 7 it is explained how SPINJ is put to the test against SPIN itself. The BEEM benchmark set was used to perform four different tests: without POR, with POR, with bitstate hashing and with hash compaction. These results show that SPINJ is on average about 3.5 times slower than SPIN, but uses less memory. These results are further discussed in the next section.

8.2 Evaluation

In the introduction we have set ourselves six goals that SPINJ should reach. In this section we are going to evaluate each of these goals.

- *The design and implementation of* SPINJ *should be object-oriented, reusable and well documented.*

  The design of SPINJ is completely object-oriented, using inheritance to create the link between the different layers in the SPINJ library. This layered design makes it much easier to reuse parts of the system. For example, we can now create a new tool layer that implements a completely new language that can reuse the abstract and generic layer from the current version of SPINJ.

  Chapter 3 to 6 give a good overview of how this library was designed and implemented, making the documentation also complete.

- *The design and implementation of* SPINJ *should be extendable, such that other algorithms are easily added. It should be possible to extend the modelling language.*

  As said before, extending the modelling language is easy be replacing (parts of) the tool layer. But also adding other storage/search/hashing algorithms is easy. All of these algorithms have an abstract base class defined (e.g. `StateStore`, `SearchAlgorithm`, `HashAlgorithm`). When creating a new implementation of one of these algorithms, only this abstract class needs to be extended.

- *The performance of* SPINJ *should be comparable to* SPIN *(both in time and space), where one order of magnitude is acceptable.*

As shown in Chapter 7 SpinJ is only about 3.5 times slower than Spin which is less than one order of magnitude. Furthermore the memory requirements for SpinJ is even less than that of Spin. This makes SpinJ usable for models with millions of states.

- SpinJ*'s simulator and verifier should use the same Java code.*

  This is true for SpinJ, since the generated model can also be executed by one of the simulation algorithms. This reduces the risk that the simulator handles certain parts of the code differently than the verifier, which was a problem in previous versions of Spin.

- SpinJ *should support several of* Spin*'s optimisation algorithms, e.g. partial order reduction, bitstate hashing and hash compaction.*

  All three examples of optimisations algorithms are implemented. The POR algorithm is implemented on the abstract layer, because it needed to know about processes. Bitstate hashing and hash compaction are both implemented as classes that extend the `StateStore`. Furthermore, statement merging is implemented in SpinJ's compiler to reduce the generated state space.

- *The output and parameters of* SpinJ *should resemble the ones of* Spin *as much as possible.*

  The parameters of Spin are partly ported to SpinJ. But a lot of options of Spin are compiler switches, which in SpinJ must be given at runtime. The syntax of these parameters is the same though.

  The output of SpinJ is very similar with regards to basic output. Unfortunately the trace files that are created when an error was found are not compatible, because the numbering of transitions is not the same.

Overall we can conclude that all the goals have been reached, making SpinJ a fast and extendible model checker.

8.3 Future work

There are still parts of SpinJ that could be improved. One of these parts is the promela compiler. Currently it is implemented specifically for the promela language, but it is possible to make this more reusable. Currently it is not possible to reuse the optimisation techniques that were implemented here, because the internal representation is specifically created for the promela language. When using a more abstract description that could be useable for other languages, parts of the compiler would become more reusable.

Other future work on SpinJ would mostly consist of extending its functionality. A few possibilities are:

- To implement Minimize Automata [24] as a storage technique.

- To implement collapse compression [19]. This would be a wrapper around the concurrent model that reimplements the encode function, which stores the encoded representation of the processes in an internal hash table.
- Extending the PROMELA compiler to support the more exotic language features. One example could be to support inline Java code (like the `c_code` block that is supported by SPIN).
- To implement a new search algorithm that supports threading, making use of multiple processors.
- To implement a matching thread-safe hash table. It is possible to implement a lock-free high-performance hash table, as described in [11].
- To add support for another modelling language (e.g. Divine [8]).

Appendix A

The generated code for a PROMELA example

This appendix shows an example of a PROMELA model, which has been compiled using the SpinJ compiler. The first section shows the example, while the second one shows the generated Java code.

A.1 The PROMELA code

```
1  chan com = [0] of {byte};
2
3  int total;
4
5  active proctype counter() {
6    byte curr;
7
8    do
9    :: com ? curr ->
10     total = total + curr;
11   od;
12 }
13
14 active proctype generator() {
15   byte cnt = 0;
16
17   do
18   :: cnt < 127 ->
19     cnt++;
20       if
21       :: com ! cnt;
22       :: com ! (cnt * 2);
23       fi;
24   :: else -> break;
25   od;
26 }
27
28 active proctype monitor() {
```

```
29    do
30    :: assert(total < 16256);
31    od;
32  }
```

