SPIN model checking: An introduction

Article *in* International Journal on Software Tools for Technology Transfer · March 2000

DOI: 10.1007/s100090050039 · Source: DBLP

3 authors:

Gerard Holzmann
California Institute of Technology
210 PUBLICATIONS **14,642** CITATIONS
SEE PROFILE

Elie Najm
Institut Mines-Télécom
59 PUBLICATIONS **375** CITATIONS
SEE PROFILE

Ahmed Serhrouchni
Institut Mines-Télécom
123 PUBLICATIONS **230** CITATIONS
SEE PROFILE

Int J STTT (2000) 2: 321–327

Special section on SPIN

SPIN model checking: an introduction

Gerard Holzmann[1], **Eli Najm**[2], **Ahmed Serhrouchni**[2]

[1] Bell Laboratories, MH 2C-521, 700 Mountain Avenue, Murray Hill, New Jersey 07974, USA;
E-mail: gerard@research.bell-labs.com
[2] Ecole Nationale Supérieure des Télécommunications, 46, rue Barrault, 75013 - Paris, France

1 Introduction

A long-standing and elusive problem in software engineering is to devise reliable means that would allow us to check the correctness of distributed systems code mechanically. Writing reliable distributed code is notoriously difficult; locating the inevitable bugs in such code is therefore important. As is well-known and often repeated, traditional testing methods are of little use in this context, because the most pernicious bugs typically depend on subtle race conditions that produce peculiar and unexpected interleavings of events.

Well-known are the deadlock and starvation problems that plagued the designers of the first distributed systems code in the 1960s and 1970s (see for instance [16] p.155). In simple cases, a small set of strictly enforced rules can preserve the sanity in a system. One such rule is the requirement that frequently used resources in an operating system can only be allocated in a fixed order, to prevent circular waiting. But the simple rules only cover the known problems. Each new system builds a new context, with its own peculiarities and hazards. This is illustrated by the well-publicized description of the hangup problem in the control software of the Mars Pathfinder a few years ago [11]. In retrospect, the hangup scenario could be understood in very simple terms, yet it was missed even in the long and unusually thorough (but traditional) software testing process that had been used.

2 Software model checking

The first attempts to develop automated techniques to analyze simple versions of distributed systems, starting in the late 1970s, focused almost exclusively on data communication protocols. The reason is that protocol descriptions were from the beginning typically expressed in graph-like or statemachine-like notations, rather than in general programming languages, which favored experiments with the construction of simple automated tools. As far as we can tell, the first person to build a general purpose verification system for protocols was Jan Hajek, then working at the University of Eindhoven in The Netherlands [4]. Hajek published little about the algorithms that he used to build his *Approver* tool, though, so we known little more than that it used graph algorithms and some heuristics to identify undesirable reachable states (nodes in the system graph) and undesirable executions sequences (finite and infinite paths in the graph). Andy Tanenbaum confirms [15] that Hajek used his system to check the correctness of the protocols that appeared in the first edition of the now classic *Computer Networks* book [14].

In the next few years many others followed, most of whom developed specialized formalisms for expressing protocol behaviors, but generally performing a comparable type of analysis. In our own group at Bell Labs we built a first version of a reachability analyzer in 1980 and successfully applied it to a local model of a telephone switch to find design errors [5]. That system, called PAN, is the earliest ancestor of today's SPIN model checker, but other than that it used an on-the-fly verification technique rooted in a depth-first exploration of a system's graph, of course, it no longer has much in common with today's SPIN.

Over the years, the tools, like PAN, that were built for reachability analysis of communication protocols evolved into full-fledged automata-based model checking systems addressing the more general area of distributed systems software. The origin and evolution of tools like SPIN is markedly different from that of the classic logic model checkers pioneered by Ed Clarke and his students at CMU in the USA, e.g., [1] and Joseph Sifakis and students at Grenoble University in France, e.g., [10]. Initially, sys-

tems such as PAN were restricted to the verification of safety properties and a very small class of liveness properties (such as proving the absence of certain types of non-progress cycles). We made no attempt to extend the logic beyond this small class of correctness properties for two basic reasons:

1. The smaller class appeared sufficient at the time for the types of applications that we applied the reachability analyzers to (i.e., relatively simple protocol models and models of mutual exclusion algorithms).
2. The computational expense of even these simple types of checks could easily swamp the resources of the best computers in the early 1980s. (Several orders of magnitude slower and smaller than the average desktop PC today.)

