Tool Presentation

The SpinJa Model Checker[⋆]

Marc de Jonge[1] and Theo C. Ruys[2]

[1] TNO, The Netherlands, marcdejonge@gmail.com
[2] RUwise Consultancy, Deventer, The Netherlands, ruys@mac.com

Abstract. SpinJa is a model checker for promela, implemented in Java. SpinJa is designed to behave similarly to Spin, but to be more easily extendible and reusable. Despite the fact that SpinJa uses a layered object-oriented design and is written in Java, SpinJa's performance is reasonable: benchmark experiments have shown that, in exhaustive mode, SpinJa is about five times slower than the highly optimized Spin. For bitstate verification runs the difference is only a factor of two.

1 Introduction

The Spin model checker [2,10] is arguably one of the most powerful and popular (software) model checkers available. Unfortunately, the highly optimized C code of Spin is difficult to understand and hard to extend. Moreover, the algorithms implemented in Spin cannot be reused in other tools.

This paper presents SpinJa[†] [11], a model checker for promela, implemented in Java. SpinJa is designed to behave similarly to Spin, but to be more easily extendible and reusable.

Related Work. Independently from SpinJa, Mordechai Ben-Ari has started the development of the model checker Erigone [1,8], a model checker for (a subset of) promela, implemented in Ada. Erigone's goal is to facilitate *learning* concurrency and model checking. nips [6,9] is another attempt to provide a stand-alone alternative model checker for promela. nips uses the virtual machine approach to model checking. Both Erigone and nips do not (yet) implement any of Spin's powerful optimization algorithms (e.g., partial order reduction, bitstate hashing, hash compaction) though.

Sec. 2 explains the design and implementation of SpinJa. Sec. 3 reports on a benchmark experiment with SpinJa and Spin. Sec. 4 concludes the paper and discusses future work.

2 SpinJa

SpinJa can be used to check for the absence of deadlocks, assertions, liveness properties and never claims in promela models. SpinJa's verification mode can

[⋆] The first version of SpinJa has been developed as part of a MSc project within the Formal Methods & Tools group of the University of Twente, The Netherlands.

[†] SpinJa stands for '*Sp*in *in Ja*va'. SpinJa rhymes with *ninja*.

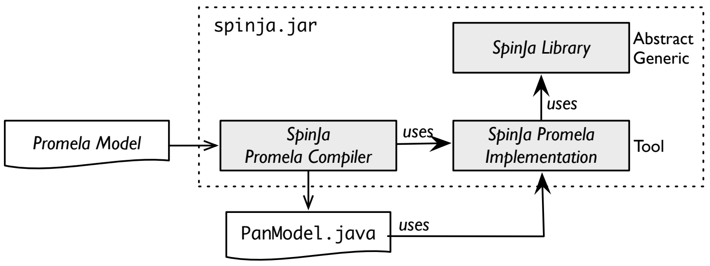

Fig. 1. Architecture of SpinJa.

exploit (nested) depth first search (DFS) or breadth first search (BFS). Bitstate hashing and hash compaction modes are also supported. Furthermore, Spin's partial order reduction and statement merging are implemented. SpinJa can also be used for simulation. To enable or disable features, SpinJa uses the same syntax for options as Spin. And the output of SpinJa also mimics Spin: seasoned Spin users should feel right at home.

SpinJa supports a large subset of the PROMELA language. Currrently, the following aspects of the PROMELA language are not yet supported: the `unless` statement, `typedef` definitions, non-FIFO communication, and embedded C code.

Design

The research question behind the development of SpinJa was the following: 'Is it possible to re-implement the core of Spin in Java using a extendible object-oriented design while being competitive in terms of memory consumption and runtime behaviour?' Beforehand – given the folklore about Java's infamous lack of speed, compared to C/C++ – we would have been happy if SpinJa would turn out to be a single order of magnitude slower than Spin.

From the start we have committed ourselves to Java and a clean object-oriented approach. This section only sketches the design of SpinJa; please refer to [3] for a detailed description.

Fig. 1 shows the high-level architecture of SpinJa. SpinJa follows the same principle as Spin: given a PROMELA model as input, a verifier program is generated: `PanModel.java`. The SpinJa library, which is used by `PanModel.java`, consists of two parts: a PROMELA specific part and a generic part. The complete SpinJa package (i.e., compiler and library) is supplied as a single Java jar-file.

For the design and implementation of SpinJa we have adopted the conceptual framework for explicit-state model checkers as developed by Mark Kattenbelt et.al. [5]. This layered framework is more flexible than the 'black box' approach of conventional model checkers. Fig. 2 shows the layered model as implemented in SpinJa. Each layer extends the layer below it and adds more functionality. This is achieved through inheritance. This way the different parts of the model checker are loosely coupled. This increases the maintainability and reusability of the whole system.

