Designing Bug-Free Protocols With SPIN

Gerard J. Holzmann

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

SPIN is an efficient, automated verification tool that can be used to design robust software for distributed systems in general, and bug-free communications protocols in particular. This paper outlines the use of the tool to address protocol design problems. As an example we consider the verification of a published protocol for implementing synchronous rendezvous operations in a distributed system.

We also briefly review some of the techniques that SPIN employs to address the computational complexity of larger verification problems.

1. Introduction

After nearly two decades of development, protocol verification has moved from a relatively esoteric art that could be practiced by few, to a well-understood engineering technique, captured in powerful tools that are available to anyone.[1] Most of today's verifiers accept high level protocol design specifications as input, and can verify these effectively for a range of correctness requirements expressed in a general logic. The use of these tools can rule out a class of design errors that can be hard, if not impossible, to avoid when conventional design techniques are used.

The first verifiers were all based on relatively minor variations on exhaustive reachability analysis (cf., [Hajek78, West78, H81]). The use of algorithms for the verification of logic formulae for finite state structures was pioneered in the early eighties by Ed Clarke and Allen Emerson in the U.S., e.g., [CE81], and by Joseph Sifakis in France, e.g., [QS82]. Much work followed on complexity management techniques, both for approximate proof techniques, as in [H88], and for property preserving reductions, such as by the use of BDDs (Binary Decision Diagrams) in symbolic model checking, [McM93], or partial order reduction techniques in traditional model checking, e.g., [P94], [HP94].

In this paper we focus on the verification system SPIN, which exploits many of the advances sketched above. We give a tutorial example of a typical application to a protocol design problem, and then review the principles on which SPIN is founded. This leads us to a brief discussion of the *design for verifiability* paradigm that SPIN supports.

2. An Example

Designing code for a distributed system is hard, and trying to detect the flaws in an existing design with manual techniques can be even harder. It is virtually impossible to imagine all the event scenarios that might occur in a run of a system with concurrent agents, and to assure that all eventualities are dealt with properly. One can select, sometimes almost at random, any manually produced protocol design documented in the literature, and show the presence of serious flaws by applying mechanical verification tools.

In some cases, a conventionally designed protocol system can be so complex that its essential properties

1. The complete source to the verifier SPIN, for instance, is available via anonymous `ftp` from the machine `netlib.att.com`, in directory `/netlib/spin`.

can only be approximated, even with automated tools. One could argue that this in itself is a design flaw: a well-designed system is *provably*, and not *arguably*, correct. Nonetheless, many such systems do exist, and therefore also automated correctness approximation techniques have become an important part of verification systems. A powerful technique for correctness approximation, included as an option in SPIN, is the so-called *supertrace* or bitstate hashing technique [H88]. It has been applied successfully to several large industrial applications [cf. C91, C94, H94, BG95].

As an example of the application of an automated verifier we will consider a protocol for implementing synchronous rendezvous operations in a distributed system: a notoriously difficult problem. The design we consider is documented in [Raynal88, p. 96-103], which is based on [Ber80].

The first step in the verification process is to built a complete and unambiguous, formal specification in the input language of the verifier, consisting of a behavior specification and a requirements specification. The input language of SPIN, called PROMELA (Protocol Meta Language), is based on Dijkstra's proposal for a high level, nondeterministic guarded command language [Dijk75], and it borrows some familiar syntax from Hoare's language CSP [Hoare78]. Other than CSP, PROMELA supports both synchronous and asynchronous message passing primitives on user-defined channels [H91], and it supports dynamic process creation.

The purpose of the example protocol is to enforce the semantics of synchronous operations by using only asynchronous message passing primitives. The informal description of the protocol, reproduced from [Raynal88, p. 101], is shown in Figures 1a and 1b.
The protocol works as follows. When a process is ready to engage in a rendezvous handshake with a peer process, it prepares a list (`listcom`) of all the events that it is prepared to handle. Typically, the process will be prepared to perform any one of several possible handshakes, with any one of several concurrently executing processes. Each process will inspect one event at a time, and poll the prospective partner in the handshake with a message of type `request` for its willingness to participate. The response to such a poll is one of three possible messages: `Y`, `N`, or `busy`.

The response `Y` means that the rendezvous offer is accepted, `N` means it was rejected permanently, and `busy` means that the other process is temporarily unable or unwilling to decide. In the first case, the list of possible events can be cleared, the process can perform the selected handshake and proceed with its computation. In the second case (a `N` response), the process will attempt to select another possible candidate for a rendezvous, and repeat the solicitation process. If no more options are open, the rendezvous attempt can be assumed to have failed. In the third case (a `busy` response), the rendezvous option is placed in a separate list, called `repeat`, for a possible later attempt.

