The Model Checker SPIN

Gerard J. Holzmann

Abstract—SPIN is an efficient verification system for models of distributed software systems. It has been used to detect design errors in applications ranging from high-level descriptions of distributed algorithms to detailed code for controlling telephone exchanges. This paper gives an overview of the design and structure of the verifier, reviews its theoretical foundation, and gives an overview of significant practical applications.

Index Terms—Formal methods, program verification, design verification, model checking, distributed systems, concurrency.

1 INTRODUCTION

SPIN is a generic verification system that supports the design and verification of asynchronous process systems [36], [38]. SPIN verification models are focused on proving the correctness of *process interactions*, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these. In focusing on asynchronous control in software systems, rather than synchronous control in hardware systems, SPIN distinguishes itself from other well-known approaches to model checking, e.g., [12], [49], [53].

As a formal methods tool, SPIN aims to provide:

1) an intuitive, program-like notation for specifying design choices unambiguously, without implementation detail,
2) a powerful, concise notation for expressing general correctness requirements, and
3) a methodology for establishing the logical consistency of the design choices from 1) and the matching correctness requirements from 2).

Many formalisms have been suggested to address the first two items, but rarely are the language choices directly related to a basic feasibility requirement for the third item. In SPIN the notations are chosen in such a way that the logical consistency of a design can be demonstrated mechanically by the tool. SPIN accepts design specifications written in the verification language PROMELA (a Process Meta Language) [36], and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL) [60].

There are no general decision procedures for unbounded systems, and one could well question the soundness of a design that would assume unbounded growth. Models that can be specified in PROMELA are, therefore, always required to be bounded, and have only countably many distinct behaviors. This means that all correctness properties automatically become formally decidable, within the constraints that are set by problem size and the computational resources that are available to the model checker to render the proofs. All verification systems, of course, do have physical limitations that are set by problem size, machine memory size, and the maximum runtime that the user is willing, or able, to endure. These constraints are an often neglected issue in formal verification. We study the limitations of the model checker explicitly and offer relief strategies for problems that are outside the normal domain of exhaustive proof. Such strategies are discussed in Sections 3.3 and 3.4 of this paper

1.1 Structure

The basic structure of the SPIN model checker is illustrated in Fig. 1. The typical mode of working is to start with the specification of a high level model of a concurrent system, or distributed algorithm, typically using SPIN's graphical front-end XSPIN. After fixing syntax errors, interactive simulation is performed until basic confidence is gained that the design behaves as intended. Then, in a third step, SPIN is used to generate an optimized on-the-fly verification program from the high level specification. This verifier is compiled, with possible compile-time choices for the types of reduction algorithms to be used, and executed. If any counterexamples to the correctness claims are detected, these can be fed back into the interactive simulator and inspected in detail to establish, and remove, their cause.

The remainder of this paper consists of three main parts. Section 2 gives an overview of the basic verification method that SPIN employs. Section 3 summarizes the basic algorithms and complexity management techniques that have been implemented. Section 4 gives three examples of typical applications of the SPIN model checker to design and verification problems. The first example is the problem of devising a correct process scheduler for a distributed operating system; the second problem is the verification of a leader election protocol for a distributed ring; the third problem is the proof of correctness of a standard sliding window flow control protocol. Section 4 concludes with a summary of a range of other significant verification problems to which SPIN has been applied. Section 5 concludes the paper.

• *G.J. Holzmann is with the Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ 07974. Email: gerard@research.bell-labs.com.*

Manuscript received Sept. 30, 1996.
Recommended for acceptance by L.K. Dillon and S. Sankar.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number 104928.0.

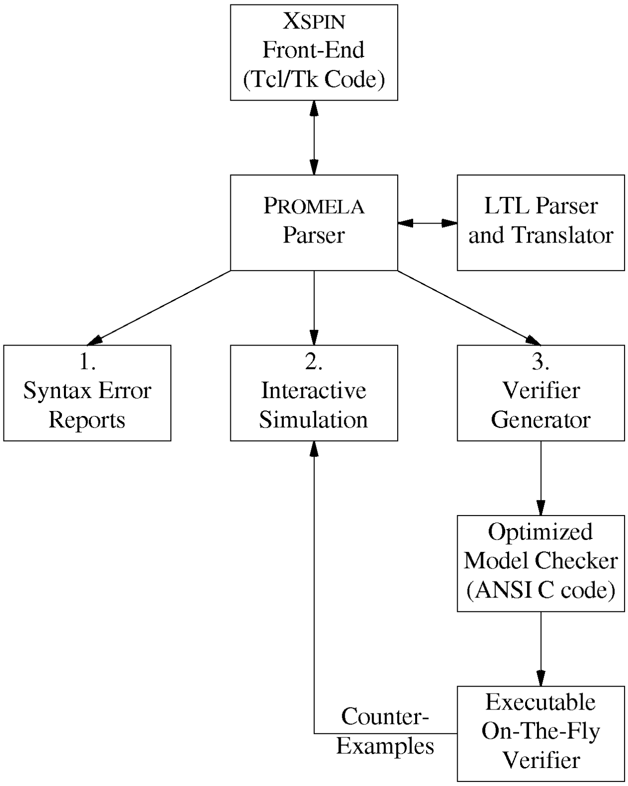


Fig. 1. The structure of SPIN simulation and verification.

2 FOUNDATION

SPIN has its roots in the earliest protocol verification systems based on on-the-fly reachability analysis from the early '80s [32], [33] [34]. The purpose of these first verifiers was to provide an effective tool that could be used to solve problems of practical significance. The fundamental computational complexity of the problem to be solved forces one to choose carefully which features will be supported, or risk compromising the practical significance of the tool itself. The predecessors of SPIN therefore supported the verification of only standard safety properties and a limited range of liveness properties.

Work on logic model checking techniques, pioneered by Clarke and Emerson [12], and by Sifakis and Queille [61], laid the foundation for a new generation of model checking tools, with a larger scope of verification capabilities. Vardi and Wolper extended this work with an automata theoretic model [77] that has become the formal basis for temporal logic model checking in the SPIN system. The framework is summarized below.

The description of a concurrent system in PROMELA consists of one or more user-defined process templates, or `proctype` definitions, and at least one process instantiation. The templates define the behavior of different types of processes. Any running process can instantiate further asynchronous processes, using the process templates.

SPIN translates each process template into a finite automaton. The global behavior of the concurrent system is obtained by computing an asynchronous interleaving product of automata, one automaton per asynchronous process behavior. The resulting global system behavior is itself again represented by an automaton. This interleaving product is often referred to as the *state space* of the system, and, because it can easily be represented as a graph, it is also commonly referred to as the global *reachability graph*.

To perform verification, SPIN takes a correctness claim that is specified as a temporal logic formula, converts that formula into a Büchi automaton, and computes the *synchronous* product of this claim and the automaton representing the global state space. The result is again a Büchi automaton.[1] If the language accepted by this automaton is empty, this means that the original claim is not satisfied for the given system. If the language is nonempty, it contains precisely those behaviors that satisfy the original temporal logic formula. In SPIN, we use the correctness claims (and temporal logic formulae) to formalize systems erroneous system behaviors, i.e., behaviors that are *undesirable*. The verification process then either proves that such behaviors are impossible or it provides detailed examples of behaviors that match.

In the worst case, the global reachability graph has the size of the Cartesian product of all component systems. The specification language PROMELA is defined in such a way that each component always has a strictly finite range. This applies to processes (which can have only finitely many control states), but also to all local and global variables (which can have only finitely many distinct values), and all message channels (each with bounded and user-defined capacity).[2] Although, in practice, the size of the global reachability never approaches the worst case size, the reachable portion of the Cartesian product can also easily become prohibitively expensive to construct exhaustively. A number of complexity management techniques have been developed to combat this problem. We will discuss the ones that are included in SPIN in separate sections below.

2.1 Temporal Logic Requirements

SPIN accepts correctness properties expressed in linear temporal logic (LTL). Vardi and Wolper showed in 1983 that any LTL formula can be translated into a Büchi automaton [78]. SPIN performs the conversion to Büchi automata mechanically based on a simple on-the-fly construction [25]. The automata that are generated formally *accept* only those (infinite) system executions that satisfy the corresponding LTL formula.

As noted briefly above, we use correctness requirements to formalize system behaviors that are claimed to be *impossible*, i.e., to formalize the potential violation of correct system behavior. Each positive LTL formula can, of course, be turned into a negative one, and vice versa, by prefixing it with a logical negation operator. At first sight, it may seem that it would not make much difference which form is chosen, but there is a difference, as first explained in [77].

A positive claim requires us to prove that the language of the system (i.e., all its executions) is included in the language of the claim. A negative claim, on the other hand, requires us

1. A Büchi automaton is an automaton defined over infinite input sequences, rather than finite ones as in standard finite state machine theory [73].
2. Variables and message channels also have a *state* that is selected from a finite domain. These *passive* components can change state only as a synchronous side-effect of transitions that are made in the *active* components in the verification model (i.e., instantiated processes).

to prove that the intersection of the language of the system and of the claim is *empty*. The size of the statespace for a language inclusion proof is at most the size of the Cartesian product of the (automata representing) system and claim, and at least the size of their sum. The worst-case statespace size to prove emptiness of a language intersection is still the size of the Cartesian product of system and claim, but, in the best case, it is *zero*. Note that if no initial portion of the invalid behavior represented by the claim appears in the system, the intersection contains no states. SPIN, therefore, works with negative correctness claims and solves the verification problem by language intersection.

A Büchi automaton *accepts* a system execution if and only if that execution forces it to pass through one or more of its accepting states infinitely often. We call such behaviors *acceptance* cycles. (Note that for an infinite execution to exist in a finite system, the behavior must be cyclic.) To prove that no execution sequence of the system matches the negated correctness claim, it suffices to prove the absence of acceptance cycles in the combined execution of the system and the Büchi automaton representing the claim. This combined execution is formally defined by a *synchronous* product of the system and the claim.