A.2 The generated Java code

```
1   package spinj.generated;
2
3   import spinj.util.DataReader;
4   import spinj.util.DataWriter;
5   import spinj.promela.Run;
6   import spinj.promela.model.*;
7   import spinj.exceptions.*;
8
9   public class TestModel extends PromelaModel {
10
11    public static void main(String[] args) {
12      Run run = new Run();
13      run.parseArguments(args);
14      run.search(TestModel.class);
15    }
16
17    public static class Channel0 extends Channel {
18      public Channel0() {
19        super(new int[] {0xff}, 0);
20      }
21
22      public void encode(DataWriter _writer) {
23        _writer.writeByte(0);
24      }
25
26      public int getSize() {
27        return 1;
28      }
29
30      public boolean decode(DataReader _reader) {
31        if(_reader.readByte() != 0) return false;
32        return true;
33      }
34    }
35
36    int com;
37    int total;
38
39    public TestModel() throws SpinJException {
40      super("Test", 7);
41
42      // Initialize the default values
43      com = addChannel(new Channel0());
44
```

```
45      // Initialize the starting processes
46      addProcess(new counter());
47      addProcess(new generator());
48      addProcess(new monitor());
49
50    }
51
52    public void encode(DataWriter _writer) {
53      _writer.writeByte(_nrProcs);
54      _writer.writeByte(_nrChannels);
55      _writer.writeByte(com);
56      _writer.writeInt(total);
57      for(int _i = 0; _i < _nrChannels; _i++) {
58        _channels[_i].encode(_writer);
59      }
60      for(int _i = 0; _i < _nrProcs; _i++) {
61        _procs[_i].encode(_writer);
62      }
63    }
64
65    public boolean decode(DataReader _reader) {
66      _nrProcs = _reader.readByte();
67      _nrChannels = _reader.readByte();
68      com = _reader.readByte();
69      total = _reader.readInt();
70
71      for(int _i = 0; _i < _nrChannels; _i++) {
72        _reader.storeMark();
73        if(_channels[_i] == null || !_channels[_i].decode(_reader)) {
74          _reader.resetMark();
75          switch(_reader.peekByte()) {
76            case 0: _channels[_i] = new Channel0(); break;
77            default: return false;
78          }
79          if(!_channels[_i].decode(_reader)) return false;
80        }
81      }
82
83      int _start = _reader.getMark();
84      for(int _i = 0; _i < _nrProcs; _i++) {
85        _reader.storeMark();
86        if(_procs[_i] == null || !_procs[_i].decode(_reader)) {
87          _reader.resetMark();
88          switch(_reader.peekByte()) {
89            case 0: _procs[_i] = new counter(true); break;
90            case 1: _procs[_i] = new generator(true); break;
91            case 2: _procs[_i] = new monitor(true); break;
92            default: return false;
93          }
94          if(!_procs[_i].decode(_reader)) return false;
95        }
96      }
97      _process_size = _reader.getMark() - _start;
```

```
98      return true;
99    }
100
101   public String toString() {
102     StringBuilder sb = new StringBuilder();
103     sb.append("TestModel: ");
104     sb.append("com = ").append(com).append('\t');
105     sb.append("total = ").append(total).append('\t');
106     for(int i = 0; i < _nrProcs; i++) {
107       sb.append('\n').append(_procs[i]);
108     }
109     for(int i = 0; i < _nrChannels; i++) {
110       sb.append('\n').append(_channels[i]);
111     }
112     return sb.toString();
113   }
114
115   public class counter extends PromelaProcess {
116     protected int curr;
117
118     public counter(boolean decoding) {
119       super(TestModel.this, new State[2], 0);
120
121       PromelaTransitionFactory factory;
122       factory =
123         new PromelaTransitionFactory(false, 18, 0, 1, "com?curr") {
124           public boolean isEnabled() {
125             if(com == -1 || _channels[com].isRendezVous() || !_channels[
                    com].canRead()) {
126               return false;
127             } else {
128               int[] _tmp = _channels[com].peek();
129               if(_tmp.length != 1) throw new UnexpectedStateException("
                      Channel returned the wrong number of variables");
130               return true;
131             }
132           }
133
134           public boolean canReadRendezvous(int[] _values) {
135             return _channels[com].isRendezVous()
136                     && _values.length == 2
137                     && _values[0] == com;
138           }
139
140           public final PromelaTransition newTransition() {
141             return new NonAtomicTransition() {
142               private int _backup_curr;
143
144               public final void takeImpl() throws ValidationException {
145                 _backup_curr = curr;
146                 int[] _tmp = _channels[com].read();
147                 curr = _tmp[0] & 0xff;
148               }
```

```
149
150             public final void undoImpl() {
151               _channels[com].sendFirst(curr);
152               curr = _backup_curr;
153             }
154           };
155         }
156       };
157     _stateTable[0] = new State(counter.this, factory, false, false,
          false);
158
159     factory =
160       new PromelaTransitionFactory(false, 2, 1, 0, "total=(total + curr
            )") {
161         public final PromelaTransition newTransition() {
162           return new NonAtomicTransition() {
163             private int _backup_total;
164
165             public final void takeImpl() throws ValidationException {
166               _backup_total = total;
167               total = (total + curr);
168             }
169
170             public final void undoImpl() {
171               total = _backup_total;
172             }
173           };
174         }
175       };
176     _stateTable[1] = new State(counter.this, factory, false, false,
          false);
177
178   }
179
180   public counter() throws ValidationException {
181     this(false);
182
183   }
184
185   public int getSize() {
186     return 3;
187   }
188
189   public void encode(DataWriter _writer) {
190     _writer.writeByte(0x0);
191     _writer.writeByte(_sid);
192     _writer.writeByte(curr);
193   }
194
195   public boolean decode(DataReader _reader) {
196     if(_reader.readByte() != 0x0) return false;
197     _sid = _reader.readByte();
198     curr = _reader.readByte();
```

```
199       return true;
200     }
201
202     public String toString() {
203       StringBuilder sb = new StringBuilder();
204       if(_exclusive == _pid) sb.append("<atomic>");
205       sb.append("counter (" + _pid + "," + _sid + "): ");
206       sb.append("curr = ").append(curr).append('\t');
207       return sb.toString();
208     }
209
210   }
211
212   public class generator extends PromelaProcess {
213     protected int cnt;
214
215     public generator(boolean decoding) {
216       super(TestModel.this, new State[3], 0);
217
218       PromelaTransitionFactory factory;
219       factory =
220         new PromelaTransitionFactory(true, 19, 0, 1, "(cnt < 127); cnt++"
              ) {
221           public final boolean isEnabled() {
222             return (cnt < 127);
223           }
224
225           public final PromelaTransition newTransition() {
226             return new NonAtomicTransition() {
227               private int _backup_cnt;
228
229               public final void takeImpl() throws ValidationException {
230                 _backup_cnt = cnt;
231                 cnt = (cnt + 1) & 0xff;
232               }
233
234               public final void undoImpl() {
235                 cnt = _backup_cnt;
236               }
237             };
238           }
239         };
240       factory.append(
241         new ElseTransitionFactory(20, 0, 2, false));
242       _stateTable[0] = new State(generator.this, factory, false, false,
              false);
243
244       factory =
245         new PromelaTransitionFactory(false, 21, 1, 0, "com!cnt") {
246           public final boolean isEnabled() {
247             return com != -1 && !_channels[com].isRendezVous() &&
                    _channels[com].canSend();
248           }
```

```
249
250           public int[] getRendezvous() {
251             if(!_channels[com].isRendezVous()) return null;
252             return new int[]{com, cnt};
253           }
254
255           public final PromelaTransition newTransition() {
256             return new NonAtomicTransition() {
257               public final void takeImpl() throws ValidationException {
258                 _channels[com].send(cnt);
259               }
260
261               public final void undoImpl() {
262                 _channels[com].readLast();
263               }
264             };
265           }
266         };
267       factory.append(
268         new PromelaTransitionFactory(false, 22, 1, 0, "com!(cnt * 2)") {
269           public final boolean isEnabled() {
270             return com != -1 && !_channels[com].isRendezVous() &&
                      _channels[com].canSend();
271           }
272
273           public int[] getRendezvous() {
274             if(!_channels[com].isRendezVous()) return null;
275             return new int[]{com, (cnt * 2)};
276           }
277
278           public final PromelaTransition newTransition() {
279             return new NonAtomicTransition() {
280               public final void takeImpl() throws ValidationException {
281                 _channels[com].send((cnt * 2));
282               }
283
284               public final void undoImpl() {
285                 _channels[com].readLast();
286               }
287             };
288           }
289         });
290       _stateTable[1] = new State(generator.this, factory, false, false,
              false);
291
292       factory =
293         new EndTransitionFactory(14);
294       _stateTable[2] = new State(generator.this, factory, true, false,
              false);
295
296     }
297
298     public generator() throws ValidationException {
```

```
299       this(false);
300
301       cnt = 0;
302     }
303
304     public int getSize() {
305       return 3;
306     }
307
308     public void encode(DataWriter _writer) {
309       _writer.writeByte(0x1);
310       _writer.writeByte(_sid);
311       _writer.writeByte(cnt);
312     }
313
314     public boolean decode(DataReader _reader) {
315       if(_reader.readByte() != 0x1) return false;
316       _sid = _reader.readByte();
317       cnt = _reader.readByte();
318       return true;
319     }
320
321     public String toString() {
322       StringBuilder sb = new StringBuilder();
323       if(_exclusive == _pid) sb.append("<atomic>");
324       sb.append("generator (" + _pid + "," + _sid + "): ");
325       sb.append("cnt = ").append(cnt).append('\t');
326       return sb.toString();
327     }
328
329   }
330
331   public class monitor extends PromelaProcess {
332
333     public monitor(boolean decoding) {
334       super(TestModel.this, new State[1], 0);
335
336       PromelaTransitionFactory factory;
337       factory =
338         new PromelaTransitionFactory(false, 23, 0, 0, "assert (total <
              16256)") {
339           public final PromelaTransition newTransition() {
340             return new NonAtomicTransition() {
341               public final void takeImpl() throws ValidationException {
342                 if(!(total < 16256)) throw new AssertionException("(total
                      < 16256)");
343               }
344             };
345           }
346         };
347       _stateTable[0] = new State(monitor.this, factory, false, false,
              false);
348
```

```
349     }
350
351     public monitor() throws ValidationException {
352       this(false);
353
354     }
355
356     public int getSize() {
357       return 2;
358     }
359
360     public void encode(DataWriter _writer) {
361       _writer.writeByte(0x2);
362       _writer.writeByte(_sid);
363     }
364
365     public boolean decode(DataReader _reader) {
366       if(_reader.readByte() != 0x2) return false;
367       _sid = _reader.readByte();
368       return true;
369     }
370
371     public String toString() {
372       StringBuilder sb = new StringBuilder();
373       if(_exclusive == _pid) sb.append("<atomic>");
374       sb.append("monitor (" + _pid + "," + _sid + "): ");
375       return sb.toString();
376     }
377
378   }
379 }
```