While the negotiation for rendezvous handshakes is in progress, independent `request` messages from other processes may also arrive.

- When a request arrives that matches the offer that is in progress, it is accepted with a `Y` response.
- When a request arrives that is incompatible with the offer in progress, and also incompatible with all other options in `listcom`, it is rejected with a `N` response.
- If the request does not match the offer, but does match one of the other options in `listcom` it triggers the reply `busy` when the sender has a higher process id than the receiver, and otherwise the response is delayed by storing the incoming request in a list `delayed`. The asymmetry is meant to break causal cycles among the processes, and prevent starvation.

Formalization

The detailed formalization in PROMELA of the relevant part of this protocol can be found in the Appendix. It contains the specification of the solicitation process, formalized as a `proctype` declaration named `agent`. A user process that is ready to perform a rendezvous operation with another process will first insert all the various options it can accept for rendezvous into a fifo buffer, which is modeled here as a standard asynchronous message channel `listcom`. Once all the options have been appended to `listcom`, the user process instantiates a copy of the `proctype agent` (as an autonomous process), providing it with an index `pid` into the array that holds the `listcom` channels for all processes.

The `agent` process starts out in the `Active` state, as recorded in its local variable `state`. The 'continue

```
when set of choices reached do
     construct listcom;
     delayed <- ∅;
     repeat <- ∅;
     (j,v) <- next (listcom)
     send (request, v, i, j) to j;
     state <- enquiring;
end do;
received (Y,j,i) do
     state <- active;
     (communication between P_i, P_j established)
end do;
received (N,j,i) do
     (continue questioning);     [see Figure 1b]
end do;
received (busy,j,i) do
     repeat <- repeat ∪ {(j,v)};
     (continue questioning);
end do;
received (request,v',k,i) do
     case
     state == active then
          send (N,i,k) to k;
     state == waiting then
          if (∃ (k,v) ∈ listcom
               and compatible ((k,v),(i,v')))
          then begin
               send (Y,i,k) to k;
               state <- active;
               end
          else
               send (N,i,k) to k
          end if;
     state == enquiring then
          case
          j == k and compatible ((j,v),(i,v'))
               then begin
                    send (Y,i,k) to;
                    state <- active;
               end
          j != k and !(∃ (k,v) ∈ listcom
                    and compatible ((k,v),(i,v'))
               then
                    send (N,i,k) to k;
          j != k and ∃ (k,v) ∈ listcom
                    and compatible ((k,v),(i,v')
               if (k > i)
               then
                    send (busy,i,k) to k
               else
                    delayed <- delayed (k,v)
               end if
          end case
     end case
end do;
```

Figure 1a -- Example Protocol [Raynal88] -- First Part

questioning' procedure is placed at the top of the agent's behavior, after a label named `question`. When executed, the first option is removed from `listcom`, and sent to the target peer process identified in that option. The state changes to `Enquiring`, and the agent will wait for a response. There is an implicit assumption here that the response will inevitably come. If message loss is possible, this would mean that the agent process could deadlock. The correctness of the protocol therefore requires absence of message loss, and we will assume so in the verification.

```
(continue questioning):
     if listcom != ∅
     then  begin
           (j,v) <- next listcom;
           send (request,v,i,j) to j;
           end;
     else if delayed != ∅
     then  begin
           (k,v) <- next delayed;
           delayed = delayed - {(k,v)};
           send (Y,i,k) to k;
           ∀ j: (j,x) ∈ delayed, send (N,i,j) to j;
           state <- active;
           end
     else if repeat != ∅
     then  begin
           (k,v) <- next repeat;
           repeat <- repeat - {(k,v)};
           send (request, v, i, k) to k;
           end;
     else
           state <- waiting;
     end if;
```

Figure 1b -- Example Protocol [Raynal88] -- Second Part

When a `Yes` response from the peer process arrives, the state switches to `Active`. The remaining data stored in `delayed`, `repeat`, and `listcom` becomes irrelevant at this point. In the informal description this data is not removed until the next time that the protocol agent is invoked for a rendezvous. The formalized version removes it as soon as the `Active` state is reached, and then terminates its session. The agent process will be reinstantiated when the next rendezvous point is reached by the user, working with only the new options that are then available. The informal description in Figure 1 is ambiguous about the precise way in which one terminates or restarts the agent sessions.