Any attempt to increase the expressive power of the logic immediately translated into huge computational expenses that made it all but impossible to apply a tool to anything other than the most trivial exercises, as the first builders of full-fledged logic model checkers also soon discovered. By virtue of our pragmatic restrictions, we were able to analyze respectably sized problems with the PAN system, reaching several millions of reachable states on machines that would be considered unusable even as toys today. Not many other tools could match this behavior.

From then on the advances in performance, especially hardware, changed all the rules in the late 1980s. In 1989 we extended the expressive power for the SPIN system to that of omega-regular properties (a set that includes Linear Temporal Logic as a special case) to take full advantage of the more powerful processors. Since then, further algorithmic improvements have been made (e.g., search reduction techniques and state compression methods) and the hardware performance improvements have continued. The result is that today most of us have access to a range of very powerful model checking engines for software verification. The ones we are interested in are based, like the first reachability analyzers from the early 1980s, on automata and graph theory, not on logic systems. SPIN is generally considered to be the leading example of a highly efficient model checker in this group.

This evolution of the automata-based systems may be contrasted with that of classic logic-based model checkers with, as its leading example, the well-known SMV model checker developed at CMU [8]. Most people are not aware that systems like SMV and SPIN are based on different types of theories and have evolved quite independently over the last two decades. The most notable difference between these tools is the type of application for which they were built and optimized: SMV is primarily built for hardware verification [8]; SPIN is built for software verification [6]. Internally the tools are almost incomparable. SMV is based on logic, uses BDD encodings for the transition relations, and it applies a basically breadth-first graph exploration technique to solve the model checking problem for the *branching* time logic CTL; SPIN is based on automata theory, it uses an explicit depth-first search for omega regular properties, with as its most prominent subset the *linear* time logic LTL [9].

Due to the differences in their domains of application, logics, and modeling languages, it is hard to make accurate comparisons between these tools. Those who have tried have typically concluded that for comparable applications, the tools are very competitive, neither tool systematically outperforming the other, e.g., [2]. Despite this there are long-standing myths that hold that either LTL or CTL model checking would necessarily be more efficient, or that CTL or LTL would be a more adequate logic to express correctness properties. In practice, both tools are virtually undefeatable within their own domain of application. For SPIN this is the verification of the interactions of asynchronously executing concurrent processes in a distributed system.

3 SPIN based model checking in a nutshell

SPIN implements what has become known as the Vardi-Wolper framework for automata-theoretic verification [12]. Curiously, it was not a deliberate plan to build SPIN to conform to this methodology, but more a discovery that it could easily do so after the tool was almost completed. SPIN and most of its predecessors are implementations of standard automata-based reachability analyzers, with primary emphasis on performance. When the implementation of SPIN was nearing completion in early 1989, it had support for two new types of correctness claims, in addition to the by then standard support for proving absence of deadlock and absence of non-progress cycles. We had added two new types of claims. One was used to express *positive* correctness requirements (behaviors required to be present) and was named an `always` claim. The other was used for expressing *negative* correctness requirements (behaviors required to be absent) and was named a `never` claim. A former colleague at Bell Labs, Costas Courcoubetis, quickly noted that the functionality of the `never` claim could be explained in terms of Vardi-Wolper's automata-theoretic method and sufficed for expressing LTL, and even the richer set of all omega regular properties in which LTL is contained. After a short discussion we decided to omit the experimental `always` claims, and to adopt the Vardi-Wolper theory as a basis for the model checker. By doing so, SPIN succeeded in combining an efficient implementation of a classic reachability analyzer with a firm and well-understood theory for LTL model checking. The combination is likely responsible for the remarkable appeal of SPIN in both academia and industry.