Appendix B

Benchmark results

Table B.1: Results of all the model that were run using SPIN and SPINJ. This benchmark was using run without Partial Order Reduction.

Model		states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	errors	depth	time (seconds)	memory (Mb)
adding.6	SPIN	7.61	11.75	1088640	99	4.38	325.43
	SPINJ	7.61	11.75	1088640	99	14.45	296.43
	Difference	0.00	0.00	0	0	3.30×	0.91×
at.4	SPIN	6.60	25.47	0	851760	10.70	493.09
	SPINJ	6.60	25.47	0	851760	36.88	346.74
	Difference	0.00	0.00	0	0	3.45×	0.70×
bakery.6	SPIN	11.85	40.40	2469	1724631	17.00	816.14
	SPINJ	11.85	40.40	2469	1724631	58.36	546.93
	Difference	0.00	0.00	0	0	3.43×	0.67×
blocks.3	SPIN	0.70	2.09	1	487724	0.95	166.43
	SPINJ	0.70	2.09	1	487724	3.78	121.60
	Difference	0.00	0.00	0	0	3.97×	0.73×
bopdp.3	SPIN	1.06	2.80	2	221	1.92	181.38
	SPINJ	1.05	2.77	2	205	6.14	128.03
	Difference	-0.01	-0.03	0	-16	3.20×	0.71×
bridge.2	SPIN	14.37	39.78	152317	125	37.90	996.72
	SPINJ	14.37	39.78	152317	108	102.22	612.43
	Difference	0.00	0.00	0	-17	2.70×	0.61×
brp.3	SPIN	2.27	5.18	6798	55183	4.52	400.75
	SPINJ	2.27	5.18	6798	41747	16.92	206.72
	Difference	0.00	0.00	0	-13436	3.74×	0.52×

(continued on the next page...)

Table B.1 – Continued

Model		states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	errors	depth	time (seconds)	memory (Mb)
cambridge.4	SPIN	2.24	5.71	144667	126174	6.13	517.36
	SPINJ	2.24	5.71	144667	120454	27.25	256.33
	Difference	0.00	0.00	0	-5720	4.45×	0.50×
driving_phils.4	SPIN	9.34	20.10	0	1880897	11.60	1399.99
	SPINJ	10.24	22.15	0	2016315	126.27	1125.31
	Difference	0.90	2.05	0	135418	10.89×	0.80×
	Out of memory: Spin & Spinj						
elevator.3	SPIN	8.61	33.40	0	249999	35.30	1399.97
	SPINJ	13.62	44.83	0	249999	203.52	1107.09
	Difference	5.01	11.43	0	0	5.77×	0.79×
	Out of memory: Spin						
elevator.4	SPIN	7.17	31.51	0	249999	37.10	1399.97
	SPINJ	13.50	49.80	0	249999	355.24	1219.76
	Difference	6.33	18.29	0	0	9.58×	0.87×
	Out of memory: Spin & Spinj						
elevator2.3	SPIN	7.67	55.38	0	883189	19.80	528.83
	SPINJ	7.67	55.38	0	883189	67.67	446.07
	Difference	0.00	0.00	0	0	3.42×	0.84×
elevator_planning.2	SPIN	11.43	93.28	7	7923957	39.40	1155.75
	SPINJ	11.43	93.28	7	7923957	138.05	798.10
	Difference	0.00	0.00	0	0	3.50×	0.69×
extinction.2	SPIN	0.81	3.58	211	166	2.36	184.41
	SPINJ	0.81	3.58	211	165	11.97	138.19
	Difference	0.00	0.00	0	-1	5.07×	0.75×
firewire_link.7	SPIN	2.47	8.23	22032	358	13.40	1361.07
	SPINJ	2.47	8.23	22032	306	63.03	516.47
	Difference	0.00	0.00	0	-52	4.70×	0.38×
fischer.6	SPIN	8.32	33.45	0	105361	16.00	584.25
	SPINJ	8.32	33.45	0	105361	59.30	448.98
	Difference	0.00	0.00	0	0	3.71×	0.77×
frogs.3	SPIN	0.76	0.77	188022	260	0.61	110.58
	SPINJ	0.76	0.77	188022	260	2.11	104.83
	Difference	0.00	0.00	0	0	3.46×	0.95×

(continued on the next page...)