Ambiguities

In the formalization, a session is also terminated when the last possible option for a rendezvous handshake has been rejected by the other agents. In that case the required synchronization has failed, and the calling user process should be blocked. There is no provision in the informal description, or in the formalized one, for notifying the user process of such an event. The informal description appears to leave the agent process in state `Waiting`, where it will continue attempts to match incoming requests against a now unavoidably empty set of options in `listcom`.

Another point of concern in the informal description is that no interaction is specified when a process is not currently prepared to engage in rendezvous operations, but is performing local computations instead. We may speculate that the response to any incoming request in such a state should be `busy`, provided that the user will be able to engage in rendezvous actions later, and it should be `N` if it will not be able to do so.

A more serious point of concern is that the process is required to respond with a definite `N` whenever it is currently not able to perform a specific rendezvous handshake, even if it might be prepared to perform such an event later, after, for instance, first completing handshakes with other processes. It is not hard to imagine scenarios that lead to system deadlock in these cases, despite the claims that such would be impossible with this type of protocol.

All the above can easily been gleaned from a static inspection of the code itself, especially if one is forced to formalize and disambiguate the informal text. It has often been observed that merely the attempt to formalize an informal design can pay off by uncovering the unclarities and ambiguities more quickly. The above arguments can serve to illustrate this point.

There are, however, also other problems with the protocol that are harder to settle by inspection. The pseudo code from Figure 1a, for instance, distinguishes between three cases when a `request` message arrives while the process is in state `enquiring`. One case is missing:

(A) `j == k` **and** `! compatible((j,v),(i,v'))`

The implication appears to be that this specific case cannot be reached, and therefore need not be specified. It is hard to decide whether or not this assumption is justified. Before reading on, take a few minutes to inspect the code and see if you can decide whether the case is redundant or not.

Verification

There are several ways to proceed with the verification, once the basic protocol has been formalized. A direct method would be to define a system of a fixed number of user processes, seed these processes with specific sets of rendezvous options, and perform the reachability analysis. This approach is flawed in several ways, though. First, we cannot guess which particular sets of rendezvous options may lead to errors. Second, we do not know if there is a minimum number of user processes that may be required to trigger a failure.

Although the verification process is restricted to finite state systems, this does not necessarily mean that we cannot perform any verifications that hold for arbitrary numbers of user processes. A useful technique is to consider the complete formalization of only a single user process, and its agent, and then model its interactions with all other, potentially infinitely many, user and agent processes with a single generalized environment process. If done right, such an environment process will capture the *worst case* behavior of the real environment. This means that there will be no possible interaction that the real environment could perform, that the formalization cannot perform, though not necessarily the reverse.

If an error is detected in such a system, we need to consider whether the real system would also be capable of performing the behavior that is required from the worst-case environment. If this is not the case, we can choose to refine the environment model and repeat the verification process. If no errors are detected, we can be certain that the real system is also error-free, since the modeled environment can perform at least all the interactions that the real environment might perform. The environment process in this method can be called a 'worst-case' or a 'generalized' [H91] automaton, or formally a 'maximal' automaton [e.g., BV95].

It is not hard to come up with a realistic worst-case formalization for the environment for the rendezvous protocol: all we need is a process that accepts any message that our agent process transmits, to any of the channels, and that responds randomly with `Y`, `N`, or `busy` messages, or that can generate some random `request` messages of its own.

A run of the verifier for that formal model can settle the ambiguity **(A)** instantly. It produces the following trivial scenario:

> Process i sends a request v to process k, offering data.
> Process k sends a request v' to process i, also offering data.

Or, in the output format[2] from SPIN:

```
. . .
 13: proc 2 (agent) line  98 "rv" Recv 2,Snd <- listcom[0]
 14: proc 2 (agent) line 100 "rv" Send request,Snd,0 -> channels[2]
. . .
 19: proc 2 (agent) line 151 "rv" Recv request,Snd,2 <- channels[0]
MSC: Cannot Happen
spin: line 221 "rv", Error: assertion violated
```

In this scenario, processes *i* and *k* are both prepared to initiate a rendezvous with each other, and offer data through `Snd` offers, but neither is able to accept the data from the other party with a matching `Rcv` offer. The missing case is reached in just a few steps. In retrospect, this counter example is so simple that it looks like it *could* easily have been detected by inspection. There is so much extraneous detail, though, that it is very hard to do so reliably. For a mechanical verifier, the task is hardly challenging. The complete reachable statespace for the formal model of this rendezvous protocol contains approximately 20,000 states, which can be searched exhaustively by SPIN in a few seconds of CPU time (we used a small 33 Mhz SGI-

2. SPIN can produce the scenarios in graphical form as message sequence charts, or in plain textual form, as shown here.

Indigo computer).