The entire computation, starting from the individual concurrent components and a single Büchi automaton representing the correctness claim, is done by SPIN in one single procedure, using a nested depth-first search algorithm [17], [36], [43]. The algorithm terminates when an acceptance cycle is found (which then constitutes a counterexample to a correctness requirement), or, when no counterexample exists, when the complete intersection product has been computed.

2.2 Domain of Application

The design of SPIN is focused on the efficient verification of asynchronous software systems. This focus affects many central tool characteristics, including the design of the specification language, the logic, the verification procedure, the reduction techniques, and the state encoding methods. We take a closer look at some of these issues in the following sections.

3 ALGORITHMS

SPIN's verification procedure is based on an optimized depth-first graph traversal method. We summarize the effect of the algorithms that are used to optimize this search in the next few sections.

3.1 Nested Depth-First Search

The cycle detection method used in SPIN is of central importance. The method is required to be compatible with all modes of verification, including exhaustive search, bit-state hashing, and partial order reduction techniques.

The classical algorithm for finding a cycle in a graph is Tarjan's depth-first search algorithm [72], which constructs the strongly connected components in linear time by adding two integer numbers to every state reached: the *dfs*-number and the *lowlink*-number. Because the state spaces that SPIN can generate may contain billions of reachable states, these two integer numbers require at least 32 bits of storage each.

If a strongly connected component in the reachability graph contains at least one accepting state, a reachable acceptance cycle has been shown to exist. Tarjan's algorithm relies on the accuracy of the *dfs* and the *lowlink* numbers and is not compatible with the bit-state hashing techniques that are also part of SPIN (summarized below). Efficient alternatives to Tarjan's algorithm exist [36], [17], [43]. With these methods, one performs a nested depth-first search, possibly visiting every state twice, but storing every state only once. The nested depth-first search can be implemented with just 2 bits of overhead per state, instead of the 64 bits of Tarjan's algorithm, using a simple encoding method [26].

The principle of the nested depth-first search algorithm is as follows. For an accepting cycle to exist in the reachability graph, at least one accepting state must be both reachable from the initial system state (the root of the graph) and it must be reachable from itself. The first depth-first search establishes which accepting states are reachable from the initial system state. The second (nested) search starts at each accepting state thus detected, and it checks whether or not that state is reachable from itself. If it is, a complete execution sequence that includes the acceptance cycle has also been constructed: It is the concatenation of all the steps that are on both the first and the second depth-first search stack. In the context of the SPIN model checker, this execution sequence always equates to a counterexample of a user-defined correctness claim, and it can be printed as proof that the correctness claim is invalid for the system as specified.

The nested depth-first search algorithm does not preserve the capability to detect *all* possible acceptance cycles that may appear in the reachability graph. It can, however, be proven to detect *at least one* such cycle if one or more cycles exists [17]. Because acceptance cycles in SPIN constitute counterexamples to correctness claims, establishing either their absence or their presence always suffices for the purposes of verification.

The nested depth first search algorithm in SPIN is extended with an optional weak fairness constraint, using Choueka's flag construction method [14], [17]. Under the weak fairness constraint, every process that contains at least one transition that remains enabled infinitely long, is guaranteed to execute that transition within finite time.

3.2 From LTL Formulae to Büchi Automata

LTL formulae can be used to express both safety and liveness properties. An LTL formula **f** may contain any lowercase propositional symbol **p**, combined with unary or binary, Boolean and/or temporal operators, using the grammar shown in Fig. 2.

For instance, the LTL property.

```
[] (p U q)
```

states that it is always guaranteed that `p` remains true at least until `q` becomes true. Similarly

```
[] (<> p)
```

states that at any point in an execution it is guaranteed that eventually `p` will become true at least once more.

```
f ::=      p
         | true
         | false
         | ( f )
         | f binop f
         | unop f

unop ::=   []       (always)
         | <>       (eventually)
         | !        (logical negation)

binop ::=  U        (strong until)
         | &&       (logical and)
         | ||       (logical or)
         | ->       (implication)
         | <->      (equivalence)
```

Fig. 2. LTL grammar.

The automata generated by SPIN for the above two formula are shown in Fig. 3, written in the syntax of PROMELA.

Both automata contain one nonaccepting state (the initial state of the Büchi automaton, `T0`) and one accepting state (named `accept` here), as illustrated in Fig. 4.

Both states contain a nondeterministic selection (`if..fi`) construct. For states `T0`, the choice is made between two possible transitions. For the `accept` states, there is only one option, so, strictly seen, the selection constructs that enclose these states are redundant. The selection of each transition in the automata is conditional on a propositional formula, i.e., the conditions are placed on the transitions of the automaton, not on the states themselves.

The translation algorithm [25] computes the states for a Büchi automaton by computing the set of subformulas that must hold in each reachable state and in each of its successor states. The formula is first converted into normal form, with negations only adjacent to atomic propositions. An initial state is created, marked with the formula that is to be matched and a dummy incoming edge. The remainder of the automaton is then computed recursively. At each stage, a subformula that remains to be satisfied is taken and, according to its leading operator, the current state may be split into two states, with each copy inheriting a different part 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 subformulae with until operators.

In the first released version of SPIN, it was the user's responsibility to convert LTL formulae into automata. The manual process, however, can be challenging and is error-prone. The built-in algorithm for this conversion that was added to SPIN removed these obstacles, and has significantly increased the accessibility of SPIN's LTL checking capabilities.

3.3 Partial Order Reduction

SPIN uses a partial order reduction method [57] to reduce the number of reachable states that must be explored to complete a verification. The reduction is based on the observation that the validity of an LTL formula is often insensitive to the order in which concurrent and independently executed events are interleaved in the depth-first search. Instead of generating an exhaustive state space that includes all execution sequences as paths, the verifier can generate a reduced state space, with only representatives of classes of execution sequences that are indistinguishable for a given correctness property. The implementation of this reduction method is based on a static reduction technique, described in [41], that, before the actual verification begins, identifies cases where partial order reduction rules can safely be applied when the verification itself is performed. This static reduction method avoids the runtime overhead that has plagued partial order reduction strategies in the past.

Fig. 5 shows a measurement of the number of reachable states that has to be generated to complete the verification for a model of a leader election algorithm [18], discussed in more detail in Section 4.2 and in Appendix B. It illustrates a best-case performance of the reduction algorithm, where exponential growth in the number of processes participating in the protocol is reduced to linear growth. In more typical cases, the reduction in the state space size and in the memory requirements is linear in the size of the model, yielding savings in memory and runtime from 10 to 90 percent [41].

```
$ spin -f "[](p U q)"
never {
T0:
       if
       :: (p) -> goto T0
       :: (q) -> goto accept
       fi;
accept:
       if
       :: ((p) || (q)) -> goto T0
       fi
}
```

```
$ spin -f "[]<>p"
never {
T0:
       if
       :: (true) -> goto T0
       :: (p) -> goto accept
       fi;
accept:
       if
       :: (true) -> goto T0
       fi
}
```

Fig. 3. PROMELA syntax for two LTL formulae.

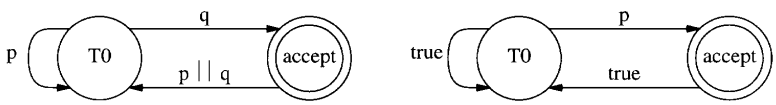


Fig. 4. Büchi automata for the LTL formulae `[] (pUq)` (left) and `[]<> p` (right).

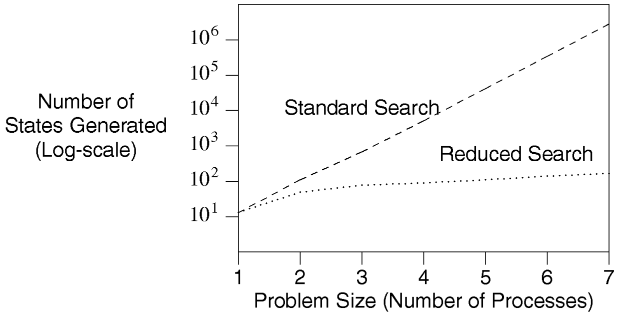


Fig. 5. Effect of partial order reduction.

An important characteristic of SPIN's static reduction method is that it cannot lead to a noticeable increase of the memory requirements, compared to exhaustive searches. The reduction method is not sensitive to decisions about process or variable orderings. The alternative reduction methods based on binary decision diagrams have been shown to lack these two important features, e.g., [8].

The correctness properties of the reduction algorithm itself (i.e., the preservation of safety and liveness properties) were verified independently with the help of the theorem prover HOL [13].

3.4 Memory Management

The size of the interleaving product that SPIN computes can, in the worst case, grow exponentially with the number of processes. Given the size of the product, expressed as the number of reachable system states R, we can place upper-bounds on the amount of memory (space) and time that would be required to complete various types of verification tasks.

To prove safety properties, such as absence of deadlock or user defined assertions, carries a computational cost that is linear in the number of reachable states R, both in (CPU) time and in (memory) space. To prove simple liveness properties, such as absence of starvation or of acceptance cycles, requires twice as much time, but no noticeable increase in the memory requirements, as explained above [26]. To prove LTL properties, the time requirements increase by a factor that can, in the worst case, itself again depend exponentially on the number of temporal operators used in the formula [77]. The space requirements, however, remain largely unaffected [26]. Meaningful LTL properties rarely have more than two or three operators, so the increase in complexity is relatively small, compared to the complexity that is contributed by the system itself.

Memory is a bounded resource on any system. It is not difficult to construct a model checker that uses only a small amount of memory, but one can only do so at the expense of unacceptable increases in runtime [35]. A main emphasis of the research in this area has therefore been on devising techniques that can economize the memory requirements of a reachability analysis, *without* incurring unrealistic increases in runtime requirements. Two such techniques are discussed in the following two sections.