Table B.1 – Continued

Model		states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	errors	depth	time (seconds)	memory (Mb)
gear.2	Spin	0.32	0.69	3564	29995	0.53	132.20
	SpinJ	0.32	0.69	3564	26217	3.05	92.84
	Difference	0.00	0.00	0	-3778	5.73×	0.70×
hanoi.2	Spin	0.53	1.59	0	531447	0.97	206.76
	SpinJ	0.53	1.59	0	531447	3.38	131.56
	Difference	0.00	0.00	0	0	3.49×	0.64×
iprotocol.4	Spin	8.29	29.63	0	1409	27.60	1399.97
	SpinJ	3.07	8.94	0	952	28.08	302.46
	Difference	-5.22	-20.69	0	-457	1.02×	0.22×
	Out of memory: Spin						
krebs.4	Spin	18.40	106.78	606	163	46.70	1322.04
	SpinJ	10.71	61.09	606	159	95.41	618.62
	Difference	-7.69	-45.68	0	-4	2.04×	0.47×
lamport.6	Spin	8.72	31.50	576	141173	11.00	504.37
	SpinJ	8.72	31.50	576	141173	38.31	334.09
	Difference	0.00	0.00	0	0	3.48×	0.66×
lamport_nonatomic.3	Spin	0.34	1.35	0	124787	1.19	119.31
	SpinJ	0.34	1.35	0	99864	4.88	89.09
	Difference	0.00	0.00	0	-24923	4.10×	0.75×
lann.3	Spin	7.73	37.08	431	1568361	35.40	1399.99
	SpinJ	13.63	71.48	432	1575709	228.89	1019.66
	Difference	5.90	34.40	1	7348	6.47×	0.73×
	Out of memory: Spin						
leader_filters.5	Spin	1.57	4.68	6090	65	1.77	172.20
	SpinJ	1.40	4.15	6090	64	5.91	128.30
	Difference	-0.17	-0.54	0	-1	3.34×	0.75×
loyd.2	Spin	0.36	0.97	0	199348	0.41	93.82
	SpinJ	0.36	0.97	0	199348	1.69	79.12
	Difference	0.00	0.00	0	0	4.15×	0.84×
mcs.3	Spin	0.57	2.08	0	141147	0.72	105.34
	SpinJ	0.57	2.08	0	141147	3.06	89.84
	Difference	0.00	0.00	0	0	4.26×	0.85×
msmie.4	Spin	7.13	11.06	640	52287	11.70	807.79
	SpinJ	7.13	11.06	640	52287	35.50	502.95
	Difference	0.00	0.00	0	0	3.03×	0.62×

(continued on the next page...)

Table B.1 – Continued

Model		states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	errors	depth	time (seconds)	memory (Mb)
needham.4	Spin	8.30	27.37	203680	54	33.90	1203.75
	SpinJ	8.30	27.37	203680	40	85.97	887.13
	Difference	0.00	0.00	0	-14	2.54×	0.74×
peg_solitaire.4	Spin	0.87	5.47	3290	24	5.22	110.81
	SpinJ	0.87	5.47	3290	24	10.91	108.02
	Difference	0.00	0.00	0	0	2.09×	0.97×
peterson.4	Spin	1.12	3.86	0	76394	1.56	134.64
	SpinJ	1.12	3.86	0	76394	4.75	110.75
	Difference	0.00	0.00	0	0	3.04×	0.82×
phils.5	Spin	0.53	4.25	1	434031	1.81	173.46
	SpinJ	0.53	4.25	1	434031	6.78	123.46
	Difference	0.00	0.00	0	0	3.75×	0.71×
pouring.2	Spin	0.05	1.23	0	93403	9.78	99.68
	SpinJ	0.05	1.23	0	46701	22.86	75.57
	Difference	0.00	0.00	0	-46702	2.34×	0.76×
protocols.5	Spin	9.36	37.09	336	1258045	25.00	1144.85
	SpinJ	6.24	23.82	336	815093	49.81	428.38
	Difference	-3.12	-13.27	0	-442952	1.99×	0.37×
public_subscribe.2	Spin	7.78	27.71	4605	650828	26.90	1399.93
	SpinJ	2.78	7.41	7200	98633	28.30	285.68
	Difference	-5.00	-20.30	2595	-552195	1.05×	0.20×
	Out of memory: Spin						
reader_writer.3	Spin	0.75	4.27	227894	81942	25.60	191.38
	SpinJ	0.75	4.27	227894	65545	33.13	131.15
	Difference	0.00	0.00	0	-16397	1.29×	0.69×
rether.3	Spin	1.01	1.40	8578	221308	1.59	282.30
	SpinJ	1.01	1.40	8578	174807	7.23	152.89
	Difference	0.00	0.00	0	-46501	4.55×	0.54×
rushhour.4	Spin	0.33	3.39	0	295929	2.34	222.19
	SpinJ	0.33	3.39	0	295929	13.17	127.57
	Difference	0.00	0.00	0	0	5.63×	0.57×
schedule_world.2	Spin	1.57	14.31	26000	20886	5.55	149.88
	SpinJ	1.57	14.31	26000	20886	18.67	139.93
	Difference	0.00	0.00	0	0	3.36×	0.93×

(continued on the next page...)