The benefits of formal verification, then, come from two different aspects:

- The user is forced to prepare an unambiguous and complete specification of the design. In the process, unclarities or flaws will unavoidably be detected.
- The more subtle types of interaction errors can usually be found quickly with the mechanical verifier. Failing that, a rigorous proof of the soundness of the basic design is obtained.

In the next section we briefly summarize how the verifier SPIN itself operates.

3. The Verifier

A PROMELA program is a textual representation of a formal labeled transition system. Each concurrent component (e.g., a protocol entity) has a well-defined control state, and each primitive statement that is executed can change this control state in a well-defined way. Each statement thus corresponds to a transition arc in the underlying labeled transition system. There is a rich theory available for establishing properties of labeled transition systems.

The semantics of PROMELA is founded on the notion of *executability*. For each type of primitive statement, a language rule defines whether the statement is executable or blocked in a given system state. This rule is comparable to the definition of a weakest precondition on the statement execution. Another rule determines precisely how the statement will modify the current system state if and when it is executed. As one small example: a condition such as `(a > b)` is a valid primitive statement in PROMELA: it is executable when the condition holds, and it has no effect on the system state when executed (other than the changing of the control state of the process executing the statement). A send action, similarly, is executable if the destination channel is currently non-full, and a receive action is executable if the source channel is non-empty.

The behavior of a system of processes is formally defined as the asynchronous interleaving product of all the component labeled transition systems (there is one such labeled transition system for each active process). Each step in the interleaving defines one global system state, that is, one unique control state for each component, and a specific value assignment to each data object and data channel. Processes can interact by manipulating shared data objects and message channels. In PROMELA, message passing through channels can be defined to be either synchronous or asynchronous, using predefined operations on finite, typed message buffers.

SPIN can compute the asynchronous interleaving of a set of concurrent processes in several different ways. The default algorithm is a standard exhaustive reachability analysis by depth-first search [H91].

Reduction Technique

An efficient partial order reduction algorithm, based on the theory from [HP94], is also available, and can be used to significantly reduce the complexity of the verification process, without affecting its scope [P94]. The effect of the partial order reduction method can be a reduction in the number of reachable system states by several orders of magnitude. Figure 2 shows a measurement of the number of reachable states that has to be generated to complete the verification for a formal model of a leader election algorithm [DKR82]. It illustrates a best-case performance of the reduction algorithm, where an exponential growth in complexity is reduced to a linear growth, with growing numbers of processes participating in the protocol.

An important characteristic of the partial order reduction method implemented in SPIN is that it can never lead to an increase rather than a decrease in computational requirements of the verification process. The same is not true, for instance, for BDD (Binary Decision Diagram) based methods.

Correctness Approximations

For problem sizes that preclude exhaustive verification because of their size, a high-coverage approximation method is also implemented in SPIN. The approximation technique minimizes the memory requirements to a small fraction of what would be needed for an exhaustive search, and it maximizes the error detection probabilities [H88],[H95].

If one reachable system state requires *S* bytes of memory to store, and our machine has *M* bytes of memory

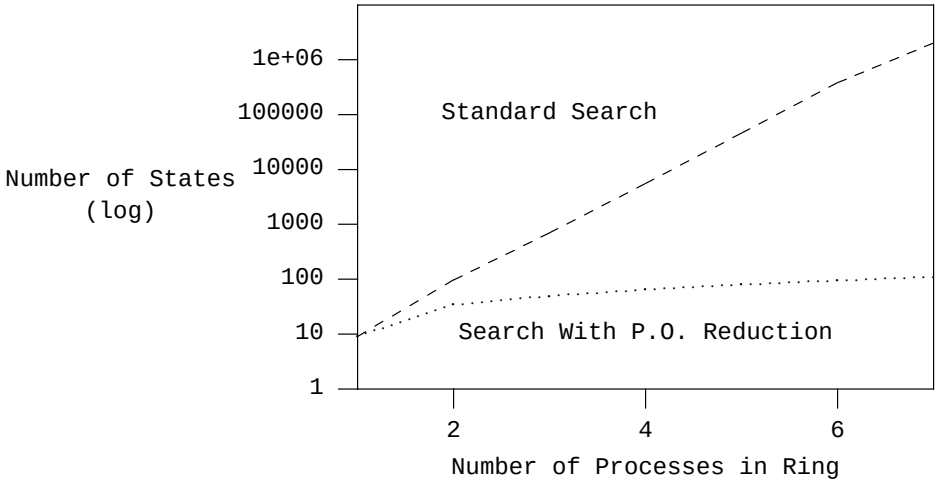

Figure 2 -- Effect of Partial Order Reduction in SPIN
(Leader election protocol for a uni-directional ring [DKR82])