3.4.1 State Compression

To make state comparisons possible, a reachable state must be compressed in the same way, whether it is generated at the beginning or at the end of a search. Dynamic Huffman coding or Lempel-Zvi & Welch compression techniques are therefore not directly usable in this type of application. Static Huffman encoding, and run-length coding do have the required properties. Their effectiveness in model checkers was studied in [39]. It was found that run-length encoding added a substantial runtime overhead (~400 percent) in return for only a modest reduction of the memory requirements (10 to 20 percent). Static Huffman encoding was found to add a smaller, but still substantial run time overhead (~300 percent) in return for a somewhat larger reduction of the memory requirements (60 to 70 percent). Greater compression typically implies greater run time penalties, cf. [79], [29].

A different state compression technique was added to the standard distribution of the SPIN software in late 1995. It delivers comparable reductions of the memory requirements, but for a relatively small run time penalty (10 to 20 percent). The method works on the premise that every process and every channel in a PROMELA specification has only relatively small number of unique local states. The large number of global states can often be attributed to the large number of possible combinations of local process and channel states. By storing the local states separately from the global states, and, using unique indices into the local state tables inside the global state table, one can then reduce the memory requirements without incurring much additional run time overhead.

Fig. 6 illustrates this method. Instead of storing the complete concatenation of all local state descriptors for variables, channels, and processes, the compression algorithm now stores each separable element alone, and uses unique indices to the local descriptors in the global state vector, see also [45].

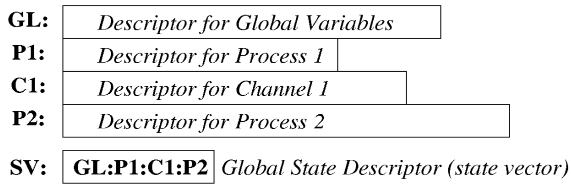


Fig. 6. State compression algorithm—indexing method.

If there are 256 or fewer local process states, the index for that process can be just 8 bits, or a single byte of memory, independent of the true number of bytes that is needed to store the complete local state. The 256 distinct local states are stored only once, but they may be referred to many times, each referral costing only 1 byte of memory within the global state descriptor.

The compression method implemented in SPIN allows the user to set the width of an index to 1, 2, 3, or 4 bytes. If the index width is chosen too tightly, a local table will overflow, a warning is issued, and the search must be repeated with a different width. The amount of reduction achieved depends on the relative size of indices and local states, which can be arbitrarily large. In practice, the most commonly observed reduction is 60 to 80 percent.

Table 1 illustrates the performance of this compression technique for a typical application. The application is a standard go-back-n sliding window that we will examine more closely in Section 4. The full text of the protocol model in the input language of the verifier, PROMELA, is given in Appendix C.

TABLE 1
EFFECT OF COMPRESSION

Type of Run	No. States	Memory (Mb)	Time (sec.)
Standard	2,435,220	156.59	107.56
Compressed	2,435,220	59.57	123.46

The standard run, without compression, explores over 2,000,000 reachable states, using 156 Mbytes of memory. When the compression algorithm discussed here is enabled, the memory requirements are reduced by 62 percent, for a run time penalty of about 15 percent. (The run times are for a 150 MHz SGI system.)

3.4.2 Bit-State Hashing

For problem sizes that preclude exhaustive verification, a high-coverage approximation of the results of an exhaustive run can be performed in relatively small amounts of memory. For this purpose, SPIN includes an implementation of the bit-state hashing or *supertrace* technique [34], [42]. With this algorithm, two bits of memory are used to store a reachable state. The bit-addresses are computed with two statistically independent hash functions.

If storing one reachable system state requires S bytes of memory, and if our machine has M bytes of memory available, the model checker exhausts its available memory after generating M/S states. If the true number of reachable states Rexceeds M/S, then the *problem coverage* of that verification run is $M/(R \times S)$. If, for example, M is 10^8 bytes, S is 10^3 bytes, and R is 10^6 states, then the maximal problem coverage would be 0.1 (meaning that just 10 percent of the reachable states are inspected).

The bit-state hashing technique can, under the same system constraints, produce an average problem coverage close to 1 (meaning close to 100 percent coverage). In general, when $M < R \times S$, the bit-state hashing technique typically realizes a far superior problem coverage than standard exhaustive searches [42].

The effect of the bit-state hashing technique on problem coverage is illustrated in Fig. 7. In this case, the algorithm was applied to a data transfer protocol that requires the generation of approximately 427,000 states (R), each taking 1.3 Kbits of memory (S) to store for an exhaustive verification. The total memory requirements for a standard search are $R \times S$, or close to 73 Mbytes of memory (about 2^{29} bits). Fig. 7 shows that with only 1 percent of the memory required for an exhaustive search, the bit-state hashing technique can realize a problem coverage close to 100 percent. For still lower amounts of memory, the coverage for a single verification run also drops to zero. It can be increased again with a sequential bit-state hashing technique [42]. In sequential bit-state hashing, multiple runs with statistically independent hashing functions can be performed until the required coverage level is reached.

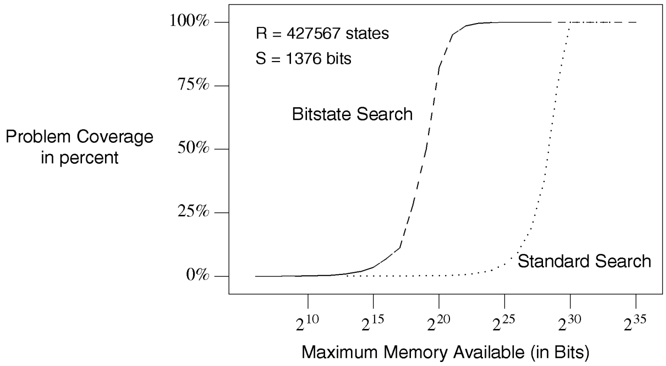


Fig. 7. Measured problem coverage [34], [42] effect of the optional bit-state hashing technique in SPIN.

The bit-state hashing techniques have been applied with good results in several large-scale industrial applications of formal verification, e.g., [11], [40].

4 PRACTICAL APPLICATIONS

As typical examples of the application of SPIN to the verification of concurrent systems, we discuss three different types of problems. The first is a protocol for scheduling processes in a distributed operating system, as discussed in [64]. The second example is the algorithm for leader election in a unidirectional ring, as given in [18], [63]. The third example is a standard flow control protocol, as given in [71]. For a tutorial introduction to the way in which design models such as these can be constructed, we refer to [36], [38].

We conclude this section with a summary of other significant applications of SPIN, concentrating on those applications that have appeared in the literature.

4.1 Process Scheduling

The first problem we consider is the problem of scheduling process executions in a distributed operating system. The problem can be remarkably hard to solve both correctly and in a manner that is also reasonably efficient. One attempt is described in [64]. The solution originally given there can be represented in the input language of the verifier with minimal effort. The complete verification model is included in Appendix A. Fig. 8 shows the labeled transition systems generated by SPIN from this specification.

In this version of the code, there are two asynchronously executing processes, a client and a server, that repeatedly make calls on the sleep and wakeup routines from the operating system. The client process consumes resources that are provided, one by one, by the server. The availability of a resource is modeled here by a zero value of the global variable `r_lock`. The client can set the value of `r_lock` to one (line 22), but only the server can reset it to zero (line 30).

If the resource is unavailable (line 12 in Appendix A), a flag is set to indicate the process's needs (line 13), its state is changed (line 14), and it is put to sleep by the operating system until it is reawakened by the server (line 16). The entire sequence is performed after first setting (line 10) and then releasing (lines 15 and 23) a spinlock that guarantees exclusive access to this section of the code.

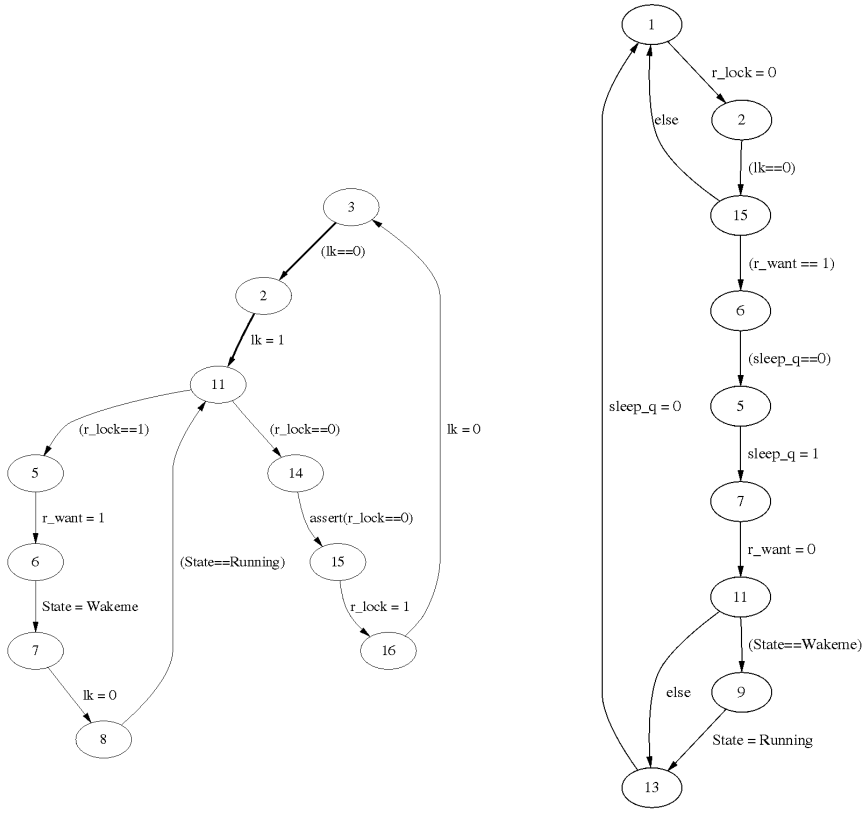


Fig. 8. Labeled transition systems for client (left) and server process (right).