Table B.1 – Continued

Model		states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	errors	depth	time (seconds)	memory (Mb)
sokoban.2	Spin	0.76	2.01	20	690	1.05	128.16
	SpinJ	0.76	2.01	20	690	4.14	122.31
	Difference	0.00	0.00	0	0	3.94×	0.95×
sorter.3	Spin	1.29	2.74	0	893	2.23	162.54
	SpinJ	1.29	2.74	0	893	5.64	123.18
	Difference	0.00	0.00	0	0	2.53×	0.76×
szymanski.4	Spin	2.31	8.55	0	74079	3.16	206.61
	SpinJ	2.31	8.55	0	74079	11.41	156.32
	Difference	0.00	0.00	0	0	3.61×	0.76×
telephony.3	Spin	0.77	3.16	0	47009	1.09	113.45
	SpinJ	0.77	3.16	0	47009	4.77	97.24
	Difference	0.00	0.00	0	0	4.38×	0.86×

Table B.2: Results of all the model that were run using Spin and SpinJ. This benchmark was using run with Partial Order Reduction.

Model		states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	errors	depth	time (seconds)	memory (Mb)
adding.6	Spin	7.61	11.75	1088640	99	4.69	354.43
	SpinJ	7.61	11.75	1088640	99	15.19	296.43
	Difference	0.00	0.00	0	0	3.24×	0.84×
at.4	Spin	6.60	25.47	0	851760	11.60	525.91
	SpinJ	6.60	25.47	0	851760	42.20	346.74
	Difference	0.00	0.00	0	0	3.64×	0.66×
bakery.6	Spin	11.85	40.40	2469	1724631	18.40	868.98
	SpinJ	11.85	40.40	2469	1724631	63.84	546.93
	Difference	0.00	0.00	0	0	3.47×	0.63×
blocks.3	Spin	0.70	2.09	1	487724	1.02	176.70
	SpinJ	0.70	2.09	1	487724	3.94	121.60
	Difference	0.00	0.00	0	0	3.86×	0.69×
bopdp.3	Spin	1.06	2.78	2	221	1.99	185.39
	SpinJ	1.05	2.77	2	205	6.91	128.03
	Difference	-0.01	-0.01	0	-16	3.47×	0.69×

(continued on the next page...)

Table B.2 – Continued

Model		states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	errors	depth	time (seconds)	memory (Mb)
bridge.2	Spin	14.37	39.78	152317	125	40.30	1051.11
	SpinJ	14.37	39.78	152317	108	121.50	612.43
	Difference	0.00	0.00	0	-17	3.01×	0.58×
brp.3	Spin	1.33	2.43	6798	19382	2.27	269.68
	SpinJ	1.33	2.40	6798	14547	9.36	149.14
	Difference	0.00	-0.03	0	-4835	4.12×	0.55×
cambridge.4	Spin	2.14	5.35	144667	114745	5.98	506.00
	SpinJ	2.14	5.34	144667	108423	27.09	247.77
	Difference	0.00	-0.01	0	-6322	4.53×	0.49×
driving_phils.4	Spin	8.78	18.82	0	1799930	11.30	1399.94
	SpinJ	10.24	22.15	0	2016096	132.03	1125.24
	Difference	1.46	3.33	0	216166	11.68×	0.80×
	Out of memory: Spin & Spinj						
elevator.3	Spin	8.39	32.62	0	249999	34.40	1399.95
	SpinJ	13.62	44.83	0	249999	212.95	1107.09
	Difference	5.23	12.22	0	0	6.19×	0.79×
	Out of memory: Spin						
elevator.4	Spin	7.01	30.76	0	249999	38.50	1399.95
	SpinJ	13.50	49.80	0	249999	375.14	1219.76
	Difference	6.48	19.03	0	0	9.74×	0.87×
	Out of memory: Spin & Spinj						
elevator2.3	Spin	7.67	55.38	0	883189	20.60	565.76
	SpinJ	7.67	55.38	0	883189	70.63	446.07
	Difference	0.00	0.00	0	0	3.43×	0.79×
elevator_planning.2	Spin	11.43	93.28	7	7923957	40.80	1229.92
	SpinJ	11.43	93.28	7	7923957	139.56	798.10
	Difference	0.00	0.00	0	0	3.42×	0.65×
extinction.2	Spin	0.44	1.36	211	166	0.95	131.58
	SpinJ	0.44	1.36	211	160	5.83	104.67
	Difference	0.00	0.00	0	-6	6.11×	0.80×
firewire_link.7	Spin	0.45	0.94	22032	285	1.84	302.09
	SpinJ	0.45	0.94	22032	240	10.16	146.71
	Difference	0.00	0.00	0	-45	5.52×	0.49×

(continued on the next page...)

Table B.2 – Continued

Model		states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	errors	depth	time (seconds)	memory (Mb)
fischer.6	SPIN	8.32	33.45	0	105361	17.20	617.14
	SPINJ	8.32	33.45	0	105361	64.91	448.98
	Difference	0.00	0.00	0	0	3.77×	0.73×
frogs.3	SPIN	0.76	0.77	188022	260	0.66	113.51
	SPINJ	0.76	0.77	188022	260	2.31	104.83
	Difference	0.00	0.00	0	0	3.52×	0.92×
gear.2	SPIN	0.32	0.69	3564	29995	0.56	134.42
	SPINJ	0.32	0.69	3564	26217	3.31	92.84
	Difference	0.00	0.00	0	-3778	5.88×	0.69×
hanoi.2	SPIN	0.53	1.59	0	531447	1.03	216.44
	SPINJ	0.53	1.59	0	531447	3.64	131.56
	Difference	0.00	0.00	0	0	3.53×	0.61×
iprotocol.4	SPIN	4.69	10.54	0	1218	10.70	841.45
	SPINJ	3.07	8.91	0	952	32.52	301.96
	Difference	-1.62	-1.63	0	-266	3.04×	0.36×
krebs.4	SPIN	17.07	67.45	606	163	32.50	1300.17
	SPINJ	10.71	61.09	606	159	105.00	618.62
	Difference	-6.35	-6.35	0	-4	3.23×	0.48×
lamport.6	SPIN	8.72	31.50	576	141173	11.80	538.62
	SPINJ	8.72	31.50	576	141173	42.53	334.09
	Difference	0.00	0.00	0	0	3.60×	0.62×
lamport_nonatomic.3	SPIN	0.28	0.88	0	99064	0.83	112.25
	SPINJ	0.28	0.86	0	79341	3.70	85.13
	Difference	0.00	-0.02	0	-19723	4.47×	0.76×
lann.3	SPIN	7.33	35.04	431	1486017	33.80	1399.94
	SPINJ	13.63	71.48	432	1575709	241.58	1019.66
	Difference	6.30	36.45	1	89692	7.15×	0.73×
	Out of memory: Spin						
leader_filters.5	SPIN	1.52	4.26	6090	65	1.72	173.96
	SPINJ	1.40	4.15	6090	64	6.70	128.30
	Difference	-0.11	-0.11	0	-1	3.90×	0.74×
loyd.2	SPIN	0.36	0.97	0	199348	0.49	96.14
	SPINJ	0.36	0.97	0	199348	1.84	79.12
	Difference	0.00	0.00	0	0	3.79×	0.82×

(continued on the next page...)