available, inevitably the verifier will run out of memory after generating M/S states. If the total number of reachable states $R > M/S$ states, then the *problem coverage* of that verification run is $M/(R*S)$.

This means that if 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% of the reachable states are inspected). Can we do better?

Fortunately, there is a better way. SPIN contains an implementation of the *supertrace* or *bitstate hashing* technique [H88],[H95], that can move the expected problem coverage in the above case arbitrarily close to unity (meaning 100% coverage), without changing the memory limits. The algorithm uses just two bits of memory per reachable state, and allows the expected coverage to be calculated with a statistical argument. The supertrace technique has been applied successfully in a number of large scale industrial applications, e.g. [C91,H94].

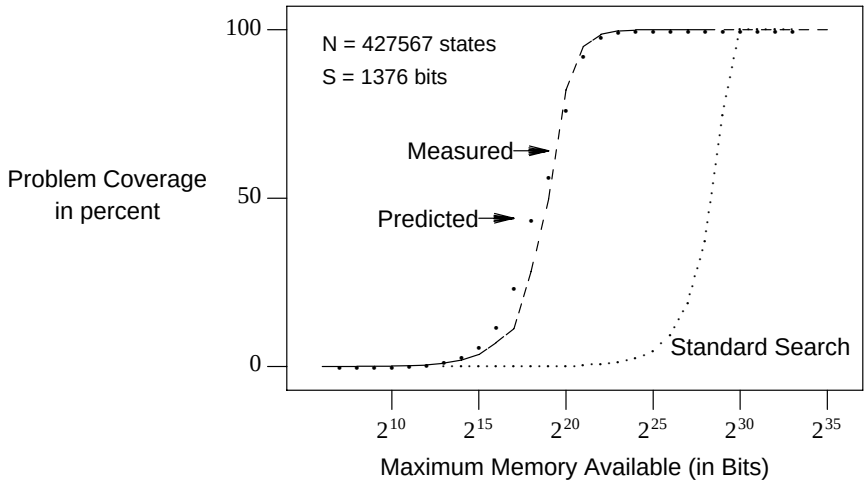

Figure 3 -- Measured and Predicted Problem Coverage [H88],[H95]
Bitstate Hashing Verification Approximation Algorithm in SPIN

The effect of the bitstate hashing technique on problem coverage is illustrated in Figure 3. In this case, the algorithm was applied to a data transfer protocol that requires the generation of approximately 427 thousand states (N), each taking 1.3 thousand bits of memory (S) to store, for a complete verification. The total memory requirements for a standard search are trivially at least $N \times S$, or close to about 73 Megabytes of memory (about 2^{29} bits). The Figure shows that with only 1 percent of the memory required for an

exhaustive search, the bitstate hashing technique can still realize an effective problem coverage close to 100 %. For still lower amounts of memory, the coverage for a single verification run drops to zero. It may be boosted up again by performing multiple runs with statistically independent hashing functions [H95]. When $M < N \times S$, the bitstate hashing technique always realizes greater problem coverage than a standard search.

Formalizing Requirements

Several basic types of correctness requirements can be standardized, and are built-in to the verification engine of SPIN. Examples are freedom from deadlock, starvation, and buffer overflow, and automatic detection of unspecified receptions, or the logical incompleteness of a specification. All specifications can be expected to fulfill those requirements, so the verification process for these can be predefined.

The user can also add formal assertions to the model, to prove or disprove the validity of process or system invariants, as was done in the example protocol for implementing rendezvous. In this case, we used an assertion to prove that a specific code fragment that should have been unreachable, was in fact reachable.

More sophisticated types of correctness requirements can be specified in the syntax of Linear Time Temporal Logic (LTL). LTL can be used to express both safety and liveness properties of either finite or infinite execution sequences. For instance, the LTL property

```
□ ( p U q )
```

states that it is ‘‘always’’ (the □ operator) guaranteed that the condition named `p` remains true at least ‘‘until’’ (the `U` operator) the condition named `q` becomes true. Similarly

```
□ ◊ p
```

states that no matter at which point in an execution we are (‘‘always’’, □) it is guaranteed that ‘‘eventually’’ (the ◊ operator) the condition `p` will become true at least once more. Other operators that can be used in LTL formula are standard boolean negation, conjunction, disjunction, and logical implication.

