

Software Verification at Bell Labs: one line of development

Gerard J. Holzmann

Collectors often greet the first report of a new type of minting error for commonly circulating coins with enthusiasm. A coin is a rare example of an object that can increase, rather than decrease, in value when it is faulty. In software design we are not so fortunate. Software faults are often intriguing, but they rarely increase the value of a product. Since the early days of computers, programmers have sought effective ways to defend against software bugs. Software verification techniques are meant to help the user locate possible defects in a software product reliably, and preferably mechanically. In this article we look at one line of research that has led to one of the most widely used verification systems for distributed software today.

Introduction

A telephone switch is built to some unusually strict reliability requirements. The average downtime, for instance, is not allowed to exceed 3 minutes *per year*. As we all know from experience, few software products today can satisfy such strict requirements. Rigorous testing of a nontrivial piece of software is perhaps even harder than designing and writing the software in the first place. This is not a new phenomenon. The first time that a *software crisis* was declared, in the late sixties, the typical size of a program was significantly smaller than it is today. The source code for one of the early versions of the Unix® operating system from 1973, for instance, counted 6,600 lines of C; the code size for the most widely used PC operating systems today exceeds this by a good three orders of magnitude.

But it is not just the average code size that has increased. Purely sequential applications, that have no significant interaction with concurrently executing applications elsewhere, have become rare. Typical applications today can interact with other applications to exchange data, and generally rely on each other's well-being to function correctly. Computer scientist Leslie Lamport once defined a distributed system succinctly as "*one in which a machine that you never knew existed can stop you from getting your work done.*"

If a medium size sequential application can challenge our reasoning skills, a large concurrent application can easily defy them. Executions in these systems are no longer deterministic: the outcome of an execution can depend on an irreproducible interleaving of asynchronously executing processes. This means that conventional software testing loses much of its value. A successful test gives no guarantee that the same test could not fail when repeated, and any defect uncovered by a test once may be impossible to reproduce later. In this article we therefore focus on one of the hardest problems in software design: verifying the logical correctness of systems of interacting concurrent processes. The most prominent examples of these applications are telephone switching systems, distributed operating systems, computer networking software, and data communications protocols.

Over the last two decades researchers at Bell Labs have made fundamental contributions to the development of verification tools for this class of applications. As an example we will discuss one main line of work at Bell Labs that has led to the development of one of the most widely used verification systems for distributed systems software today: the **Spin** model checker [H97]. The following sections briefly describe how a model checker for software works, and what makes it efficient. We briefly review the main theoretical advances that have been made, as well as significant improvements that have been made in the integration of these techniques into the standard software development process. We also illustrate the cases where software verification techniques systematically outperform conventional testing techniques within this realm of applications, both in thoroughness and in overall efficiency. Several other approaches to the software verification problem are also being pursued at Bell Labs. We refer especially to the work on Cospan [K94], Verisoft [GHJ98], and VFSM [FMGS97].

The First Tools

Before we can build a tool that can mechanically intercept software defects, we should be able to define what precisely a software defect *is*. Some types of defects will be clear: it should be impossible for a program to crash, hang, or corrupt data when it is executed. But it is not enough to just list some of the things that could *prevent* a program from performing its function: we need to state precisely what a program is meant to accomplish before we can thoroughly verify its operation. This requires a level of rigor in the formal treatment of properties and specifications that was not available until fairly recently. Among the first software applications that could be specified in a form that was amenable to automated verification, were data communications protocols. By the late seventies it had become standard to specify protocols formally as finite state automata. One of the early examples of this is the definition of the simple *alternating bit protocol*, which was proposed to improve the reliability of computer to computer communications over noisy channels [B69].