Client Process	Server Process	Line No.	Local States	Comment
[r_lock = 1]		22	16,15	Consume Resource
[lk = 0]		23	3,15	Release the lock
[lk==0]		10	2,15	Test the lock
[lk = 1]		10	11,15	Set the lock
[r_lock==1]		12	5,15	Resource not available
[r_want = 1]		13	6,15	Set want flag
	[r_want==1]	33	6,6	Server checks flag
	[sleep_q==0]	35	6,5	Wait lock on sleep_q
	[sleep_q = 1]	35	6,7	Set it
	[r_want = 0]	37	6,11	Resets the flag
	[else]	44	6,13	No process sleeping!
	[sleep_q = 0]	46	6,1	Release the lock
	[r_lock = 0]	30	6,2	Provide new resource
[State = Wakeme]		14	7,2	Client goes to sleep!
[lk = 0]		15	8,2	Releases the lock
	[lk==0]	31	8,15	Server checks the lock
	[else]	47	8,1	No flag is set
Non-Progress Cycle:				
	[r_lock = 0]	30	8,2	The server repeats this
	[lk==0]	31	8,15	forever, while the client
	[else]	47	8,1	remains suspended.

Fig. 9. An error scenario generated by SPIN (annotated).

The server checks for a suspended client process each time that it makes a new resource available. If the client process is asleep, it is reawakened. This sequence can be performed without requiring the spinlock to be set explicitly by the server process as well. It suffices to have the server wait only until the client process has released the lock. The client process can safely reset it, even when the server is in the middle of its wakeup routine.

After a period of experimentation with the sleep-wakeup routines described, it was discovered that a race condition could allow a client process to be suspended without ever being reawakened by the server. A fix was proposed (shown on line 39). Two questions were phrased by the designer of this code: 1) could a mechanical verification system have found the original problem? and 2) is the modified version indeed free from race conditions? Running the verification of the model as specified in Appendix A settles both issues in a few seconds: the answer to the first question is *yes*, the answer to the second question is *no.*

With or without the proposed change from line 39, SPIN quickly generates the execution scenario from Fig. 9, proving that a process can still remain suspended indefinitely. The first two columns in Fig. 9 show the statements executed by the client or the server process, in sequence from top to bottom. The third column gives the line number of that statement in the listing from Appendix A. The fourth column gives the local states of the client and the server, as shown in Fig. 8. In this scenario, the client process executes some statements and pauses in its local state 6. The server then takes over and pauses in its local state 2. When the client process then executes two more steps, the trap has been set and closed.

The scenario is reported by SPIN in a verification run for nonprogress cycles subject to the weak fairness constraint. The true cause of the error is that the client process can execute the code between lines 12 and 15 without having first obtained the critical lock from line 10. This happens each time a suspended client process is reawakened by the server. In this case, the process will repeat the check for the availability of the resource without first reclaiming the lock. The correct fix, that provably removes all erroneous behaviors, is to add a jump `goto sleep` directly after the client process resumes executing (i.e., between lines 16 and 17 in Appendix A).

The design problem discussed here is subtle enough that it can stump the most experienced designers for weeks or more. Yet in this case the verification task is an almost trivial exercise that can be done within one hour from start to finish. There are no more than 300 reachable states in the product state space for this verification model. None of the verification runs performed take more than 0.1 CPU/sec on an average workstation.

4.2 Leader Election

The second problem we consider is a standard algorithm for leader election in a unidirectional ring. An efficient algorithm to solve this problem was published by Dolev, Klawe and Rodeh in 1982 [18], and can be found in many standard textbooks. The description we have used is from pp. 37-40, [63].

In this version of the algorithm, all processes will participate in the election; that is, they cannot decide to join in at a later point in the execution. The verification model given in Appendix B is a direct translation of the algorithm from pp. 38–39, [63]. There are several interesting properties we may want to prove about this algorithm, but the most important one is that under no circumstance should it be possible for more than one process to declare to be the leader of the ring.

It is simple enough to add a global variable to the model that can be used to count the number of leaders in the system. This is done on lines 8 and 56 in Appendix B. The correctness requirement itself can be specified with the temporal property.

```
[] (nr_leaders <= 1)
```

but it can also (and more efficiently) be verified by adding an in-line assertion to the model.

```
assert (nr_leaders == 1)
```

at the point where a process claims to have won the election (i.e., at line 57 in Appendix B) and increments the variable `nr_leaders`.

For the model as given in Appendix B, a verification run with SPIN is completed in under 0.1 CPU/sec, and the global state space (using partial order reduction, cf. Fig. 5) contains no more than 108 reachable system states.

That the number of leaders can never exceed one does not guarantee, though, that eventually a leader is elected. To prove this property, we need a temporal logic formula, such as:

```
<>[] (nr_leaders == 1)
```

The negated version of this property is translated by SPIN into a simple two-state Büchi automaton. The verification that indeed this negated property is not part of the model takes under 0.1 CPU/sec. The number of reachable states increases to 202.

4.1.1 A Modification

As a separate issue we can ask how the algorithm should be modified for processes to defer their decision to participate in the election. According to p. 38 [63], this could be done provided that *"a process would decide to participate (only when) first receiving a message concerning the election."* An attempt to modify the model for this extension is to remove line 18, and to add the following additional option to the model, between the current lines 19 and 20:

```
:: atomic {
     (!election_started && !Active) ->
          /* 1 process can start election */
          /* by sending the first msg */
          /* and becoming Active */
     election_started = 1;
     out!one(mynumber);
     first_message = 0;
     Active = 1}
  }
```

We also add the following code between lines 20 and 21, and lines 37 and 38, to allow a nonactive process to join in at the first received message:

```
if
:: (first_message && !Active) ->
     if
     :: Active = 1; /* join election */
        out!one(mynumber)
     :: skip        /* or stand aside */
     fi
:: else
fi;
first_message = 0;
```

We have used two additional variables, a global variable `election_started` (initial value zero), that allows one arbitrary process to start the election, and a local variable `first_message` (initial value one) that limits the moment when a process can join the election to the first incoming message. The initial value of local variable `Active` is set to zero.

It can quickly be shown by a model checker (and quite hard to show manually) that this extension does *not* preserve the critical correctness requirement. A verification run demonstrates first that the assertion on line 29 is no longer valid: The winner of the election does not necessarily hold the largest number of the ring, because not all processes necessarily participate in the election. After removing the assertion, SPIN succeeds in proving that there can never be more than one leader (i.e., the assertion on line 57 remains valid). The proof of the second property, that inevitably one of the processes must be elected leader, also proceeds without problems, taking just under 4 sec of CPU time for five participating processes.

In the algorithm as given here, the initialization of the processes in the ring is predefined. A version with a randomized initialization requires 21 additional lines of PROMELA. Both versions of the algorithm can be found in the PROMELA distribution among the test-cases for the model checker.

4.3 Flow Control

Tanenbaum, in his book on computer networks [71], gives, among many others, a succinct description of a standard go-back-n flow control protocol. The protocol can be translated with little effort into the 50 line PROMELA specification shown in Appendix C. Its critical correctness properties can be proven in a few minutes of CPU time.

The first step in working with SPIN is to produce the verification model. This model has very much the role of an engineering prototype in the design of a new distributed algorithm, concurrent system, or, as in this case, protocol. During the construction, SPIN can be used to provide quick checks of the syntactical correctness of the growing model. When the model has been completely specified, random or interactive simulations can be performed to check (test) its approximate working. Debugging statements, such as `printf`s and inline assertions, can be added at this stage to help the user develop a better understanding of system behavior.

The description from [71] was converted into PROMELA in about half an hour. The result of a first random simulation run with this model, as displayed in a message sequence chart by XSPIN, is shown on the left-hand side of Fig. 10. The model contains two print statements (also included in Appendix C) that are not part of Tanenbaum's description, but are used to help in debugging the model. The first print statement records when a protocol entity accepts a correctly received message, the second records when the sending protocol times out, waiting for acknowledgments.

Inspecting the first message sequence chart, we notice that a `timeout` statement is executed, which correctly recovers the protocol from a hang-state. Closer inspection, however, reveals that the first retransmitted message after the `timeout` is not recognized as a duplicate message by the receiver, but erroneously accepted. The cause of this error can be traced back to an incorrect modeling of the increment operation `inc()` (i.e., with our macro `Wrong`). The correct interpretation of this operation can be found 18 pages earlier in Tanenbaum (shown as the macro `Right`, in Appendix C). The difference between `Right` and `Wrong` in this case expressed precisely the essential property for the correct working of the go-back-n sliding window protocol: the range of the sequence numbers must be at least one greater than the maximum window size. Correcting the problem changes the simulation trace to the one shown on the right-hand side in Fig. 10