A useful property for the rendezvous protocol from the previous sections could be, for instance:

```
□ ( request(i,j,Snd) / request(j,i,Rcv) -> ◊ rendezvous(i,j) )
```

where the `request(...)` and `rendezvous(...)` are macros representing boolean propositions on system states, to be defined separately. The above requirement then can be used to express that two matching requests from processes `i` and `j` must eventually lead to a successful rendezvous. We already know that this property does not hold for the protocol as specified.

The use of temporal logic for expressing properties of distributed systems was first proposed by Amir Pnueli, in the late seventies [Pnueli77]. By the mid eighties it had become a generally accepted formalism. As far as we know, SPIN is the first generally available on-the-fly verification system that directly supports LTL.

There are also several other, closely related, formalisms and logics available today in verification systems, such as especially the computation tree logic CTL developed by Clarke, Emerson, Sistla and others, e.g., [CES86].

4. Computational Complexity

The size of the interleaving product that SPIN must compute, in the worst case, can 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 *R*, both in (cpu) time and in (memory) space. To prove simple liveness properties, such as absence of starvation or of non-progress cycles, requires twice as much time, but only insignificantly increase in memory (two bits per reachable state extra [H91, CVWY92, GH93]). To prove LTL properties, the time requirements increase by a factor that can, in the worst case, itself again increase exponentially with the number of temporal operators that is used in the formula. The space requirements remain largely

unaffected. Fortunately, meaningful LTL properties very rarely have more than two or three operators, so the increase in complexity introduced here remains manageable.

5. Design for Verifiability

A protocol that is designed without concern for either manual proof or automated verification can quickly become so complex that not even automated approximation techniques will be able to reasonably assess its properties. It can, therefore, be argued that a design should be considered fatally flawed if it cannot be verified, at least at some level of abstraction. This approach can be named ''design for verifiability,'' and is summarized as follows.

1. The designer is asked to make a clean distinction between requirements and behaviors, and to specify these two distinct parts of the design in an unambiguous and formal way. This formal version of the design serves the role of an engineering prototype for the final design.
2. The designer is asked to use the formal prototype to verify its properties. The requirements and the behaviors can first both be checked on their internal logical consistency. Next they can be checked against each other for consistency and completeness.
3. Only if the first two steps were successfully completed can the design be refined toward more detailed code, and ultimately into an implementation.

A good design method, therefore, requires the availability of unambiguous, and independent, notations for both behavior specifications and requirements. It also requires the availability of a rigorous methodology, for performing verifications. Ideally that methodology is tool-based.

The use of the language PROMELA for producing high-level specifications, the logic LTL for formalizing requirements, and the tool SPIN to perform the verifications is meant to match this paradigm of design.

The main purpose of a formal method is to enable a designer to document design decisions unambiguously, at a relatively high level of abstraction, and to enable him/her to analyze the design decisions, well *before* the implementation stage is reached. Especially for the production of software that is to perform reliably in a distributed system, this type of design method can be more reliable, reproducible, and effective than conventional ad hoc, a posteriori testing methods.

There will not be many software design problems that would defy treatment in a more formal manner. The approach is called *design for verifiability* because it assures that verification can be performed at least at the first few levels of abstraction. Major structural or logical design errors are most easily caught and repaired by applying rigor in the early phases of design, and disproportionally hard in the final phases. By contrast, the dominant design technique in use today is best described as *design by trial and error*: one uses mostly informal arguments in the early phases of design, and attempts to perform rigorous testing only in the final phases of the design, where it is most difficult and costly to do so.

6. Conclusions

The number of both academic and industrial applications of formal verification is steadily growing. Though this is encouraging, a sobering thought is that the number of algorithms and systems that is designed without any benefit from verification tools still completely dominates the industry. In part this is due to the perceived scope of automated verification techniques, which has always been somewhat restricted.

Today, however, tools like SPIN can solve significant problem sizes efficiently. The tool was explicitly designed to require no detailed knowledge of proof theory, verification algorithms, or formal methods theories from its users. The tool can either be used stand-alone, or in combination with a general graphical interface that presents the user with an overview of verification or simulation options at each step. Routine verification tasks, then, can become as simple as a series of button clicks. With this new class of tools, the creative efforts of the designer need no longer be spent on the mechanics of verification, but instead can be focused on the core design issues: the construction of proper design abstractions, and the formalization of necessary and sufficient correctness requirements.