Table B.2 – Continued

Model		states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	errors	depth	time (seconds)	memory (Mb)
mcs.3	Spin	0.51	1.67	0	134904	0.63	104.83
	SpinJ	0.51	1.65	0	134904	2.97	87.64
	Difference	0.00	-0.02	0	0	4.75×	0.84×
msmie.4	Spin	7.13	11.06	640	52287	13.10	836.08
	SpinJ	7.13	11.06	640	52287	44.72	502.95
	Difference	0.00	0.00	0	0	3.41×	0.60×
needham.4	Spin	2.36	3.80	203680	54	4.99	398.18
	SpinJ	2.36	3.80	203680	40	14.91	298.61
	Difference	0.00	0.00	0	-14	2.99×	0.75×
peg_solitaire.4	Spin	0.87	5.47	3290	24	4.24	115.09
	SpinJ	0.87	5.47	3290	24	11.16	108.02
	Difference	0.00	0.00	0	0	2.63×	0.94×
peterson.4	Spin	0.75	2.49	0	68989	1.00	118.41
	SpinJ	0.75	2.47	0	68989	3.56	96.75
	Difference	0.00	-0.02	0	0	3.56×	0.82×
phils.5	Spin	0.53	4.25	1	434031	1.88	183.14
	SpinJ	0.53	4.25	1	434031	7.48	123.46
	Difference	0.00	0.00	0	0	3.98×	0.67×
pouring.2	Spin	0.05	1.23	0	93403	9.06	100.83
	SpinJ	0.05	1.23	0	46701	23.13	75.57
	Difference	0.00	0.00	0	-46702	2.55×	0.75×
protocols.5	Spin	3.14	8.15	336	730301	6.53	488.71
	SpinJ	2.96	7.71	336	571812	20.58	252.96
	Difference	-0.18	-0.44	0	-158489	3.15×	0.52×
public_subscribe.2	Spin	2.71	5.46	7200	274039	5.58	584.22
	SpinJ	1.72	3.56	7200	86341	15.72	213.48
	Difference	-0.99	-1.90	0	-187698	2.82×	0.37×
reader_writer.3	Spin	0.75	4.27	227894	81942	24.90	195.17
	SpinJ	0.75	4.27	227894	65545	37.94	131.15
	Difference	0.00	0.00	0	-16397	1.52×	0.67×
rether.3	Spin	0.99	1.34	8578	221308	1.64	282.76
	SpinJ	0.99	1.34	8578	174807	7.75	151.14
	Difference	0.00	-0.00	0	-46501	4.73×	0.53×

(continued on the next page...)

Table B.2 – Continued

Model		states ($\cdot 10^6$)	transitions ($\cdot 10^6$)	errors	depth	time (seconds)	memory (Mb)
rushhour.4	Spin	0.33	3.39	0	295929	2.47	230.99
	SpinJ	0.33	3.39	0	295929	13.84	127.57
	Difference	0.00	0.00	0	0	5.60×	0.55×
schedule_world.2	Spin	1.57	14.31	26000	20886	5.70	156.88
	SpinJ	1.57	14.31	26000	20886	19.11	139.93
	Difference	0.00	0.00	0	0	3.35×	0.89×
sokoban.2	Spin	0.76	2.01	20	690	1.03	130.99
	SpinJ	0.76	2.01	20	690	4.27	122.31
	Difference	0.00	0.00	0	0	4.15×	0.93×
sorter.3	Spin	1.29	2.74	0	893	2.33	167.42
	SpinJ	1.29	2.74	0	893	6.58	123.18
	Difference	0.00	0.00	0	0	2.82×	0.74×
szymanski.4	Spin	2.27	7.12	0	68686	2.86	213.62
	SpinJ	2.27	7.10	0	68686	10.91	154.72
	Difference	0.00	-0.03	0	0	3.81×	0.72×
telephony.3	Spin	0.77	3.16	0	47009	1.17	117.33
	SpinJ	0.77	3.16	0	47009	5.28	97.24
	Difference	0.00	0.00	0	0	4.51×	0.83×

Table B.3: Results of all the model that were run using Spin and SpinJ. This benchmark was using run with Bitstate Hashing.

Model		states missed	transitions missed	errors missed	depth	time (seconds)	memory (Mb)
adding.6	Spin	3	6	0	99	6.70	512.20
	SpinJ	9	11	3	99	12.55	256.20
	Difference				0	1.87×	0.50×
at.4	Spin	1	4	0	851760	14.90	629.10
	SpinJ	1	3	0	851760	40.66	287.07
	Difference				0	2.73×	0.46×
bakery.6	Spin	18	64	0	1724631	22.50	646.58
	SpinJ	13	43	0	1724631	59.34	287.07
	Difference				0	2.64×	0.44×

(continued on the next page...)