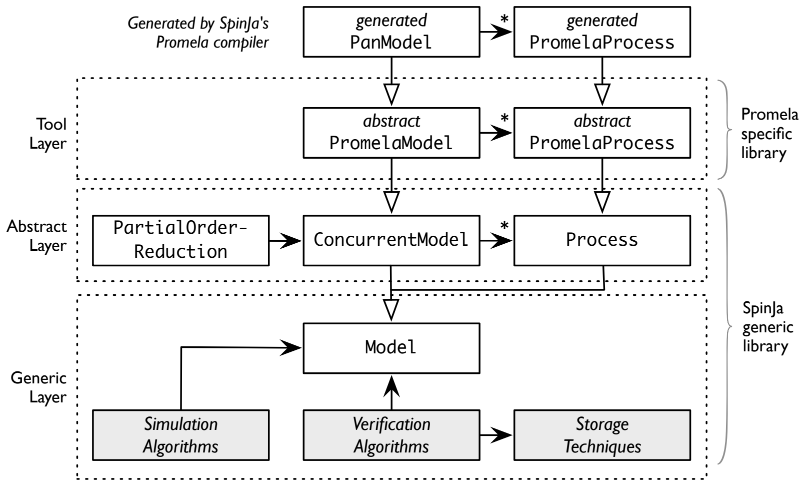

Fig. 2. Layered design of SpinJa.

Generic layer. The Generic layer is the lowest layer. A `Model` is responsible for generating all the states at run-time. The `Model` knows how to take `Transitions` from one state to the next. On the basis of just `State` and `Transition` objects, several important algorithms are implemented within the generic layer: search (DFS, BFS) and simulation algorithms, storage methods, hashing methods.

Abstract layer. The Abstract layer extends on the Generic layer by adding concurrency to the model: a `ConcurrentModel` is a collection of `Process`es which are `Model` objects themselves. The class `PartialOrderReduction` changes the way the set of successor transitions of a `ConcurrentModel` are being computed. The Generic and Abstract layers are completely independent from the PROMELA language and could serve as the basis for any explicit-state model checker.

Tool Layer. Within the Tool layer the abstract `PromelaModel` is defined which forms the blueprint for the generated Java verifier `PanModel`. The `PromelaModel`, for example, knows how to create and terminate PROMELA proctypes and how to add `Channel` objects.

Compiler. The SpinJa PROMELA compiler is responsible for translating the PROMELA specification to the generated Java program: `PanModel`. Much effort has been put into implementing the same semantics for PROMELA as Spin does: in almost all cases, the resulting automata of the SpinJa compiler are the same as the ones that Spin produces. Consequently, for most PROMELA models, the number of states and transitions traversed by SpinJa and Spin are the same.

Implementation. The current version of SpinJa is version 0.9. SpinJa is released as open-source under the Apache 2.0 license and is available from [11]. The development took roughly one man year of work (i.e., [4] and [3]). The code base of SpinJa 0.9 consists of 144 files, 25k loc and its size is 1100Kb. By comparison,

verification mode	⌀ *states*	⌀ *transitions*	⌀ SpinJa	⌀ Spin no -O2	⌀ Spin with -O2	SJa/(S no -O2)	SJa/(S with -O2)
Exhaustive, with POR	4.2×10^6	19.9×10^6	38.7	14.8	8.63	2.6×	4.5×
Exhaustive, no POR	4.2×10^6	20.2×10^6	38.4	13.9	8.06	2.8×	4.8×
Bitstate Hashing (with POR)	4.2×10^6	19.9×10^6	21.3	17.0	10.67	1.3×	2.0×
Hash Compaction (with POR)	4.2×10^6	19.9×10^6	20.5	14.7	8.56	1.4×	2.4×

Table 1. Summary of benchmark experiments: SpinJa vs. Spin (*no* `-O2` and *with* `-O2`). Running times are in seconds.

Spin 5.2.4 consists of 48 files, 48k loc and is 1200Kb and Erigone 2.1.5 consists of 71 files, 8k loc and is 300Kb.

3 Experiments

As mentioned in Sec. 2, we are interested in how a Java verifier generated by SpinJa compares with the C verifier generated by Spin with respect to time and memory consumption. In this section we report on a benchmark experiment. For this paper, we have used ten medium size PROMELA models from the BEEM benchmark suite [7]: *at.4*, *elevator2.3*, *fischer.6*, *hanoi.3*, *loyd.2*, *mcs.3*, *peterson.4*, *sorter.3*, *szymanski.4* and *telephony.4*.

We compared Spin against SpinJa in (i) a default exhaustive safety run, (ii) with partial order reduction (POR) disabled, (iii) in bitstate hashing mode (with `-k3 -w31`), and (iv) in hash compaction mode. We considered two versions of Spin's `pan` verifier: (i) default compilation without optimizations and (ii) a compilation with `gcc`'s `-O2` optimization option set. All runs were limited to 1Gb of memory and a depth of 1.2×10^6 steps. All models could be verified completely except for *hanoi.3* which requires an even larger depth. The experiments were run on an Apple MacBook 2.4Ghz Intel Core 2 Duo, with 4Gb of RAM, running Mac OS X 10.5.8, with Spin 5.2.4, SpinJa 0.9, `gcc` 4.2.1 and Java 1.6.0_17.