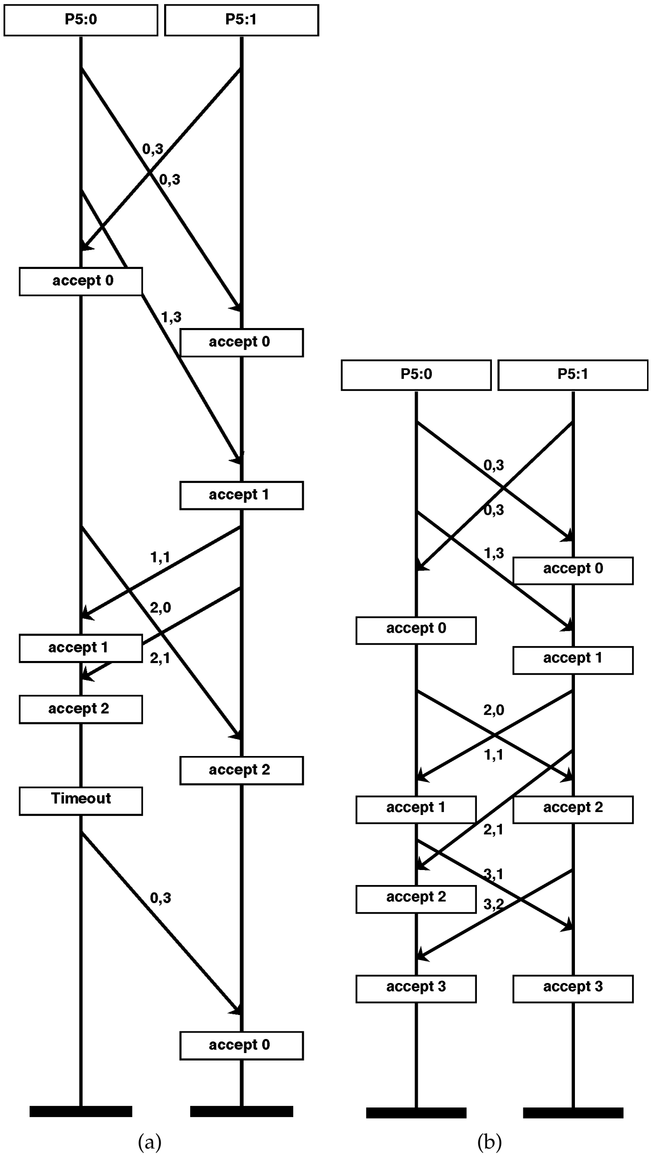


Fig. 10. Two simulation runs.

The simulation runs are of inestimable value in model building. They can, however, only give us an impression of the true properties of the protocol. To prove with mathematical certainty that the protocol correctly transfers messages from sender to receiver, we must use model checking techniques.

4.3.1 Safety Properties

To get a first measure of the complexity of this protocol, we can perform an exhaustive verification run with SPIN to prove some basic safety properties, such as absence of deadlock, unreachable code, unspecified receptions, etc. To do so, we can prompt SPIN to generate the C-code for an optimized, special purpose, model checking system for this specific protocol model. The code is compiled, with either the default choices for reduction techniques and memory management, or with explicit user overrides. If we compile the basic verifier in this way for pure safety properties, with all reduction options enabled (i.e., with state compression, partial order reduction, and with a transition coarsening strategy that we will discuss shortly), we can measure the size of the state space that the SPIN verifier builds to complete the proof. The results, varying the range of the sequence numbers from 3 to 5, are shown in Table 2.

TABLE 2
COMPLEXITY INCREASE WITH RANGE OF SEQUENCE NUMBERS

MaxSeq	No. States	Memory (Mb)	Time (sec.)
3	14,645	2.0	1.2
4	127,599	10.9	12.5
5	916,180	66.9	97.1

As can be expected, every increase in the sequence number range increases the number of reachable states, the memory requirements and the time requirements for the proof. Depending on the size of the available machine, the maximum range of sequence numbers for which the correct working of the protocol can be proven mechanically in this fashion will be between 5 and 7. With special techniques that exploit specific knowledge about this application, greater reductions in the memory requirements may be obtained, so that the scope of this proof can be substantially extended [28], but we will limit ourselves here to only the built-in capabilities of SPIN.

To see what the relative effect is on the complexity of the verification of each of the complexity control measures we have taken, Table 3 shows the effect of each of the three techniques when applied in isolation.

- The first row in Table 3 shows the number of reachable states, and the memory and time requirements (user plus system time), for the same verification run as in Table 2, for the model exactly as given in Appendix C, without the use of partial order reduction or compression.
- The second row shows the effect of enabling the state compression method: the memory requirements are reduced to roughly 50 percent of the first run, while time time requirements remain largely unaffected in this case.
- The third row shows the effect of adding only the partial order reduction method: the number of reachable states and the memory requirements drop by more than 50 percent, and the run time reduces by approximately 75 percent.
- The fourth row shows the effect of transition coarsening. To achieve coarsening, we can define some of the transition sequences in the protocol as indivisible, by marking them as either `atomic` or deterministic d_step sequences in PROMELA.

  As one example, we can note that the timeout in the protocol is intended to prevent a potential deadlock in the protocol, when messages are lost. The retransmission sequence that is then initiated merely retransmits all unacknowledged messages, in their original order. There is no interaction between sender and receiver in this phase, and nothing is gained by interleaving the actions of sender and receiver processes within it, e.g., by considering the cases where the receiver starts processing the first retransmitted messages while the retransmission of other messages is still in progress. In cases such as this one, marking these local sequences as indivisible can help to reduce the complexity of the verification task considerably. At the same time, it places the additional responsibility on the user of this technique to show by other means that this model reduction technique is justified and cannot mask errors
- The last row combines the effect of all techniques, and matches the first entry in Table 2. Compared to the first row, all techniques combined succeed in reducing the memory and time requirements of the search by, respectively, a factor of 10 and 20. The difference is not too important for the safety properties we have considered so far, but it will be of value when we move on to prove more complex correctness properties for this protocol.

4.3.2 Liveness Properties

To prove liveness properties, such as faithful message transfer, we must extend the verification model further. We can, for instance, add two environment processes to the model, one to submit data to be transferred by the protocol, and the other to check that the data accepted by the receiver matches the data that was submitted by the sender. The question then is, how many different data items do we need to use to obtain a reliable proof? The minimal number turns out to be independent of both the sequence number range and the window size: Just three distinct data items suffice for any flow control protocol, as first shown by Wolper [82]. Traditionally, three different colors are used, e.g., `red`, `white`, and `blue`. The input data sequence consists of one `red` and one `blue` message inserted randomly in an infinite sequence of `white` messages. We check at the receiver that neither the `red` nor the `blue` message can disappear from the data sequence, nor arrive out of order. If this is impossible, the correctly ordered delivery for any two messages in the sequence is guaranteed.

Fig. 11 shows the PROMELA description of an environment process that supports this proof technique. The `Source` process generates the infinite sequence of `white` data messages, with the `red` and `blue` data items randomly inserted. It is a simple three state machine. In the first two states, a nondeterministic choice is made at each step between the sending of a white message without changing state, or the sending of a colored message, followed by a transition to the next state. In the last state (labeled `end`), only more `white` messages can be generated.

TABLE 3
EFFECT OF COMPLEXITY CONTROL MEASURES (MAXSEQ = 3)

No. States	Memory (Mb)	Time (sec.)	Compression	Partial Order	Coarsening
402,997	26.7	25.8			
402,997	13.8	23.7	+		
182,735	12.5	6.5		+	
16,441	2.5	1.4			+
14,645	2.0	1.2	+	+	+

```
mtype { red, white, blue };/* enumeration type */
mtype rcvd = white, sent = white;/* global monitor variables */
chan source = [0] of { mtype };  /* synchronous message channel */

active proctype Source()
{
	do
	:: source!white
	:: source!red -> break
	od;
	do
	:: source!white
	:: source!blue -> break
	od;
end:	do
	:: source!white
	od
}
```

Fig. 11. Environment process for proving correctness of data transfer in the go-back-n protocol.

To complete this extension, we must also make some small modifications of the protocol processes itself. One of the processes must now retrieve a message to be sent from the `source` channel (which we have implemented as a zero-length rendezvous channel here). The other process can simply send only white data messages. Immediately after sending a colored message, the sending process is made to update the global monitor variable `sent`, and mark it with the color of the last data item transmitted. Similarly, the matching receiver process is modified to update the global monitor variable `rcvd` immediately after accepting an incoming data message. Since the protocol is symmetrical, only message transmissions in one direction need to be marked and monitored in this way.

We can now express the correctness requirements as LTL formulae by using the following six definitions of propositional symbols:

```
#define sw    (sent == white)
#define sr    (sent == red)
#define sb    (sent == blue)

#define rw    (rcvd == white)
#define rr    (rcvd == red)
#define rb    (rcvd == blue)
```

We would like to verify two properties:

- Message can be lost. Violations of this requirement (remember, SPIN works with negated correctness requirements) can be expressed in the formula:

  `![](sr -> <> rr)`

  The last part of this formula states that the sending of a `red` message, `sr`, logically implies (–>) its eventual (<>) reception, `rr`. This condition is claimed to be always (`[]`) true. The negated version, finally, formalizes all possible behaviors for which the condition is violated.
- No messages can arrive our of order. We can express violations of this requirement in the formula:

  `(!rr U rb)`

  The `U` in this formula is the strong until formula (see Section 3.2). The formula then states that it is an error if the red message is not received until after (`U`) the `blue` message was received.
- The two properties can also trivially be combined into a single formula, formalizing the undesirable execution sequences that are claimed to be impossible:

  `(![](sr -> <> rr)) || (!rr U rb)`

In general, it is more efficient to separate orthogonal requirements, and to prove each property separately with smaller verification runs [58]. Each of the first two formulae above separately translates into a two-state Büchi automaton. The combination of the two formulae, however, translates into a six-state automaton, which is more expensive to verify.

As a rough estimate of the additional expense of the verification of these correctness requirements, we note that we have added an environment process of three states, two monitor variables with three possible values each, and a Büchi automaton of two states. In addition, we have extended the possible contents of a window full of messages, and hence of the receiver process's input queue, by a factor of roughly $MaxSeq^3$, or 27 when MaxSeq = 3. Multiplying these factors give a potential increase in the size of the reachable state space of 3 × 3 × 3 × 2 × 27 or 1,458.

Table 4 shows the size of the state space that SPIN constructs to prove that the correctness requirements are indeed satisfied for the go-back-n protocol.