SPIN is based on the principle of *on-the-fly* verification, first discussed in [H85,H87,H88]. It also includes support for efficient approximations of verification problems for large problem sizes [H88,H95], for partial

order reduction techniques [HP94,P94], and for the formalization of correctness requirements in Linear Time Temporal Logic, based on the algorithm from [GPWV95].

SPIN is only one from a range of verification systems that is either already available, or that is actively being worked in academia or in industry. (An up-to-date list of formal methods tools, for instance, can be found on the internet at `http://www.comlab.ox.ac.uk/archive/formal-methods.html`.) If in the next few years the use of verification tools spreads sufficiently, the adoption of protocol designs that have never been verified may well be frowned upon as inconceivably courageous; a thing of the past.

7. References

[BV95] O. Bernholz (Kupferman), M. Vardi, 'On the Complexity of Branching Modular Model Checking,' *Proc. CONCUR95 Conference*, Philadelphia, August 1995.

[Ber80] A.J. Bernstein, 'Output guards and non-determinism in CSP,' *ACM Toplas*, Vol. 2, No. 2, Apr. 1980, pp. 234-238.

[BG95] B. Boigelot and P. Godefroid, 'Model checking in practice: an analysis of the ACCESS.bus(tm) protocols using SPIN,' report Univ. of Liege, Belgium and AT&T Bell Laboratories, September 1995, submitted for publication.

[C94] Cattel, T. (1994) 'Modelization and verification of a multiprocessor realtime OS kernel,' *Proc. 7th FORTE Conference*, Bern, Switzerland, pp. 35-51.

[C91] J. Chaves, 'Formal methods at AT&T, an industrial usage report,' *Proc. 4th FORTE Conference*, Sydney, Australia, 1991, pp. 83–90.

[CE81] E. M. Clarke, E. A. Emerson, 'Synthesis of Synchronization Skeletons for Branching Time Temporal Logic,' *Proc. Logic of Programs: Workshop*, Yorktown Heights, NY, May 1981, LNCS, Vol 131, Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, A. P. Sistla, 'Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications,' *Toplas*, Vol 8, No 2, pp 244-263, 1986.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, 'Memory efficient algorithms for the verification of temporal properties,' *Formal Methods in Systems Design*, Vol I, 1992, pp. 275–288.

[DKR82] D. Dolev, M. Klawe, and M. Rodeh, 'An O(n log n) unidirectional distributed algorithm for extrema finding in a circle,' *Journal of Algorithms*, Vol 3. (1982), pp. 245-260.

[Dijk75] E.W. Dijkstra, 'Guarded commands, nondeterminacy and formal derivation of programs,' *Comm. of the ACM*, Vol. 18, No. 8, pp. 453-457.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, P. Wolper, 'Simple on-the-fly automatic verification of linear temporal logic,' *Proc. IFIP/WG6.1 Symp. on Protocol Specification, Testing, and Verification*, PSTV95, Warsaw, Poland, June 1995.

[GH93] P. Godefroid and G. J. Holzmann, 'On the verification of temporal properties,' *Proc. IFIP/WG6.1 Symp. on Protocol Specification, Testing, and Verification*, PSTV93, Liege, Belgium, June 1993.

[Hajek78] J. Hajek, 'Automatically verified data transfer protocols,' *Proc. 4th International Computer Conference*, Kyoto, Japan, 1978, pp. 749-756.

[Hoare78] C.A.R. Hoare, 'Communicating sequential processes,' *Comm. of the ACM*, Vol. 21, No. 8, pp. 666-677.

[H81] G.J. Holzmann, 'An algebra for protocol validation,' *Proc. First IFIP/WG6.1 Symp. on Protocol Specification, Testing, and Verification*, PSTV81, NPL, Teddington, England, 1981.

[H85] G.J. Holzmann, 'Tracing protocols,' *AT&T Technical Journal*, Vol 64, December 1985, pp. 2413-2434.

[H87] G.J. Holzmann, 'Automated protocol validation in Argos, assertion proving and scatter searching,' *IEEE Trans. on Software Engineering*, Vol. 13, No. 6, June 1987, pp. 683-697.

[H88] G. J. Holzmann, 'An improved protocol reachability analysis technique,' *Software, Practice and Experience*, 18(2):137-161, 1988. First published in: Proc. PSTV87 Conference, Zurich, Sw. 1981.