Table 1 – Alternating Bit Protocol, State Machine Specification

State \ Event	Transmit A0	Transmit A1	Receive B0	Receive B1	Receive Error
S0	S1	-	-	-	-
S1	-	-	S3	S2	S2
S2	S1	-	-	-	-
S3	-	S4	-	-	-
S4	-	-	S5	S0	S5
S5	-	S4	-	-	-

Table 1 shows a sample state machine specification for one party in an alternating bit protocol. The specification for the other half of the protocol is obtained by replacing A0 with B0, A1 with B1, and vice versa. A graphical representation of both automata is given in **Figure 1**. Protocol execution for the party specified in **Table 1** begins in state S0 and for its peer in state S1. The column entries specify the next state that is reached when the event specified in the column heading occurs. In state S0 the only option is to transmit a message of type A with sequence number 0 and then move to state S1. In S1 the protocol machine waits for a response of type B. If the message carries the sequence number 0, the protocol machine advances to state S3. If the sequence number is 1 or if the message is corrupted, the machine moves to state S2 instead and repeats the last transmission. In the absence of errors, the protocol machine will cycle through the states S0, S1, S3, S4, S0 forever. To achieve useful data transmission, a payload of data can be attached to each A and B message. New data is obtained only after the immediately preceding message was correctly acknowledged. In **Table 1**, this occurs at states S2 and S4. Note that every execution of this protocol defines a path in each of the graphs from **Figure 1**. The system execution as a whole can be seen as an interleaving in time of local steps in the two automata. In this case, the number of interleavings that can occur is relatively small. For larger systems, with many asynchronously executing processes, this number can be extremely large, making it very hard to reason about these systems.

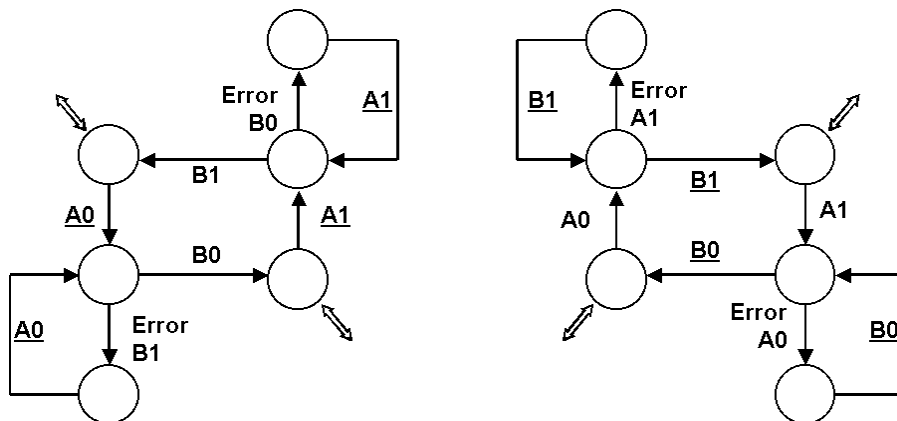


Figure 1.

Finite State Automata for the alternating bit protocol, as defined in 1969 in [B69].

The specification of a protocol in the form of an automaton, with simple and well-known semantics, made it possible very early on to devise equally simple automated means of analysis. In a commentary on the paper that first introduced the alternating bit protocol, published in 1969, W.C. Lynch wrote, with considerable foresight: “*I feel it should be possible to develop proof procedures for the reliability of automata...*” [L69]. Initially, though, the analytical techniques that were tried were mostly restricted to manual types of analyses. It took another ten years before work on automated tool support started in earnest.

Upon joining Bell Labs in 1980 the author of this paper was challenged by Doug McIlroy to find a way to prove the correctness of the software for a small phone system, called *TPC* (for *The Phone Company*), written by Lee McMahon. To solve that problem, we wrote a reachability analyzer for a small formal model of the code, and were able to mechanically intercept some subtle problems that had escaped detection in conventional testing. This analyzer, called *PAN* (for *Protocol Analyzer*), used a depth-first search algorithm to recursively explore the reachable states of the modeled system [T72], cf. **Figure 5**. Simple types of defects, such as system deadlocks, cases of incompleteness, buffer overflow, and the presence of unexecutable code fragments (dead code), were easily detected with this algorithm. The decision to perform the analysis *during* the search process, rather than *after* it, later proved to be a critical advantage in the analysis of large-scale systems. *On-the-fly verification* is the de facto standard for *software* model checking systems today.

