The State of SPIN

Gerard J. Holzmann and Doron Peled

Bell Laboratories
700 Mountain Avenue
Murray Hill, NJ 07974
{gerard, doron}@research.bell-labs.com

Abstract. The number of installations of the SPIN model checking tool is steadily increasing. There are well over two thousand installations today, divided roughly evenly over academic and industrial sites. The tool itself also continues to evolve; it has more than doubled in size, and hopefully at least equally so in functionality, since it was first distributed in early 1991. The tool runs on most standard workstations, and starting with version 2.8 also on standard PCs.

In this overview, we summarize the design principles of the tool, and review its current state.

1 Background

SPIN is a general state-based model-checking tool designed for the efficient verification of logically distributed process systems. Processes in SPIN are always *asynchronous*. Synchronization, where desired, must be specified explicitly.

The native specification language of SPIN is called PROMELA. PROMELA is a non-deterministic guarded command language, in the tradition of [3] and [5] with a small influence from the language **C** [11]. The language was designed to encourage abstraction. The purpose of model checking in SPIN is to perform *design verification* well before the coding stage of a design is reached. A basic notion in the language is that of *executability*: every PROMELA statement can enforce synchronization constraints through the rules of executability. Whenever a statement is unexecutable, for instance, it blocks the execution of the corresponding process, unless alternative executions for that process were specified. The most recent version of the language supports data structures, interrupts, and a rich variety of both synchronous and asynchronous message passing primitives.

The semantics of a PROMELA model are based on the *interleaving model* of execution, where concurrently executed atomic operations from different processes are considered to be executable in any arbitrary time-order. The model is appropriate for modeling distributed software. For synchronous hardware a different semantics interpretation is usually chosen, e.g. [12]. Most model checkers today have adopted those alternative semantics. These systems can still simulate interleaving semantics at the language level, but the price to pay for this in efficiency can be substantial. The optimizations builtin to SPIN fully exploit the asynchronous process model.

1.1 State-based model-checking

SPIN's verification procedure is based on the reachability analysis of a model, using an optimized *depth-first-search* graph traversal method. A number of special-purpose algorithms are used to avoid a purely exhaustive search procedure (e.g., partial order reduction, state compression, and sequential bitstate hashing). We summarize some of the newer algorithms in the sequel.

1.2 Correctness properties

SPIN can verify both safety and liveness properties *on-the-fly*. By default, SPIN will check a set of basic properties such as absence of *deadlock* and *unreachable code*. It will also check that any user-defined *process assertions* or *invariants* cannot be violated, and that the system can only terminate in user-defined valid end-states.

The specification language includes two types of labels that can be used to define two complementary types of liveness properties: acceptance and progress. In the syntax of Linear Temporal Logic (LTL) [15], an acceptance property corresponds to formulae of the type $\Box\Diamond p$, where p is a user-defined accepting state. The violation of a progress property corresponds to formulae of the type $\Diamond\Box\neg p$ with p a user-defined progress state.

Correctness requirements can also be expressed directly in LTL syntax. For example, the formula $\Box(request \rightarrow \Diamond granted)$ asserts that at any point in the execution, if a request was made, it is eventually granted. SPIN versions 2.7 and later include a translation algorithm that converts LTL formulae like these into PROMELA *never-claims*. Never-claims formalize the potential violations of a correctness requirement, i.e., behavior that should *never* happen. More specifically, a never-claim can be used to represent a Büchi automaton (an automaton over infinite words), and it is this capability that is exploited by the LTL translator.

Although the expressive power of LTL is smaller than that of never-claims [17], the use of LTL can be simpler and more direct.

2 Basic Algorithms

2.1 Automata Intersection

SPIN uses finite automata based model-checking. Each process of the checked model is translated into a finite automaton. The checked property, representing the *violations* of correctness properties, is translated into a *property* automaton (i.e., the never-claim).

SPIN checks the given model against the given property by calculating the intersection of the corresponding automata. This is done *on-the-fly*, namely, the state space of the model intersected with the property automaton need only be built up to the point where the non-emptiness of the resulting automaton can be proven. A non-empty intersection means the possibility of violating a correctness requirement. An execution sequence that illustrates this is produced, which the

user can use to retrace the error, for instance, as a message sequence chart in XSPIN (an independent graphical user interface for SPIN, written in Tcl/Tk).

The computation of the intersection can either be done in a conventional exhaustive manner, or, when this proves to be impossible because of state space explosion, with an efficient proof approximation method based on bitstate hashing [6]. With a careful choice of hashing functions [9], the probability of an exhaustive proof remains very high. Both the exhaustive and the bistate modes of verification are based on a partial order reduction theory that limits the work to a small subset of states that suffices to render the proof (see also Section 2.3 below).

2.2 Cycle Detection

The model-checking is performed by checking the non-emptiness of the intersection of Büchi automata. The non-emptiness of the resulting automaton is examined by checking for the existence of at least one reachable cycle that contains an accepting state. Such a cycle constitutes a counterexample to the claim that no executions can exist that violate a correctness requirement.

The classical algorithm for finding a cycle in a graph is Tarjan's depth-first-search, which constructs the strongly connected components in linear time. If a reachable component contains an accepting state, a reachable acceptance cycle must exist. To find such a cycle requires searching the component for the accepting states, and constructing a path through at least one of them. An efficient alternative for this algorithm [7, 2] performs a nested depth-first-search, again visiting every state up to two times, but storing every state only once (with just two bits of overhead per state). The algorithm starts the search in the normal depth-first-order. Whenever it finishes processing an accepting state (i.e., in postorder), a search for a cycle is started in a logically separate statespace. SPIN uses a variant of this search procedure to prove acceptance and non-progress properties [10].