Table B.3 – Continued

Model		states missed	transitions missed	errors missed	depth	time (seconds)	memory (Mb)
bridge.2	SPIN	38	104	0	125	44.40	512.20
	SPINJ	66	212	0	108	108.83	256.20
	Difference				-17	2.45×	0.50×
cambridge.4	SPIN	0	0	0	114745	6.08	521.76
	SPINJ	0	0	0	108423	28.22	260.05
	Difference				-6322	4.64×	0.50×
elevator2.3	SPIN	2	11	0	883189	26.80	611.82
	SPINJ	4	30	0	883189	73.61	287.07
	Difference				0	2.75×	0.47×
fischer.6	SPIN	2	12	0	105361	20.40	526.15
	SPINJ	17	68	0	105361	64.47	260.04
	Difference				0	3.16×	0.49×
iprotocol.4	SPIN	0	0	0	1218	11.30	520.78
	SPINJ	0	0	0	952	31.27	260.05
	Difference				-266	2.77×	0.50×
krebs.4	SPIN	10	39	0	163	43.30	1024.20
	SPINJ	9	49	0	159	102.31	256.20
	Difference				-4	2.36×	0.25×
lamport.6	SPIN	3	12	0	141173	16.10	521.85
	SPINJ	7	22	0	141173	40.05	260.05
	Difference				0	2.49×	0.50×
msmie.4	SPIN	0	0	0	52287	14.50	523.71
	SPINJ	0	0	0	52287	43.69	260.05
	Difference				0	3.01×	0.50×
needham.4	SPIN	0	0	0	54	5.51	512.20
	SPINJ	0	0	0	40	15.17	256.21
	Difference				-14	2.75×	0.50×
protocols.5	SPIN	0	0	0	730301	7.61	596.58
	SPINJ	0	0	0	571812	20.25	287.07
	Difference				-158489	2.66×	0.48×
public_subscribe.2	SPIN	0	0	0	274039	6.06	583.59
	SPINJ	0	0	0	86341	16.70	287.08
	Difference				-187698	2.76×	0.49×

(continued on the next page...)

Table B.3 – Continued

Model		states missed	transitions missed	errors missed	depth	time (seconds)	memory (Mb)
szymanski.4	Spin	0	0	0	68686	4.03	523.90
	SpinJ	0	0	0	68686	11.05	260.05
	Difference				0	2.74×	0.50×

Table B.4: Results of all the model that were run using Spin and SpinJ. This benchmark was using run with Hash Compaction.

Model		states missed	transitions missed	errors missed	depth	time (seconds)	memory (Mb)
adding.6	Spin	0	0	0	99	3.98	401.31
	SpinJ	0	0	0	99	9.58	512.20
	Difference				0	2.41×	1.28×
at.4	Spin	0	0	0	851760	10.40	491.35
	SpinJ	0	0	0	851760	33.47	543.07
	Difference				0	3.22×	1.11×
bakery.6	Spin	0	0	0	1724631	15.90	608.93
	SpinJ	0	0	0	1724631	46.72	543.07
	Difference				0	2.94×	0.89×
bridge.2	Spin	0	0	0	125	36.80	530.22
	SpinJ	0	0	0	108	101.23	512.20
	Difference				-17	2.75×	0.97×
cambridge.4	Spin	0	0	0	114745	5.03	305.72
	SpinJ	0	0	0	108423	25.13	516.05
	Difference				-6322	5.00×	1.69×
elevator2.3	Spin	0	0	0	883189	17.80	494.38
	SpinJ	0	0	0	883189	57.24	543.07
	Difference				0	3.22×	1.10×
fischer.6	Spin	0	0	0	105361	15.00	427.99
	SpinJ	0	0	0	105361	53.74	516.04
	Difference				0	3.58×	1.21×
iprotocol.4	Spin	0	0	0	1218	9.27	353.28
	SpinJ	0	0	0	952	27.33	516.05
	Difference				-266	2.95×	1.46×

(continued on the next page...)

Table B.4 – Continued

Model		states missed	transitions missed	errors missed	depth	time (seconds)	memory (Mb)
krebs.4	Spin	0	0	0	163	28.20	837.68
	SpinJ	0	0	0	159	83.02	1024.19
	Difference				-4	2.94×	1.22×
lamport.6	Spin	0	0	0	141173	10.30	431.21
	SpinJ	0	0	0	141173	43.30	516.05
	Difference				0	4.20×	1.20×
msmie.4	Spin	0	0	0	52287	11.90	402.69
	SpinJ	0	0	0	52287	36.06	516.05
	Difference				0	3.03×	1.28×
needham.4	Spin	0	0	0	54	4.52	301.22
	SpinJ	0	0	0	40	12.13	512.21
	Difference				-14	2.68×	1.70×
protocols.5	Spin	0	0	0	730301	5.67	392.82
	SpinJ	0	0	0	571812	18.06	543.07
	Difference				-158489	3.19×	1.38×
public_subscribe.2	Spin	0	0	0	274039	4.73	371.72
	SpinJ	0	0	0	86341	14.64	543.08
	Difference				-187698	3.10×	1.46×
szymanski.4	Spin	0	0	0	68686	2.58	310.21
	SpinJ	0	0	0	68686	8.75	516.05
	Difference				0	3.39×	1.66×

Appendix C

Supported features by SpinJ

C.1 PROMELA language

The PROMELA language is supported by SpinJ, but not all its features. The following parts are supported and tested:

- Definition of `proctype` with parameters.
- Definition of the `init` process.
- The `never` claim.
- `mtype` definition.
- Global and local variables:
  - `bit`, `bool`, `byte`, `pid`, `short`, `int`, `mtype` types.
  - Channels, both rendez-vous and buffered.
- The following statements:
  - `XR` and `XS` statements.
  - `if`-selections.
  - `do`-loops.
  - Labels.
  - `goto`-statements, except inside `d_step`-sequences.
  - Channel send and receive statements.
  - `else`-statement.
  - Assignments.
  - Basic expressions.
  - `printf`-statements.
  - `assert`-statements.
  - `break`-statements.
  - `atomic` sequences.
  - `d_step` sequences.

C.2 Algorithms

SpinJ supports several different type of algorithms:

- Search algorithms:
  - Depth First Search
  - Breadth First Search
  - Nested Depth First Search
- Storage algorithms:
  - Probing hash table
  - Linked-list hash table
  - Array-list hash table
  - Bitstate hashing
  - Hash compaction
- Hash functions:
  - Jenkins Hash
  - Hsieh Hash
- Optimisation algorithms:
  - Partial Order Reduction
  - Statement merging

C.3 Command line options

The SpinJ promela compiler has the following command line options:

- `-j` In stead of Java code, it compiles it directly and packes it into a jar-file.
- `-n`*name* Sets the name of the model. By default the original filename is used to generate a name.
- `-o3` Disables statement merging during compilation.
- `-s`*dir* Sets the output directory for the source code. Default: `spin/generated`.

On runtime the following command line options can be given when running a model:

- `-A` Disables assertion checking.
- `-DARRAY` Uses a hash table with array lists.