SPIN accepts specifications written in a meta-language named PROMELA, (short for Process Meta Language). The semantics of the language are carefully chosen to make it impossible to define anything other than models for which the reachable system states can, in principle,

be enumerated exhaustively, thus guaranteeing the decidability of standard safety and liveness properties. The three main types of objects that can be manipulated are:

a. processes,
b. channels, and
c. variables.

Only a bounded number of each of these types of objects can be instantiated. The focus of the language, and of the model checking system behind it, is on the specification of the *interactions* in a distributed system (i.e., on coordination); the focus is deliberately not on the specification of *computations*. This means that the language is strong in its support for all the ways in which concurrent processes might interact, through message channels, rendezvous operations, or through the manipulation of shared variables. For the same reason the language is weak in the expression of the computational aspects of programming. There is, for instance, no support for floating point operations, no support for explicit memory or process management, and only rudimentary support for functions.

Each process definition in PROMELA is converted by SPIN into an optimized automaton description that is used inside the model checking procedure. To perform model checking, SPIN in effect computes the cross product of the behaviors of all executing processes, while at the same time attempting to prove or disprove a logic property specified as a `never` claim. If a property is specified in LTL, SPIN will first convert the LTL formula into an equivalent `never` claim, using the procedure outlined in [3]. Formally, the `never` claim specifies a Büchi automaton (a specific type of omega automaton) and the acceptance conditions from this automaton are evaluated on the global systems execution graph. The efficiency of the SPIN system comes from its on-the-fly procedure for performing this check that requires only small amounts of memory to be consumed for each state that is reached. The details of this so-called *nested depth-first search algorithm* are documented in [7].

4 A small example

A small example can help to illustrate the main concepts in automata-based model checking tools. For inspiration we will use the well publicized flaw in the control software for the Pathfinder mission to Mars in early 1997. We will avoid a detailed description of the problem as it occurred, and instead focus on its essence: an unforeseen conflict between two sets of coordination rules. For our purposes we can limit the problem to just two concurrent processes running at two different priority levels; the problem as it occurred on Mars involved three processes and three priority levels.

The first coordination rule served to enforce a simple mutual exclusion policy for access of processes to shared resources, such as a central data bus. The separate uses of these shared resources may not overlap in time, hence each process has to gain access to the resource by first setting a lock that will block further attempts to access the same resource until the lock is released. Handling access to shared resources in a distributed system is of course quite standard, a practice going back to the late 1960s, and fully trusted.

The second coordination rule served to enforce a simple scheme of process priorities. Some tasks are clearly of greater urgency than other tasks. Some types of processing and data transfer could, for instance, be done at any time, while certain types of data gathering must be completed within strict deadlines, at risk of losing some of the data. Scheduling process executions on a priority basis is equally well understood. Also here no grave problems are to be suspected. In concurrent systems, though, there is no guarantee that if one combines two sound and trusted schemes, the result will also be sound and trustworthy. And so it was that the Pathfinder control software repeatedly got stuck in deadlock states, only to recover by automatic reboots that would restore the system to a healthy state when a long period of inactivity was detected.

How did this happen? It is simple enough to build an abstract model of a system that uses both mutual exclusion rules and process priority rules, and then check with SPIN if there is any chance that the system might deadlock. (Of course, it is much easier to do this once an error has been identified – the real challenge in model checking is to locate errors that are not yet known to exist.) First, let us model the access of a single process to a shared resource. In PROMELA this looks as follows:

```
active proctype high()
{
end:do
    :: h_state = waiting;
       atomic { mutex == free -> mutex = busy };
       h_state = running;

       /* critical section - consume data */

       atomic { h_state = idle; mutex = free }
    od
}
```