Table 1 summarizes the *averages* (i.e. ⌀) of running the verifiers: the number of states, the number of transitions and the running time (i.e., *user time*) of the verifiers in the various verification modes. Generation and compilation time have not been taken into account. The last two columns list the average speed difference factor between SpinJa and Spin *without* `-O2`, and SpinJa and Spin *with* `-O2`. Several important observations can be made from Table 1:

– Spin's performance profits a lot from `gcc`'s `-O2` optimization option.
– Using Java for the implementation of a model checker seems acceptable: without `-O2`, Spin is less than three times faster than SpinJa.
– SpinJa is surprisingly close to Spin for the two approximate verification modes: the difference in speed is only a factor of two.

We do not report on the memory consumption of the verifiers in Table 1. The reason for this is that there is no substantial difference between SpinJa and Spin with respect to memory. In most cases, SpinJa needs slightly less memory than Spin though.

For a more extensive benchmark comparison between Spin 5.1.6 and SpinJa 0.8 (under Windows XP and using Microsoft's C compiler) we refer to [3].

4 Conclusions

In this paper we presented SpinJa, a model checker for promela, implemented in Java. Due to its layered, object-oriented design we believe that the design and implementation of SpinJa is clear, readable and extensible. Compared to the highly optimized C-based Spin, the Java-based SpinJa is clearly slower. For bitstate hashing mode, however, the difference is only a factor of two. To extend the usability of SpinJa further, several improvements are planned:

- Support for the missing promela features.
- Support for modelling languages other than promela.
- Implementation of Spin's `-DREVERSE`, `-DP_RAND`, `-DT_REVERSE` and `-DT_RANDOM` to increase the coverage of bitstate hashing verification runs.
- Improvement of SpinJa's explicit-state storage algorithms.
- Support for embedding Java code (similar to the support for C code in Spin); this would make it possible to use SpinJa to model check Java applications.
- Development of a code generator for promela on basis of SpinJa's generated verifier code.

References

1. M. Ben-Ari. Teaching Concurrency and Model Checking. In C. S. Pasareanu, editor, *Proc. of SPIN 2009*, LNCS 5578, pages 6–11. Springer, 2009.
2. G. J. Holzmann. *The* Spin *Model Checker – Primer and Reference Manual.* Addison-Wesley, Boston, Massachusetts, USA, 2004.
3. M. de Jonge. The SpinJ Model Checker. Master's thesis, University of Twente, Enschede, The Netherlands, September 2008.
4. M. A. Kattenbelt. Towards an Explicit-State Model Checking Framework. Master's thesis, University of Twente, Enschede, The Netherlands, September 2006.
5. M. A. Kattenbelt, T. C. Ruys, and A. Rensink. An Object-Oriented Framework for Explicit-State Model Checking. In *Proc. of VVSS 2007, Eindhoven, The Netherlands, March 2007*, TUE Computer Science Reports, Nr. 07-04, pages 84–92, 2007.
6. M. Weber. An Embeddable Virtual Machine for State Space Generation. In D. Bosnacki and S. Edelkamp, editors, *Proc. of SPIN 2007*, LNCS 4595, pages 168–185. Springer, 2007.
7. BEEM: BEnchmarks for Explicit Model checkers. `http://anna.fi.muni.cz/models/`.
8. Erigone. `http://code.google.com/p/erigone/`.
9. nips. `http://www.ewi.utwente.nl/~michaelw/nips/`.
10. Spin. `http://spinroot.com/`.
11. SpinJa. `http://code.google.com/p/spinja/`.

Presentation Plan

SpinJa is a command-line program which is executed from a terminal window. As such, a demo with SpinJa will *not* be a breathtaking visual experience. Still, we believe that a demonstration with SpinJa should be interesting and helpful for people that want to experiment with the tool.

We plan to split the presentation into three parts:

1. (50%) an introduction to the *design* of SpinJa, explaining its high-level, object-oriented architecture,
2. (20%) a *demo* showing how SpinJa works, followed by
3. (30%) an *explanation* of the Java application that is being generated by the SpinJa promela compiler.

1. Architecture of SpinJa (50%)

Seen from the user's perspective, SpinJa might be just an alternative implementation of Spin. This is not the real strength of SpinJa though.

We believe that the strength and promise of SpinJa lie within its extendible and reusable design. Therefore, the main part of the presentation will be devoted to explaining the (rationale behind the) layered, object-oriented design of SpinJa. We will show that the SpinJa library is general enough to form the basis for any explicit model checker.

2. Demo (20%)

In the demo we will show how SpinJa should be used. We will discuss a small promela model (without errors) and show how to verify this promela model with SpinJa.

We will also demonstrate SpinJa's simulation mode by simulating the promela model using SpinJa's random and interactive simulation modes.

3. Explanation of generated Java application (30%)

After the demo we will dive into the generated Java code (`PanModel.java`) of the promela model of the demo. We will show that this Java code is human readable and easy to understand. We will explain the connection between the generated model and SpinJa's library.

Availability

Both the source and binary distribution of SpinJa are available from [11].