- `-DBITSTATE` Uses bitstate hashing.
- `-DBFS` Uses breadth first search.
- `-DGRANDOM` Uses the random simulation.
- `-DGTRAIL` Uses the guided simulation.
- `-DGUSER` Uses the interactive simulation.
- `-DHC` Uses hash compaction.
- `-DNOREDUCE` Disables partial order checking.
- `-E` Ignores invalid end states.
- `-N` Disables the `never`-claim.
- `-a` Enables nested depth first search, to search for acceptance cycles.
- `-b` Exceeding the depth limit is considered and error in stead of a warning.
- `-c`N Stops the verification after N errors. When N is 0, it never stops.
- `-k`N Sets N bits per state when using bitstate hashing or hash compaction. For hash compaction this number must be a multiple of 8.
- `-m`N Sets the maximum search depth.
- `-v` Prints the version and exists.
- `-w`N Sets the number of hash entries in the state storage to 2^N.

Bibliography

[1] *Spinj manual*, http://spinroot.com/spin/Man/Manual.html.

[2] *Acm cites tool to detect software "bugs" for prestigious award*, march 2002, http://www.acm.org/announcements/ss_2001.html.

[3] *Acm turing award honors founders of automatic verification technology*, february 2008, http://www.acm.org/press-room/news-releases/pdfs/turing07.pdf.

[4] ISO 12207, *International standard, information technology software life cycle process. iso 12207*, 1995.

[5] Rajeev Alur and David L. Dill, *A theory of timed automata*, Theoretical Computer Science **126** (1994), no. 2, 183–235.

[6] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton, *Model-checking continuous-time markov chains*, ACM Trans. Comput. Logic **1** (2000), no. 1, 162–170.

[7] Christel Baier and Joost-Pieter Katoen, *Principles of model checking*, The MIT Press, 2008.

[8] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček, *DiVinE – A Tool for Distributed Verification (Tool Paper)*, Computer Aided Verification, LNCS, vol. 4144/2006, Springer Berlin / Heidelberg, 2006, pp. 278–281.

[9] Guillaume Brat, Klaus Havelund, Seungjoon Park, and Willem Visser, *Model checking programs*, In IEEE International Conference on Automated Software Engineering (ASE, 2000, p. 2000.

[10] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, *Nusmv: A new symbolic model verifier*, Springer, 1999, pp. 495–499.

[11] Cliff Click, *A lock-free hash table*, may 2007, http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf.

[12] Peter Dillinger and Panagiotis Manolios, *Fast and accurate bitstate verification for spin*, Proceedings of the 11th SPIN Workshop (2004).

[13] Doron A. Peled Edmund M. Clarke, Orna Grumberg, *Model checking*, ch. 10, MIT Press, 1999.

[14] Cormac Flanagan and Patrice Godefroid, *Dynamic partial-order reduction for model checking software*, Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages **40** (2005), no. 1, 110-121.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns*, Addison-Wesley, Boston, 1995, 0201633612.

[16] Doron A. Peled Gerard J. Holzmann, *An improvement in formal verification*, Proceedings of the FORTE 1994 Conference **4** (1994), no. 7.

[17] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper, *Simple on-the-fly automatic verification of linear temporal logic*, Protocol Specification Testing and Verification (Warsaw, Poland), Chapman & Hall, 1995, pp. 3–18.

[18] Gerard J. Holzmann, *Algorithms for automated protocol validation*, AT&T Technical Journal **69** (1988), no. 2, 32-44.

[19] ———, *The model checker spin*, IEEE Transactions on software engineering **23** (1997), no. 5, 279-295.

[20] ———, *An analysis of bitstate hashing*, Tech. report, Formal Methods in Systems Design, 1998.

[21] ———, *The spin model checker: Primer and reference manual*, 2005.

[22] Gerard J. Holzmann and Dragan Bosnacki, *The design of a multi-core extension of the spin model checker*, 2007.

[23] Gerard J. Holzmann and Rajeev Joshi, *Model-driven software verification*, In Proc. 2001 ACM SIGPLANSIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE01, ACM Press, 2004, pp. 80–89.

[24] Gerard J. Holzmann and Anuj Puri, *A minimized automaton representation of reachable states*, International Journal on Software Tools for Technology Transfer **2** (1999), no. 3, 270–278.

[25] G.J. Holzmann, D. Peled, and M. Yannakakis, *On nested depth-first search*, The Spin Verification System (1996), 23–32.

[26] Paul Hsieh, *Hash functions*, http://www.azillionmonkeys.com/qed/hash.html.

[27] Michael R. A. Huth and Mark D. Ryan, *Logic in computer science: Modelling and reasoning about systems*, Cambridge University Press, Cambridge, England, 2000.

[28] Bob Jenkins, *A hash function for table lookup*, http://burtleburtle.net/bob/hash/doobs.html.

[29] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled, *Model checking*, The MIT Press, 1999.

[30] Heinz Kabutz, *What is faster - `LinkedList` or `ArrayList`?*, 7 2005, http://www.javaspecialists.eu/archive/Issue111.html.

[31] M.A. Kattenbelt, *Towards an explicit-state model checking framework*, Master's thesis, University of Twente, EWI, Formal Methods & Tool group, 2006.

[32] M.A. Kattenbelt, T.C. Ruys, and A. Rensink, *An object-oriented framework for explicit-state model checking*, Proceedings of the 3rd European Symposium on Verification and Validation of Software Systems, 2007.

[33] Matthias Kuntz and Kai Lampka, *Probabilistic methods in state space analysis*, Validation of Stochastic Systems, 2004, pp. 339–383.

[34] K. L. Mcmillan, *The smv system - for smv version 2.5.4*, 2000, http://www.cs.cmu.edu/ modelcheck/smv/smvmanual.ps.gz.

[35] Stephan Merz, *Model checking: A tutorial overview*, MOVEP, 2000, pp. 3–38.

[36] R. Pelánek, *Beem: Benchmarks for explicit model checkers*, http://anna.fi.muni.cz/models/.

[37] ———, *Beem: Benchmarks for explicit model checkers*, Proc. of SPIN Workshop, LNCS, vol. 4595, Springer, 2007, pp. 263–267.

[38] Malcolm Wallace, *hi - hmake interactive - compiler or interpreter*, Proceedings of the 12th International Workshop on Haskell (2000).

[39] Michael Weber, *An embeddable virtual machine for state space generation.*, SPIN (Dragan Bosnacki and Stefan Edelkamp, eds.), Lecture Notes in Computer Science, vol. 4595, Springer, 2007, pp. 168–186.

[40] Pierre Wolper and Dennis Leroy, *Reliable hashing without collision detection*, Proc. 5th Int. Conference on Computer Aided Verification (1993), 59-70.