The keyword `active` says that the process declaration that follows is to be instantiated in the initial system state, causing one process to execute the behavior that is declared here. The process name we have used here is `high`. There are no startup parameters, so the argument list that follows the process name is empty here. The body of the process is a single iteration, called a `do` loop in PROMELA. In this case there is only one option for execution within the loop. That option is preceded by the double colon flag `::`. The process starts each cycle of its execution by setting the value of global state variable `h_state` to the value `waiting` to signal that this process is now waiting to gain access to its critical section (accessing the shared resource). The process then checks if

the resource is free or locked, by checking the value of a special `mutex` variable. If the value of `mutex` is `free` the process atomically sets it to `busy` and proceeds. If the variable does not have the proper value, the process waits until it does. Once the lock has been passed, the process state is set to the new value `running` to indicate that we are now executing inside the critical section. The semantics of PROMELA dictates that there can be an arbitrary delay in between the execution of any two atomic statements of a process, so we do not explicitly have to represent the access to the critical section itself: any amount of computation might be performed in this step. At some point this access to the shared resource ends and the lock is to be released. Here we use another `atomic` sequence to secure that both the process state and the value of the `mutex` variable are reset at the same instant of time. Execution now resumes from the start of the loop, and the process will once again attempt to gain access to the critical section to perform more tasks.

We will implement the priority rule by preventing a second (low priority) process from executing whenever the high priority process above is *not* in its `idle` state (i.e., it is either `running` or `waiting` to run). The code looks as follows:

```
active proctype low() provided (h_state == idle)
{
end:do
    :: l_state = waiting;
        atomic { mutex == free -> mutex = busy};
        l_state = running;

        /* critical section - produce data */

        atomic { l_state = idle; mutex = free }
    od
}
```

The model of the low priority process is almost identical to that of the high priority process with a few small exceptions. First, the process records its state in a global variable `l_state` instead of `h_state`. And second, we have added a `provided` clause to the process name on the first line of this declaration. This clause limits execution of the process to the cases where the value of global variable `h_state` is equal to `idle`, as intended.

To complete the specification, and turn it into a valid PROMELA specification, we insert the following global declarations:

```
mtype = { free, busy, idle, waiting, running };

mtype h_state = idle;
mtype l_state = idle;
mtype mutex = free;
```

This defines the five symbolic names we have used, and it declares the three global variables, properly initialized to their respective values.

As a first test, we can simulate the execution of this system with SPIN. The output of such a run is, with `$` the command prompt:

```
$ spin -p pathfinder
  0:proc - (:root:) creates proc  0 (high)
  0:proc - (:root:) creates proc  1 (low)
  1:proc  1 (low) line  39 "pathfinder" (state 9)
    [l_state = waiting]
  2:proc  1 (low) line  41 "pathfinder" (state 4)
    [((mutex==free))]
  3:proc  1 (low) line  41 "pathfinder" (state 3)
    [mutex = busy]
  4:proc  0 (high) line 26 "pathfinder" (state 9)
    [h_state = waiting]
  5:proc  1 (low) line  42 "pathfinder" (state 5)
    <<Not Enabled>> timeout
  5:proc  1 (low) line  42 "pathfinder" (state 5)
    <<Not Enabled>>
#processes: 2
        h_state = waiting
        l_state = waiting
        mutex = busy
  4:proc  1 (low) line  42 "pathfinder" (state 5)
  4:proc  0 (high) line 28 "pathfinder" (state 4)
2 processes created
$
```

The parameter `-p` tells the SPIN simulator to print every single step in the process executions. We then see that in the first step the two process instantiations are created. At this point either the low priority process or the high priority process could start executing. In this scenario the low priority process starts first by updating its state to `waiting`, testing the availability of the lock, and setting it to `busy`. The high priority process then follows, finds the lock to be set, so it cannot proceed, and oddly we see that the low priority process can also not proceed (it is noted as `"Not Enabled"`), prevented from executing by the `provided` priority clause.

A `timeout` now occurs, to see if any process can get unwedged in this way, but the situation remains as it is and system execution comes to a halt. This simple simulation identifies the deadlock, but in general one cannot of course rely on this fortunate behavior. A simulation is much like a single test randomly chosen from among thousands or millions of possible tests. With some luck, it will shake out most of the bugs, but we have no guarantee that all bugs are caught.