The runs shown in Table 4 were all performed with all reduction options listed in Table 3 enabled. The first two runs did not include an explicit correctness requirement and are for comparison. For each property, we performed one run for ideal channels, and another for lossy channels. The protocol maintains its correctness for both choices. For the lossy channels, we have to extend the property to be verified somewhat to rule out pseudo-counterexamples (i.e., those that involve infinite repetition of message loss, or infinite deferrals of the retransmission of messages). Those changes are included in the measurements shown above.

TABLE 4
VERIFICATION OF THE GO-BACK-n PROTOCOL (MAXSEQ = 3)

No. States	Memory (Mb)	Time (sec.)	Property	Channels
14,645	2.0	1.2	*none*	*ideal*
59,310	3.3	3.9	*none*	*lossy*
69,891	4.9	8.5	(!rr U rb)	*ideal*
931,274	39.3	123.7	(!rr U rb)	*lossy*
439,831	20.2	76.2	`(![](sr -> <> rr))`	*ideal*
4,790,030	199.1	1243.2	`(![](sr -> <> rr))`	*lossy*

The second requirement is clearly more expensive to verify than the first. The maximum increase of the state space size that is incurred when the properties are added, however, is only about 20 percent of the worst case increase we estimated above. Including the possibility of message loss can be seen to increase the expense of the verification by up to an order of magnitude.

4.4 Other Applications

SPIN has been distributed[3] freely, in source form, since early 1991. Today, the system has been installed on several thousand machines worldwide, with a number of active users that can no longer be accurately counted. Work on the system can be classified in four broad categories:

1) theoretical studies,
2) empirical studies of the relative effectiveness of different types of search and storage algorithms,
3) extensions and revisions of the SPIN code, and
4) significant practical applications.

Examples in each category may be found in the proceedings of the SPIN Workshops that have been held since 1995 [68]. Examples of modifications of the SPIN software, for instance, include extensions for real-time verification [74], reactive systems modeling [54], bisimulation equivalence proofs [21], different types of partial order reduction [27], [76], process algebras [24], alternate state machine models [69], alternate compression techniques [79], [29], [28], [45], and implementation generation [5], [51].

Applications of SPIN to real-life problems also span a broad range of problems. The obvious applications are to prove correctness of generic distributed algorithms, such as the leader election algorithm illustrated in Fig. 1, nonstandard mutual exclusion algorithms [37], communications network design problems [65], or protocol design problems [2], [3], [22], [23], [7], [16], [36], [51]. In the course of the work on SPIN, we have also constructed verification models for, e.g., the Cambridge ring protocol [56], and the IEEE logical link control protocol LLC 802.2 [52]. Others constructed fragments of larger protocol applications such as XTP [70] and TCP/IP. These and other unpublished models are available from the author.

SPIN has also been applied to the verification of data transfer protocols [5], bus protocols [6], address registration protocols [55], error control protocols [66], requirements analysis [4], controllers for reactive systems [10], distributed process scheduling algorithms [59], fault tolerant systems [1], hardware-software codesign [80], asynchronous hardware designs [62], multiprocessor designs [76], local area network controllers [30], microkernel design [19], [75], operating systems code [9], [64], railway signaling protocols and circuitry [36], [20], [15], rendezvous algorithms [44], security protocols [47], flood surge control systems [48], feature interaction problems [50], ethernet collision avoidance techniques [46], and self-stabilizing protocols [67].

5 CONCLUSION

Most mature engineering disciplines include a methodology for constructing and analyzing prototypes of designs. Concurrent systems is, compared to civil engineering or physics, a relatively young discipline and it is not surprising that comparable tools are still somewhat scarce. Still, the first attempts to develop the basic methodology for of on-the-fly automated verification date back more than a decade, e.g., [31], [81], [32], [33]. The most recent versions of these algorithms, as captured in tools such as SPIN, begin to provide some of the required capability. The design methodology that is supported by SPIN can be summarized as follows:

- A distinction is made between behavior and requirements on behavior. The designer specifies the two aspects of the design in an unambiguous way by defining a verification or prototype in the language PROMELA.
- The prototype is verified using the model checker SPIN. The requirements and behaviors are checked for both their internal and their mutual consistency.
- The design is revised until its critical correctness properties can successfully be proven. Only then does it make sense to refine the design decisions further toward a full systems implementation.

The increasing number of applications of SPIN, and the growing acceptance of formal methods in general, are hopeful signs that this paradigm of design is maturing, and is gaining recognition where it counts: among the practitioners of distributed systems design.

3. The Spin model checker software can be retrieved by anonymous ftp from directory `/netlib/s;in` on host `netlib.bell-labs.com`.

APPENDIX A – VERIFICATION MODEL OF A PROCESS SCHEDULING ALGORITHM

The model below for implementing sleep and wakeup routines in a distributed operating systems kernel is based on [64].

```
mtype = {Wakeme, Running};            /* two symbolic names */

bit    lk,  sleep_q;                   /* boolean variables */
bit    r_lock,   r_want;
mtype  State =   Running;            /* variable of type mtype */

active proctype client()
{
sleep:                                 /* the sleep routine */
  atomic { (lk == 0) -> lk = 1};      /* SPINlock(&lk) */
  do                                  /* while r.lock is set */
  :: (r_lock == 1) ->                 /* r.lock == 1 */
      r_want = 1;              /* set the want flag */
      State = Wakeme;          /* remember State */
      lk = 0;                  /* freelock(&lk) */
      (State == Running);      /* wait for wakeup */
  :: else ->                      /* r.lock == 0 */
      break                    /* break from do-loop */
  od;
progress:                            /* progress label */
  assert(r_lock == 0);           /* should still be true */
  r_lock = 1;                    /* consumed resource */
  lk = 0;                        /* freelock(&lk) */
  goto sleep
}

active proctype server()           /* interrupt routine */
{
wakeup:                            /* wakeup routine */
 r_lock = 0;                     /* r.lock = 0 */
 (lk == 0);                      /* waitlock(&lk) */
 if                              /* selection structure */
 :: r_want ->                    /* someone is sleeping */
      atomic {                 /* get spinlock on sleep queue */
            (sleep_q == 0) -> sleep_q = 1
      };                       /* end of indivisible fragment */
      r_want = 0;              /* reset the want flag */
#ifdef PROPOSED_FIX
       (lk == 0);              /* waitlock(&lk) */
#endif
       if                      /* selection structure */
       :: (State == Wakeme) -> /* the client process is asleep
             State = Running; /* wake-up the client process */
       :: else -> skip         /* else do nothing */
       fi;                     /* end of selection structure */
       sleep_q = 0             /* release spinlock on sleep queue */
  :: else -> skip                /* else do nothing */
  fi;                            /* end of selection structure */
  goto wakeup                    /* jump to the wakeup label */
}
```

APPENDIX B – VERIFICATION MODEL OF A LEADER ELECTION ALGORITHM

The model below follows the description of the leader election protocol in a unidirectional ring from [82] as it is discussed and formalized in [63].

```
#define N    5         /* nr of processes */
#define I    3         /* node given the smallest number */
#define L   10         /* size of buffer (>= 2*N) */

mtype = {one, two, winner}; /* three symbolic msg names */
chan q[N] = [L] of {mtype, byte}; /* asynchronous channel */

byte nr_leaders = 0; /* count the number of process that
                        think they are leader of the ring */
proctype node (chan in, out; byte mynumber)    /* process template */
{ bit Active = 1, know_winner = 0;
  byte nr, maximum = mynumber, neighbourR;

```

```
  xr in;    /* claim exclusive recv access to channel in */
  xs out;   /* claim exclusive send access to channel out */

  printf(“MSC: percentd\n,” mynumber);
  out!one(mynumber);    /* send msg of type one, with par mynumber */
end:    do
  :: in?one(nr) ->      /* receive msg of type one, with par nr */
       if
       :: Active ->
            if
            :: nr != maximum ->
                 out!two(nr);
                 neighbourR = nr
            :: else ->
                 /* max is greatest number */
                 assert(nr == N);
                 know_winner = 1;
                 out!winner,nr;
            fi
        :: else ->
             out!one(nr)
        fi

  :: in?two(nr) ->
       if
       :: Active ->
            if
            :: neighbourR > nr && neighbourR > maximum ->
                  maximum = neighbourR;
                  out!one(neighbourR)
            :: else ->
                 Active = 0
            fi
       :: else ->
            out!two(nr)
       fi
  :: in?winner,nr ->
       if
       :: nr != mynumber ->
            printf(“MSC: LOST\n”);
       :: else ->
            printf(“MSC: LEADER\n”);
            nr_leaders++;
            assert(nrleaders == 1)
       fi;
       if
       :: know_winner
       :: else -> out!winner,nr
       fi;
       break
  od
}

init {         /* the initial process */
  byte proc;
  atomic {   /* atomically activate N copies of proc template node */
       proc = 1;
       do
       :: proc <= N ->
            run node (q[proc-1], q[procpercentN], (N+I-proc)percentN+1);
            proc++
       :: proc > N ->
            break
       od
  }
}
```

APPENDIX C – VERIFICATION MODEL OF A SLIDING WINDOW PROTOCOL

This model of a go-back-n sliding window protocol follows the description from p. 214, pp. 232-233, [71]. The model below includes some annotations to facilitate random and guided simulations with SPIN.

```
#define MaxSeq      5    /* window size */
#define Wrong(x)    x = (x+1) percent (MaxSeq)
#define Right(x)    x = (x+1) percent (MaxSeq + 1)
#define inc(x)           Right(x)

chan q[2] = [MaxSeq] of { byte, byte };   /* message passing channel */

active [2] proctype p5()     /* starts two copies of proctype p5 */
{ byte NextFrame, AckExp, FrameExp, r, s, nbuf, i;
  chan in, out;
  in = q[_pid];
  out = q[1-_pid];
  xr in; xs out;        /* partial order reduction claims */

  do
  :: nbuf < MaxSeq ->   /* outgoing messages */
       nbuf++;
       out!NextFrame,   (FrameExp + MaxSeq) percent (MaxSeq + 1);
       inc(NextFrame)

  :: q[_pid]?r,s ->       /* incoming messages */
       if
       :: r == FrameExp ->
            printf(“MSC: accept percentd\n,” r);
            inc(FrameExp)
       :: else /* ignore message */
       fi;
       do
       :: ((AckExp <= s) && (s < NextFrame))
       || ((AckExp <= s) && (NextFrame < AckExp))
       || ((s < NextFrame) && (NextFrame < AckExp)) ->
             nbuf--;
             inc(AckExp)
       :: else -> break
       od

  :: timeout -> &&    /* retransmission timeout */
        NextFrame = AckExp;
        printf(“MSC: timeout\n”);
        i = 1;
        do
        :: i <= nbuf ->
              out!NextFrame, (FrameExp + MaxSeq) percent (MaxSeq + 1);
              inc(NextFrame);
              i++
        :: else -> break
        od
  od
}
```


ACKNOWLEDGMENTS

The design and implementation of the SPIN model checker is based on contributions, ideas, and inspiration from many friends and colleagues over a long period of time, most notably Costas Courcoubetis, Peter Van Eijk, Michael Ferguson, Patrice Godefroid, Doug McIlroy, Doron Peled, Rob Pike, Jim Reeds, Moshe Vardi, Pierre Wolper, and Mihalis Yannakakis.