[H91] G. J. Holzmann, *Design and validation of computer protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[H94] G. J. Holzmann, 'The theory and practice of a formal method: NewCoRe,' *Proc. 13th IFIP World Computer Congress*, Hamburg, Germany, 1994, pp. 35–44.

[HP94] G. J. Holzmann and D. Peled, 'An improvement in formal verification,' *Proc. 7th FORTE Conference*, Bern, Switzerland, 1994.

[H95] G. J. Holzmann, 'An analysis of bitstate hashing,' *Proc. IFIP/WG6.1 Symp. on Protocol Specification, Testing, and Verification*, PSTV95, Warsaw, Poland, June 1995.

[McM93] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publ., 1993.

[P94] D. Peled, 'Combining Partial Order Reductions with On-the-fly Model Checking,' *6th Int. Conf. on Computer Aided Verification*, Stanford, Ca., June 1994.

[Pnueli77] A. Pnueli, 'The temporal logic of programs,' *Proc. 18th IEEE Symposium on Foundations of Computer Science*, 1977, Providence, R.I., pp. 46-57.

[QS82] J. P. Queille and J. Sifakis, 'Specification and verification of concurrent programs in Cesar,' *Proc. 5th Int. Symposium on Programming*, LNCS Vol. 137, pp 195-220, Springer Verlag, 1982.

[Raynal88] M. Raynal, *Distributed algorithms and protocols*, Wiley, 1988, pp. 96-103.

[West78] C. H. West, 'General Technique for Communications Protocol Validation,' *IBM Journal of Research and Development*, Vol 22, No. 4, p. 393, 1978.

Appendix
Formal Model in PROMELA for the Example Protocol in Figures 1a an 1b (Fragment)

```
proctype agent(byte pid)
{       byte state = Active;
        byte ch, from, req;
        byte j, w, h;
        byte i, v, k;

question:       /* See Figure 1b */
        if      /* choice */
        :: listcom[pid]?j,w ->              /* retrieve element */
                channels[j]!request,w,pid;
                state = Enquiring

        :: empty(listcom[pid]) ->/* the list is empty */
                if      /* choice */
                :: delayed[pid]?j,w ->
                        channels[j]!Yes,w,pid;
                        do      /* iteration */
                        :: delayed[pid]?i,v ->
                                channels[i]!No,v,pid
                        :: empty(delayed[pid]) ->
                                break
                        od;
                        state = Active;
                        printf("MSC: Success0);
                        goto done

                :: empty(delayed[pid]) ->
                        if
                        :: repeat[pid]?j,w ->
                                channels[j]!request,w,pid

                        :: empty(repeat[pid]) ->
                                state = Waiting;
                                printf("MSC: Failed0);
                                goto done
        fi      fi      fi;

                /* See Algorithm in Figure 1a */
end:            /* a valid termination point */
        do      /* iteration */
        :: channels[pid]?Yes,v,i ->
                state = Active;
                printf("MSC: Success0);
                goto done

        :: channels[pid]?No,v,i ->
                goto question

        :: channels[pid]?busy,v,i ->
                repeat[pid]!i,v;
                goto question
```

```
        :: channels[pid]?request,v,k ->
                if
                :: state == Active ->
                        channels[i]!No,v,pid

                :: state == Waiting ->
                        /* listcom is empty; cannot really get here */
                        channels[i]!No,v,pid

                :: state == Enquiring ->
                        if
                        :: j == k && v != w ->    /* compatible */
                                channels[k]!Yes,v,pid;
                                state = Active;
                                printf("MSC: Success0);
                                goto done
                        :: j != k ->
                                h = len(listcom[pid]);
                                do
                                :: h > 0 ->
                                        listcom[pid]?from,req;
                                        if
                                        :: k == from && v != req ->
                                                break
                                        :: else ->
                                                listcom[pid]!from,req;
                                                h--
                                        fi
                                :: else ->
                                        break
                                od;
                                if
                                :: h > 0 ->
                                        if
                                        :: k > pid ->
                                                channels[k]!busy,v,pid
                                        :: else
                                                delayed[pid]!k,req
                                        fi
                                :: else ->
                                        channels[k]!No,v,pid
                                fi
                        :: else ->
                                printf("MSC: Cannot Happen0);
                                assert(0)      /* always false */
                fi      fi
        od;
done:
        /* clear all buffers -
         * responding No to all requests
         * that are pending in delayed[pid]
         */
}
```