This is how Glenn Reeves, the leader of the software team for the Pathfinder spacecraft, described the version of the problem in the real software, with three priority levels:

> The failure turned out to be a case of priority inversion (how we discovered this and how we fixed it are covered later). The higher priority bc_dist task was blocked by the much lower priority ASI/MET task that was holding a shared resource. The ASI/MET task had acquired this resource and then been preempted by several of the

medium priority tasks. When the bc_sched task was activated, to setup the transactions for the next 1553 bus cycle, it detected that the bc_dist task had not completed its execution. The resource that caused this problem was a mutual exclusion semaphore used within the select() mechanism to control access to the list of file descriptors that the select() mechanism was to wait on. [11]

To perform a formal verification of the system, which is able to give us a more firm guarantee on system correctness, we proceed as follows:

```
$ spin -a pathfinder # generate a model specific
                     # verifier
$ cc -o pan pan.c    # compile it
$ pan                # and run it
pan: invalid endstate (at depth 4)
pan: wrote pathfinder.trail
(Spin Version 3.3.5 -- 28 September 1999)
Warning: Search not completed
    + Partial Order Reduction

Full statespace search for:
    never-claim             - (none specified)
    assertion violations    +
    acceptance   cycles     - (not selected)
    invalid endstates    +

State-vector 20 byte, depth reached 4, errors: 1
       5 states, stored
       1 states, matched
       6 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493     memory usage (Mbyte)
$
```

After exploring only five reachable states, the model checker reports an error state: a deadlock, which is euphemistically noted as an `"invalid endstate"`. Any reachable system state where processes are not stopped at an explicit `"end"` label, or at the end of the code (which is not reachable here), will be reported as an invalid endstate. In this case, it's the same scenario we saw before. Replaying the error trail found by the model checker, using SPIN's guided simulation option, produces:

```
$ spin -t -p pathfinder # replays the scenario in
                        # pathfinder.trail
  1:proc  1 (low) line  40 "pathfinder" (state 1)
    [l_state = waiting]
  2:proc  1 (low) line  41 "pathfinder" (state 2)
    [((mutex==free))]
  2:proc  1 (low) line  41 "pathfinder" (state 3)
    [mutex = busy]
  3:proc  1 (low) line  42 "pathfinder" (state 5)
    [l_state = running]
  4:proc  0 (high) line  27 "pathfinder" (state 1)
    [h_state = waiting]
spin: trail ends after 4 steps
#processes: 2
        h_state = waiting
        l_state = running
        mutex = busy
  4:proc  1 (low) line  46 "pathfinder" (state 8)
  4:proc  0 (high) line  28 "pathfinder" (state 4)
2 processes created
$
```

which is a slightly terser description of the same four steps leading into the error state.

Remarkably, the entire system state space that has to be searched to prove any property for this system is no larger than 12 reachable system states, connected by 15 transitions.

```
$ pan -c0    # don't stop at errors,
             # complete the search
pan: invalid endstate (at depth 4)
(Spin Version 3.3.5 -- 28 September 1999)
    + Partial Order Reduction

Full statespace search for:
    never-claim          - (none specified)
    assertion violations +
    acceptance   cycles  - (not selected)
    invalid endstates +

State-vector 20 byte, depth reached 4, errors: 2
      12 states, stored
       3 states, matched
      15 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493     memory usage (Mbyte)

unreached in proctype high
    (0 of 12 states)
unreached in proctype low
    (0 of 12 states)
$
```

This qualifies the problem as one of the smallest model checking problems of practical significance known today, even four states smaller than the ruling candidate so far (the alternating bit protocol with 16 reachable system states).

Despite the fatal deadlock problem, we can show that the model succeeds in correctly enforcing the mutual exclusion and the process priority rules. To prove mutual exclusion, for instance, we can formulate the following property in LTL:

```
<> p
```

("eventually p becomes true") with `p` defined as: `"(h_state == running && l_state == running)"`, and show with a model checking run that this property cannot be satisfied (meaning that the corresponding system states are effectively unreachable, as intended).