REFERENCES

[1] A. Agarwal, “A Unified Approach to Fault-Tolerance in Communication Protocols, Based on Recovery Procedures,” PhD thesis, Computer Science Dept., Concordia Univ., Montreal, Canada, 1995.

[2] M. Alipour, “On the Application of an Automated Validation Tool to Realistic Protocols,” MSc thesis, INRS-Telecommunications, Univ. du Quebec, Canada, Aug. 1994.

[3] P.R. D’Argenio, J.P. Katoen, T. Ruys, and J. Tretmans, “Modeling and Verifying a Bounded Retransmission Protocol,” *Proc. COST 247 Int’l Workshop Applied Formal Methods in System Design*, Maribor, Slovenia, June 1996.

[4] A. Basu, M. Hayden, G. Morrisett, and T. von Eicken, “A Language-Based Approach to Protocol Construction,” *Proc. ACM SIGPLAN Workshop Domain Specific Languages (WDSL)*, Paris, Jan. 1997.

[5] R. Bharadwaj and C. Hemeyer, “Verifying SCR Requirements Specifications Using State Exploration,” *Proc. First ACM/SIGPLAN Workshop Automatic Analysis of Software*, R. Cleaveland and D. Jackson, eds., pp. 9-24, Paris, Jan. 1997.

[6] B. Boigelot and P. Godefroid, “Model Checking in Practice: An Analysis of the ACCESS Bus Protocol Using SPIN,” *Proc. Formal Methods Europe (FME96)*, Oxford, England, *Lecture Notes in Computer Science 1,051*, pp. 465-478. Springer-Verlag, Mar. 1996.

[7] L. Bouvin, "Design of Validation Models in PROMELA for the Medium, Access Protocol of the PTM Project," Report Royal Inst. of Technology, Stockholm, Sweden, Aug. 1991.

[8] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Informatics Computing*, vol. 98, no. 2, pp. 142-170, June 1992.

[9] T. Cattel, "Modeling and Verification of a Multiprocessor Real-Time OS Kernel," *Proc. Seventh Int'l Conf. Formal Description Techniques*, pp. 35-50, Berne, Switzerland, Oct. 1994.

[10] T. Cattel, "Using Concurrency and Formal Methods for the Design of Safe Process Control," *Proc. PDSE/ICSE018 Workshop*, Berlin, Mar. 1996.

[11] J. Chaves, "Formal Methods at AT&T, An Industrial Usage Report," *Proc. Fourth FORTE Conf. Formal Description Techniques,* pp. 83-90, Sydney, Australia, 1991.

[12] E.M. Clarke, E.A. and Emerson, "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic," *Proc. Logic of Programs: Workshop,* Yorktown Heights, N.Y., *Lecture Notes in Computer Science 131*. Springer-Verlag, May 1981.

[13] C-T. Chou, and D. Peled, "Verifying a Model-Checking Algorithm," *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS96),* Passau, Germany, *Lecture Notes in Computer Science 1,055*, pp. 241-257. Springer-Verlag, Mar. 1996.

[14] Y. Choueka, "Theories of Automata on Mega-Tapes: A Simplified Approach," *J. Computer and System Science*, vol. 8, pp. 117-141, 1974.

[15] A. Cimatti, A. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso, "Model Checking Safety Critical Software with SPIN: An Application to a Railway Interlocking System," *Proc. Third SPIN Workshop*, R. Langerak, ed., Twente Univ., The Netherlands, Apr. 1997.

[16] J.C. Corbett, "Evaluating Deadlock Detection Methods for Concurrent Software," *IEEE Trans. Software Eng*., vol. 22, no. 3, pp. 161-180, Mar. 1996.

[17] C. Courcoubetis, M.Y. Vardi, P. Wolper, M. Yannakakis, "Memory Efficient Algorithms for the Verification of Temporal Properties," *Formal Methods in Systems Design*, vol. I, pp. 275-288, 1992.

[18] D. Dolev, M. Klawe, and M. Rodeh, "An O(n log n) Unidirectional Distributed Algorithm for Extrema Finding in a Circle," *J. Algorithms*, vol. 3, pp. 245-260, 1982.

[19] G. Duval and J. Julliand, "Modeling and Verification of the RUBIS Micro-Kernel with SPIN," *Proc. First SPIN Workshop*, J.-Ch. Gregoire, ed., INRS Quebec, Canada, Oct. 1995.

[20] P. Van Eijk, "Verifying Relay Circus Using State Machines," *Proc. Third SPIN Workshop*, R. Langerak, ed., Twente Univ., The Netherlands, Apr. 1997.

[21] H. Erdogmus, "Verifying Semantic Relations in SPIN," *Proc. First SPIN Workshop*, J.-Ch. Gregoire, ed., INRS Quebec, Canada, Oct. 1995.

[22] M.J. Ferguson, "Validation of the Radio Link Protocol," Data Services Task Group of ANSI Accredited TIA TR45-3, Contribution TR45.3.2.5/93.08.25.02, Sept. 1993.

[23] M.J. Ferguson, "Formalization and Validation of the Radio Link Protocol RLP1," *Computer Networks and ISDN Systems*, to appear, 1997.

[24] F. Gagnon, "Boulier, un validateur pour la language de spécification Gaston," PhD thesis, Univ. de Quebec, Canada, July 1995.

[25] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper, "Simple On-The-Fly Automatic Verification of Linear Temporal Logic," *Proc. IFIP/WG6.1 Symp. Protocol Specification, Testing, and Verification (PSTV95),* pp. 3-18, Warsaw, Poland, Chapman & Hall, June 1995.

[26] P. Godefroid, and G.J. Holzmann, "On the Verification of Temporal Properties," *Proc. IFIP/WG6.1 Symp. Protocol Specification, Testing, and Verification (PSTV93),* pp. 109-124, Liege, Belgium, North-Holland, June 1993.

[27] P. Godefroid, "Partial Order Methods for the Verification of Concurrent Systems," *Lecture Notes in Computer Science 1,032*. Springer-Verlag, 1996.

[28] P. Godefroid, "Symbolic Protocol Verification with Queue BDDs," *Proc. Logic in Computer Science*, pp. 198-206, Rutgers Univ., New Brunswick, N.J., July 1996.

[29] J.-C. Gregoire, "State Space Compression in SPIN with GETSs," *Proc. Second SPIN Workshop*, Rutgers Univ., New Brunswick, N.J., DIMACS/32, Am. Math. Soc., Aug. 1996.

[30] M. Griffioen, "Specification and Verification of a Wireless LAN Controller Chip Using PROMELA and SPIN," Technical Report, AT&T Network Wireless Systems, The Netherlands, 1996.

[31] J. Hajek, "Automatically Verified Data Transfer Protocols," *Proc. Fourth ICCC*, pp. 749-756, Kyoto, Aug. 1978.

[32] G.J. Holzmann, "PAN: A Protocol Specification Analyzer," Technical Report TM81-11271-5, AT&T Bell Laboratories, Mar. 1981.

[33] G.J. Holzmann, "Tracing Protocols," *AT&T Technical J.,* vol. 64, pp. 2,413-2,434, Dec. 1985.

[34] G.J. Holzmann, "An Improved Protocol Reachabily Analysis Technique," *Software, Practice and Experience*, vol. 18, no. 2, pp. 137-161, Feb. 1988.

[35] G.J. Holzmann, "Algorithms for Automated Protocol Verification," *AT&T Technical J.*, vol. 69, no. 1, pp. 32-44, Jan. 1990.

[36] G.J. Holzmann, *Design and Validation of Computer Protocols*. Englewood Cliffs, N.J.: Prentice Hall, 1991.

[37] G.J. Holzmann, "Protocol Design: Redefining The State of the Art," *IEEE Software*, pp. 17-22, Jan. 1992.

[38] G.J. Holzmann, P. Godefroid, and D. Pirottin, "Coverage Preserving Reduction Strategies for Reachabily Analysis," *Proc. IFIP/WG6.1 Symp. Protocol Specification, Testing, and Verification (PSTV92),* pp. 349-364, Orlando, Fla., North-Holland, June 1992.

[39] G.J. Holzmann, "Design and Validation of Protocols: A Tutorial," *Computer Networks and ISDN Systems*, vol. 25, no. 9, pp. 981-1,017, 1993.

[40] G.J. Holzmann, "The Theory and Practice of a Formal Method: NewCoRe," *Proc. 13th IFIP World Computer Congress*, pp. 35-44, Hamburg, Germany, North-Holland, Aug. 1994.