Automata and Logic

Around 1983, another Bell Labs researcher at that time, Pierre Wolper, joined with Moshe Vardi in a study of a theory that could support the automatic verification of a broader range of system properties [WVS83, VW86]. The properties that Vardi and Wolper considered were specified in a logic that had been proposed by Amir Pnueli in the late seventies for reasoning about concurrent systems, and that was quickly gaining popularity at this time [P77]. This *temporal logic* allowed one to reason about system executions, by formally relating the occurrence of events in time. A simple example of a temporal logic formula that states a system requirement is the following:

$$(request \rightarrow (\diamond response))$$

The validity of a temporal formula is defined over, possibly infinite, system executions. This formula above uses two Boolean operands, *request* and *response*, two temporal operators \square and \diamond , and one logical operator \rightarrow . The box operator \square is pronounced *always*. The formula $\square p$, for instance, is true for a system execution σ if and only if the operand p remains invariantly true for all steps of σ . The box operator \diamond is pronounced *eventually*. The formula $\diamond p$ is true for execution σ if operand p becomes true at least once in system execution σ . The operator \rightarrow , finally, stands for logical implication: $(p \rightarrow q)$ means $(\neg p \vee q)$, i.e., either p is false or q is true.

The example formula expresses the property that it should *always* be the case in an execution that there is either no *request* at all, or if a *request* occurs there will *eventually* also be a *response*. If we wanted to exclude the possibility that no requests are issued at all, we can formulate an additional property:

$$\diamond (request)$$

which states that there will *always eventually* be a *request*, or we can combine the two properties into a single one, using the logical *and* operator \wedge , as:

$$(\diamond (request)) \wedge (request \rightarrow (\diamond response))$$

Examples of these types of requirements are readily found in practical applications. In the alternating bit protocol, for instance, we could formulate the requirement that the transmission of an A0 message (the request) is always eventually followed by the reception of a B1 message (the response). Similarly, in a telephone system, if a subscriber has call-forwarding, the arrival of an incoming call (the request) should always eventually be followed by the forwarding event (the response).

Automata Theoretic Verification

Vardi and Wolper showed that any property expressed in the formalism of temporal logic can be converted into an automaton that can then be used in an algorithmic verification process, provided that the system for which we want to check the property is also specified as an automaton. Assume we are given a system automaton S and a temporal logic formula f . We can check if S satisfies f with the following procedure. We first negate property formula f , using standard Boolean negation, and then derive the corresponding automaton $A(\neg f)$ using Vardi and Wolper's procedure. Note that while f makes a positive statement about the *correct* executions of our system, $\neg f$ captures the executions that are *not* correct. That is $\neg f$ expresses the potential *violations* of f .

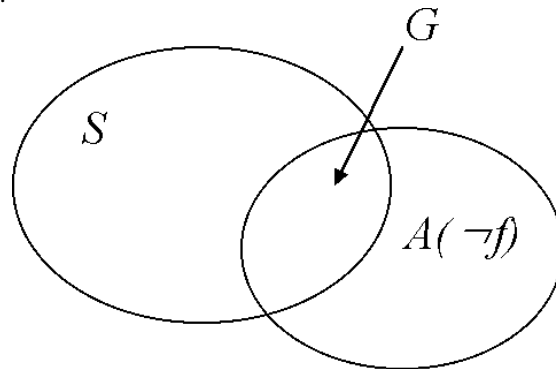


Figure 2.

S is an automaton that defines system behavior, and **A** an automaton that defines all possible violations of system requirement f , we can compute the intersection **G** of the languages defined by **S** and **A** to find executions in **S** that violate f . By showing that **G** is empty, we can prove that **S** satisfies f .

Next, we search for any execution that is allowed by *both* S and $A(\neg f)$, as illustrated in **Figure 2**. Formally this search amounts to the construction of a “synchronous product” G of S and $A(\neg f)$, written $G = S \otimes A(\neg f)$. If this synchronous product is non-empty, there exist executions of S that violate f . The existence of any one such execution suffices to disprove the validity of f , and, conversely, proving that there can be no such executions suffices to prove that f cannot be violated.