For completeness, the steps we would take to convert the LTL property into a `never` claim, add it to the model with the proper definition of `p`, and then to generate, compile, and run the model checker can be as follows:

```
$ cp pathfinder path2
$ echo "#define p (h_state == running && \
                   l_state == running)" >> path2
$ spin -f "<>p" >> path2
$ spin -a path2
$ cc -o pan pan.c
```

The model checker is now ready to use.

```
$ pan -a    # search for acceptance cycles
(Spin Version 3.3.5 -- 28 September 1999)
    + Partial Order Reduction

Full statespace search for:
   never-claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid endstates    - (disabled by never-claim)

State-vector 24 byte, depth reached 9, errors: 0
   12 states, stored
    5 states, matched
   17 transitions (= stored+matched)
    0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493     memory usage (Mbyte)

unreached in proctype high
   (0 of 12 states)
unreached in proctype low
   (0 of 12 states)
```

No errors are reported, meaning that the LTL property that expresses a violation of the mutual exclusion rules can indeed not be satisfied.

Rather than memorizing all the steps that one would type to a command line interface, most SPIN users prefer to use SPIN's graphical user interface, XSPIN. The GUI, illustrated in Fig. 1, is written in Tcl/Tk and has options for graphically displaying the automata structures produced by SPIN from PROMELA language descriptions, and for generating graphical message sequence charts for the error traces. The range of options on search algorithms that is available for application in more challenging model checking experiments is also accessible via more user-friendly panels that explain the purpose and usage of each such option.

All information about SPIN and PROMELA is available online at url:
`http://cm.bell-labs.com/cm/cs/what/spin`.

5 The SPIN workshops

The SPIN workshops started in 1995, at the initiative of Jean-Charles Grégoire from the University of Montreal, in Canada. What we assumed at the time would be a single occurrence has continued, with the attendance for some of the workshops rivaling that of the large formal methods conferences with which they are often co-located.

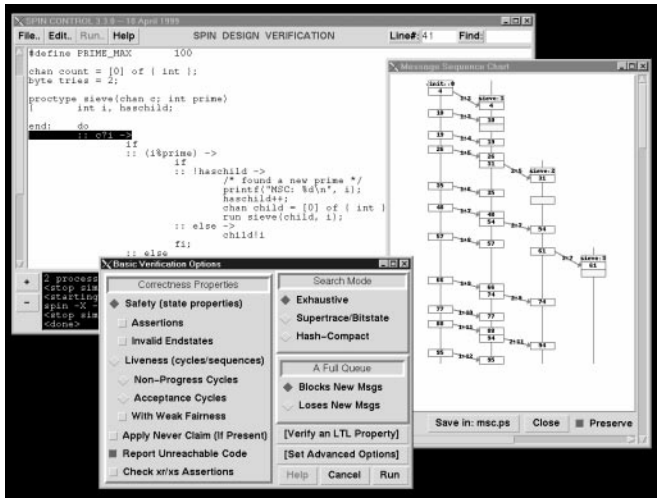

Fig. 1. SPIN's graphical user interface

For this special section of STTT we have included a representative set of papers from SPIN98, the fourth SPIN Workshop, held in November 1998 in Paris, France. The papers we selected focus on both the application and the theory of model checking with SPIN: we start with three theory oriented papers, and conclude with three application oriented papers.

Model checking is often only feasible for problems of a realistic size if we first apply judicious abstractions that help to make the problem solvable with finite means. If a model checker fails to solve our problem within the available means, this simply means that the abstractions we employed were insufficient. It does not necessarily mean that the problem itself is unsolvable. Abstraction is both the key to the successful application of model checking, and its primary weakness. After all, the abstractions are today most often applied manually and rarely the subject of verification themselves. The first paper from the workshop, by Yonit Kesten and the workshop's keynote speaker Amir Pnueli, addresses this subject and proposes a set of rules for the application of *sound* abstractions of both systems and properties of systems.

The second paper, by Lynette Millett and Tim Teitelbaum, takes another approach at the abstraction problem. With basic data dependency analysis and program slicing techniques it is sometimes possible to derive a simplified model of a system to solve a verification problem. The paper gives an overview of this approach.