[41] G.J. Holzmann and D. Peled, "An Improvement in Formal Verification," *Proc. Seventh FORTE Conf. Formal Description Techniques,* pp. 177-194, Bern, Switzerland, Oct. 1994.

[42] G.J. Holzmann, "An Analysis of Bit-State Hashing," *Proc. IFIP/WG6.1 Symp. Protocol Specification, Testing, and Verification (PSTV95),* pp. 301-314, Warsaw, Poland, Chapman & Hall, June 1995.

[43] G.J. Holzmann, D. Peled, and M. Yannakakis, "On Nested Depth-First Search," *Proc. Second SPIN Workshop*, Rutgers Univ., New Brunswick, N.J., DIMACS/32, Am. Math. Soc., Aug. 1996.

[44] G.J. Holzmann, "Designing Bug-Free Protocols with SPIN," *The Computer Comm. J.*, to appear 1997.

[45] G.J. Holzmann, "State Compression in SPIN: Recursive Indexing and Compression Training Runs," *Proc. Third SPIN Workshop*, Twente Univ., R. Langerak, ed., The Netherlands, Apr. 1997.

[46] H.E. Jensen, K. Larsen, and A. Skou, "Modeling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL," *Proc. Second SPIN Workshop*, Rutgers Univ., New Brunswick, N.J., DIMACS/32, Am. Math. Soc., Aug. 1996.

[47] A. Joesang, "Security Protocol Verification Using SPIN," *Proc. First SPIN Workshop*, J.-Ch. Gregoire, ed., INRS Quebec, Canada, Oct. 1995.

[48] P. Kars, "The Application of PROMELA and SPIN in the BOS Project," *Proc. Second SPIN Workshop*, Rutgers Univ., New Brunswick, N.J., DIMACS/32, Am. Math. Soc., Aug. 1996.

[49] R.P. Kurshan, *Computer-Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[50] F.J. Lin, "Two Applications of PROMELA/SPIN," *Proc. First SPIN Workshop*, J.-Ch. Gregoire, ed., INRS Quebec, Canada, Oct. 1995.

[51] S. Loeffler and A. Serhrouchni, "Protocol Design: From Specification to Implementation," *Proc. Fifth Open Workshop for High Speed Networks*, Paris, Mar. 1996.

[52] "IEEE Std. 802-2-1985, ISO DIS 8802/2," IEEE Standards for Local Area Networks: Logical Link Control, Published by the IEEE Standards Board, 345 E. 47th Street, New York, NY 10017, USA, 111 pp., ISBN 471-82748-7, 1984. Revised as 802-2-1989 in Aug. 1989.

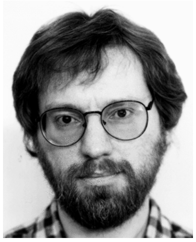
[53] K.L. McMillan, *Symbolic Model Checking*. Boston: Kluwer Academic, 1993.

[54] E. Najm and F. Olsen, "Reactive EFSMs, Reactive PROMELA/RSPIN," *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS96),* pp. 349-368, Passau, Germany, *Lecture Notes In Computer Science 1,055*, Springer-Verlag, Mar. 1996.

[55] T. Nakatani, "Verification of a Group Address Registration Protocol using PROMELA and SPIN," *Proc. Third SPIN Workshop*, R. Langerak, ed., Twente Univ., The Netherlands, Apr. 1997.

[56] R.M. Needham and A.J. Herbert, *The Cambridge Distributed Computing System*. London: Addison-Wesley, 1982.

[57] D. Peled, "Combining Partial Order Reductions with On-The-Fly Model-Checking," *Proc. Sixth Int'l Conf. Computer Aided Verification (CAV94)*, pp. 377-390, Stanford, Calif., *Lecture Notes In Computer Science 818*. Springer-Verlag, 1994.
[58] D. Peled, "On Projective and Separable Properties," *Colloquium on Trees in Algebra and Programming*, pp. 291-307, Edinburgh, Scotland, *Lecture Notes In Computer Science* 787. Springer-Verlag, 1994.
[59] R. Pike, D. Presotto, K. Thompson, and G.J. Holzmann, "Process Sleep and Wakeup on a Shared-Memory Multiprocessor," *Proc. the Spring EurOpen Conf.*, pp. 161-166, Tromso, Norway, 1991.
[60] A. Pnueli, "The Temporal Logic of Programs," *Proc. 18th IEEE Symp. Foundations of Computer Science*, Providence, R.I., pp. 46-57, 1977.
[61] J.P. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in Cesar," *Proc. Fifth Int'l. Symp. Programming*, pp. 195-220, *Lecture Notes In Computer Science 137*. Springer-Verlag, 1981.
[62] B. Rahardjo, "SPIN as a Hardware Design Tool," *Proc. First SPIN Workshop*, J.-Ch. Gregoire, ed., INRS Quebec, Canada, Oct. 1995.
[63] M. Raynal, *Distributed Algorithms and Protocols*. New York: John Wiley & Sons, 1992.
[64] L.M. Ruane, "Process Synchronization in the UTS Kernel," *Computing Systems*, *Proc. Usenix Conf.*, vol. 3, no. 3, pp. 387-421, 1990.
[65] T.C. Ruys and R. Langerak, "Validation of Bosch's Mobile Communication Network Architecture with SPIN," *Proc. Third SPIN Workshop*, R. Langerak, ed., Twente Univ., The Netherlands, Apr. 1997.
[66] T.S. Chan and I. Gorton, "Formal Validation of a High Performance Error Control Protocol Using SPIN," *Software, Practice and Experience,"* vol. 26, no. 1, pp. 105-124, Jan. 1996.
[67] S. Shukla, D.J. Rosenkrantz, and S.S. Ravi, "Simulation and Validation of Self-stabilizing Protocols," *Proc. Second SPIN Workshop*, Rutgers Univ., New Brunswick, N.J., DIMACS/32, Am. Math. Soc., Aug. 1996.
[68] *Proc. First SPINWorkshop*, J.-Ch. Gregoire, ed., INRS Quebec, Canada, Oct. 1995.
Proc. Second SPIN Workshop, J.-Ch. Gregoire, G.J. Holzmann, and D. Peled, eds., Rutgers Univ., New Brunswick, N.J., DIMACS/32, *Am. Math. Soc.*, Aug. 1996.
Proc. Third SPIN Workshop, R. Langerak, ed., Twente Univ., The Netherlands, Apr. 1997.
[69] M. Staskauskas, "Tales from the Front: Industrial Experience with Formal Validation," *Proc. First SPIN Workshop*, J.-Ch. Gregoire, ed., INRS Quebec, Canada, Oct. 1995.
[70] W.T. Strayer, B.J. Dempsey, and A.C. Weaver, *XTP—The Xpress Transfer Protocol*. Reading, Mass.: Addison-Wesley, 1992.
[71] A. Tanenbaum, *Computer Networks*, second edition. Englewood Cliffs, N.J.: Prentice Hall, 1989.
[72] R.E. Tarjan, "Depth First Search and Linear Graph Algorithms," *SIAM J. Computing*, vol. 1, no. 2, pp. 146-160, 1972.
[73] W. Thomas, "Automata on Infine Objects," *Handbook of Theoretical Computer Science*, J. Van Leeuwen, ed., pp. 133-187, Elsevier Science, 1990.
[74] S. Tripakis and C. Courcoubetis, "Extending PROMELA and SPIN for real-time," *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS96)*, pp. 329-348, Passau, Germany, *Lecture Notes In Computer Science 1,055*. Springer-Verlag, Mar. 1996.
[75] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back, "Formal Methods: A Practical Tool for OS Implementers," *Proc. HotOS-VI, Sixth IEEE Workshop Hot Topics in Operating Systems*, Cape Cod, Mass., May 1997.
[76] R. Nalumasu and G. Gopalakrishnan, "Explicit Enumeration Based on Verification Made Memory Efficient," *Proc. Conf. Computer Hardware Description Languages (CHDL'95)*, pp. 617-622, Chiba, Japan, 1995.
[77] M.Y. Vardi and P. Wolper, "An Automata-Theoretic Approach to Automatic Program Verification," *Proc. First IEEE Symp. Logic in Computer Science*, pp. 322-331, 1986.
[78] M.Y. Vardi and P. Wolper, "Reasoning About Infinite Computations," *Information and Computation*, vol. 115, pp. 1-37, 1994, appeared as a conference paper in 1983.
[79] W. Visser, "Memory Efficient Storage in SPIN," *Proc. Second SPIN Workshop,* Rutgers Univ., New Brunswick, N.J., DIMACS/32, Am. Math. Soc., Aug. 1996.
[80] A.S. Wenban, J.W. O'Leary, and G.M. Brown, "Codesign of Communication Protocols," *Computer*, vol. 26, no. 12, pp. 46-52, Dec. 1993.
[81] C.H. West, "General Technique for Communications Protocol Validation," *IBM J. Research and Development*, vol. 22, no. 3, pp. 393-404, 1978.
[82] P. Wolper, "Expressing Interesting Properties of Programs in Propositional Temporal Logic," *Proc. 13th ACM Symp. Principles of Programming Languages*, pp. 148-193, St. Petersburg, Fla., Jan. 1986.



Gerard J. Holzmann received an MSc degree in electrical engineering in 1976 and a PhD degree in technical sciences in 1979 from the University of Technology in Delft, The Netherlands. He joined the Computing Sciences Research Center at Bell Laboratories, Murray Hill, New Jersey, in 1980, where today he is a distinguished member of technical staff (DMTS) in the Computing Principles Research Department. In 1980, Holzmann wrote one of the first automated protocol verification systems, called PAN Since then, he has written several other on-the-fly verification systems for a variety of applications. His latest system, SPIN, is considered to be one of the most efficient and most widely used LTL model checking systems Dr. Holzmann has written books on digital image editing, on the history of communications, and on protocol verification. He serves as an editor for the journal, *Formal Methods in Systems Design* (Kluwer).