2.3 Partial Order Reduction

SPIN uses a *partial order reduction* algorithm [8, 13, 14] to reduce the state space explosion. The reduction is based on the observation that usually the checked property is insensitive to the order in which concurrent or independently executed events are interleaved. Thus, instead of generating a state-space which includes all the execution sequences, one can generate a reduced state space, with only representatives for classes of sequences that are undistinguishable from each other.

Reduction is achieved by changing the depth first search algorithm, used in the model-checking engine, such that from each state, only a sufficiently big subset of the successors are generated. This subset has to obey certain restrictions, guaranteeing that enough representative execution sequences is generated. The implementation is based on a careful analysis of the enabled atomic transitions from the currently analyzed state. To achieve a small overhead for doing the

extra checks, most of the analysis is done at the time of compiling the checked model. However, some reduction decisions are necessarily deferred to run time. The reduction algorithm as implemted was proven to preserve all safety and liveness properties [8, 1].

2.4 Linear Temporal Logic to Automata Translation

The translation of an LTL formula into an automaton is inherently exponential [16]. SPIN uses an algorithm [4] that in practice tends to produce small automata. The translation algorithm computes the states of an automaton by compiling the set of subformulas that need to hold in each state, and in each of its successors. The algorithm starts by converting the formula into a normal form, in which negations are pushed inward, to appear adjacent to only atomic propositions. An initial state is created, containing the formula to be translated and a dummy incoming edge. The automaton is then computed recursively, At each stage, a subformula that remains to be satisfied is taken and, according to the main logical operator, the current state may be split into two, with the two copies inheriting different parts of the subformula. In the last phase of the translation, some of the states are identified as accepting according to the presence or absence of subformulas of the type $\varphi U \psi$.

3 User Support

All the sources to the SPIN system and its graphical interface XSPIN are available for general educational use, free of charge, and for industrial applications for a small licensing fee. The latest version of the tool is always available via anonymous ftp from `netlib.att.com` from the directory `/netlib/spin`. Much of the documentation for SPIN is distributed together with the tool. The main reference on the implementation of the tool itself is the book [7]. The recent extensions, including the partial order reduction, and the LTL translation algorithm are reported in research papers [4, 8, 9]. A new book describing the automata theoretic background of SPIN is in preparation.

An informal SPIN *Users Group* was formed early in 1995. Anyone interested in the background of the tool, major applications, and extensions that are being planned, can subsribe to the mailing list of the group by sending a one line message `Subscribe` to `spin_list@research.bell-labs.com`.

A first SPIN *Workshop* was held in Montreal on 16 October 1995. The proceedings are available via: `http://netlib.att.com/netlib/spin/news.html`. A second workshop is held 5 August 1996, at Rutgers University.

References

1. C-T. Chou, D. Peled, Verifying a Model-Checking Algorithm, TACAS'96, Tools and Algorithms for the Construction and Analysis of Systems, Passau, Germany, March 1996.

2. C. Courcoubetis, M. Vardi, P. Wolper, M, Yannakakis, Memory-efficient algorithms for the verification of temporal properties, Formal methods in system design 1 (1992) 275–288.
3. E. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM, 18(8), 1975, 453–457.
4. R. Gerth, D. Peled, M.Y. Vardi, P. Wolper, Simple On-the-fly Automatic Verification of Linear Temporal Logic, PSTV95, Protocol Specification Testing and Verification, Warsaw, Poland. Chapman & Hall, Germany, 1995, 173–184.
5. C.A.R. Hoare, Communicating Sequential Processes, Comm. ACM, 21(8), 1978, 666–677.
6. G.J. Holzmann, An Improved Protocol Reachability Analysis Technique, Software Practice and Experience, Feb 1988, Vol 18, No 2, pp. 137-161.
7. G.J. Holzmann, Design and Validation of Computer Protocols, Prentice Hall, 1992.
8. G.J. Holzmann, D. Peled, An Improvement in Formal Verification, $7th$ Int. Conf. on Formal Description Techniques, Berne, Switzerland, 1994, 177–194.
9. G.J. Holzmann, An Analysis of Bitstate Hashing, PSTV95, Protocol Specification Testing and Verification, Warsaw, Poland, Chapman & Hall, Germany, 1995, 301–314.
10. G.J. Holzmann, D. Peled, M. Yannakakis, On Nested Depth-First Search, In preparation, 1996.
11. B.W. Kernighan, D.M. Ritchie, The C programming Language, Prentice Hall, 1988.
12. R.P. Kurshan, Computer-Aided Verification of Coordinating Processes, Princeton University Press, 1994.
13. D. Peled, Combining Partial Order Reductions with On-the-fly Model-Checking, Proc. CAV'94, 6th International Conference on Computer Aided Verification, LNCS 818, Springer-Verlag, 377-390, 1994, Stanford CA, USA.
14. D. Peled, Partial Order Reduction: Model-Checking using Representatives, Proc. MFCS'96, 21st International Symposium on Mathamatical Foundations of Computer Science, September 1996, Cracow, Poland.
15. A. Pnueli, The temporal logic of programs, Proc. of the $18th$ IEEE Symp. on Foundation of Computer Science, 1977, 46-57.
16. M.Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, Proc. of the $1st$ Symposium on Logic in Computer Science, 1986, Cambridge, England, 322–331.
17. P. Wolper, Temporal Logic Can be More Expressive, Information and Control 56 (1983), 72–99.