SPIN, as a software verification engine, supports the use of LTL (linear temporal logic) for the specification of correctness properties. Most hardware model checkers, however, support branching time logics, such as CTL

or CTL*. LTL model checking has an efficient implementation in the SPIN system, compatible with its basic on-the-fly verification algorithm and with the proof approximation methods that it offers. In CTL* one can make statements about the existence of paths through the state graph, using universal and existential quantification. This means that reachability information per se, as exploited in SPIN's nested depth-first search algorithm, is no longer sufficient to immediately determine the validity of a temporal formula. Within the context of an explicit state model checker it would mean that one has to store substantially more information for each reachable system state, see [13], drastically degrading the performance of the model checker. Willem Visser and Howard Barringer discuss a new algorithm for CTL* model checking that they argue is potentially compatible with the on-the-fly verification methods used in SPIN.

The next few papers deal with applications. The challenge to link model checkers to widely used CASE tools, or to apply model checking techniques to standard, implementation level, languages is pursued by many. As an example of this work, the paper by Klaus Havelund and Thomas Pressburger, from NASA Ames Research Center, tackles the problem of SPIN's application to the verification of JAVA programs. In converting a JAVA program into a SPIN model, many issues have to be resolved, specifically to guarantee that the model remains bounded. The paper summarizes the progress the authors have made.

In the next paper Rene de Vries and Jan Tretmans, from Twente University in The Netherlands, consider how the search engine in SPIN can be used in a novel way, for the generation conformance test cases, based on a theory of testing that they have developed. The special section is concluded with a paper by Moataz Kamel and Stefan Leue, from the University of Waterloo in Canada, presenting the results of an application of SPIN model checking to one of the components of the OMG's common object request broker architecture (CORBA).

We hope you find this special section of interest and look forward to greeting you at future SPIN workshops, of which we hope there will be many. The organizers of SPIN98.

References

1. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. Logic of Programs: Workshop, Yorktown Heights, NY. LNCS 131. Berlin, Heidelberg, New York: Springer-Verlag, 1981
2. Corbett, J.C.: Evaluating Deadlock Detection Methods for Concurrent Software. IEEE Trans. on Software Eng. 22(3), 1996
3. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple On-the-fly Automatic Verification of Linear Temporal Logic. Proc. IFIP Symp. on Protocol Specification, Testing and Verification, Warsaw, Poland. Chapman & Hall, Germany, pp. 173–184, 1995
4. Hajek, J.: Automatically verified data transfer protocols. Proc. 4th ICCC, Kyoto, pp. 749–756
5. Holzmann, G.J.: A Theory for protocol validation. IEEE Trans. on Computers C-31:8, pp. 730–738
6. Holzmann, G.J.: Design and Validation of Computer Protocols, Englewood Cliffs, NJ: Prentice Hall, 1991
7. Holzmann, G.J., Peled, D., Yannakakis, M.: On nested depth-first search. In: he Spin Verification System, DIMACS/32, American Mathematical Society, pp. 23–32. 1996
8. McMillan, K.L.: Symbolic model checking: an approach to the state explosion problem, PhD Thesis, Kluwer Academic, 1993
9. Pnueli, A.: The temporal logic of programs. Proc. 18th IEEE Symp. on Foundations of Computer Science, Providence, RI, pp. 46–57 1997
10. Queille, J.P.: Le systéme César: description, spécification et analyse des applications réparties, PhD Thesis, CS Dept., Univ. Grenoble, France, 1982
11. Reeves, G.E.: What really happened on Mars, `http://www.research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html`
12. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. Proc. 1st Symposium on Logic in Computer Science, Cambridge, pp. 322–331, 1986
13. Villiers, P. de: Validation of a Microkernel: a Case study, PhD Thesis, University of Stellenbosch, South Africa, Dec. 1999
14. Tanenbaum, A.S.: Computer Networks, Englewood Cliffs, NJ: Prentice Hall, 1st edition, 1981
15. Tanenbaum, A.S.: Email exchange, 23 January 1995
16. Watson, R.W.: Timesharing System Design Concepts, New York: McGraw-Hill, 1970