A number of practical issues, however, have to be solved before this method can be used in earnest.

1. When we verify a property of a system of asynchronously executing concurrent processes, the system automaton S is itself defined as an interleaving product of automata. In the alternating bit protocol, for instance, S would express the combined behavior of the two communicating parties. An efficient way to compute S from smaller components is therefore essential for this method to work.
2. For any serious application, e.g. the call processing software of a telephone switch, the system automaton S can be very large. The cost of verification can increase linearly with the size of both S and of $A(\neg f)$. This means that we need very good algorithms for reducing the sizes of S and $A(\neg f)$.
3. And finally, if we want to apply this method to the verification of software, we need good ways to relate program source code to automata specifications, covering not just control flow but also the use of data.

Since the automata theoretic method was first proposed, significant progress has been made on each of these issues. Together the improvements that have been made make it possible to apply the automata theoretic verification method to software verification problems of impressive size. We will briefly consider the algorithms that have been developed, and mention some of the recent applications of this technology.

Computing System Behavior

The verification procedure would be useless if it could not be applied to large problems. A typical application today, such as the verification of significant portions of the call processing software for a telephone switch [HS2000], can involve the concurrent execution of large numbers of potentially

interacting processes, each with access to considerable amounts of user data. If the individual behavior of each process can be captured in the form of an automaton, the joint behavior of all these automata defines system automaton S . The automaton S can be large and expensive to compute. If S is known to contain defects, though, it is wasteful to spend significant resources to compute it exhaustively if also a small initial portion can suffice to expose them. It is also redundant to compute all of S if only the portion that intersects with A ($\neg f$) is needed (cf. **Figure 2**). The verification system **Spin** uses an *on-the-fly* procedure to compute the intersection product $G = S \otimes A$ ($\neg f$) directly, i.e., without first separately computing S . The computation can stop as soon as the first counter-example to a logic property has been generated, which often happens quickly after the search begins. But even if no defects are to be found in S , it is only necessary to compute the fragment of S that is relevant to the verification of property f . The method that used to compute $G = S \otimes A$ ($\neg f$) in **Spin** is an optimized nested depth-first search algorithm [CVWY92].

Reduction Methods

In some cases it can be too expensive to compute even the fraction of $G = S \otimes A$ ($\neg f$) that is needed to decisively prove or disprove f . If S contains defects they are not necessarily present in the fraction of G that has been computed by the time available resources, such as time or memory, run out.

The graph in **Figure 3** illustrates some of the improvements in the performance that have been achieved in the solution of this problem. As an example we used the model of the *TPC* phone switch software from 1980 (**A** in **Figure 3**) that triggered the first work in this area at Bell Labs. We'll review the main reduction methods below.

The first method is based on the use of dedicated data structures that can store the reachable states from the product automaton G frugally. For a small runtime penalty one can use standard memory compression techniques. Clearly, the more aggressive the compression method is, the larger the fraction of G is that can be computed, and the more thorough the search for defects can be. The *supertrace* algorithm (**B**), introduced in 1987, pushes this to a limit by using lossy compression to record each reachable state from G in just two bits of memory [H87]. The supertrace algorithm uses the fact the bit-addresses also carry information, so that effectively, for the price of n bits one can record up to $32 \times n$ bits of information per state (assuming an address length of 32 bits). Because only minimal information is recorded, the supertrace algorithm also speeds up the verification process. Even in cases where the full size of G far exceeds available resources, the supertrace algorithm can still approximate the results of an exhaustive verification in very short time. The algorithm is today used for large problem sizes in almost all verification tools that have been developed.

Another method to reduce the cost of verification is based on the observation that the full product G contains considerable redundancy. A system of concurrently executing processes allows an overwhelmingly large number of *possible* executions. Many of these executions, however, differ in unimportant ways, e.g. in the way that logically independent actions are interleaved in time. In the late eighties and early nineties several people, including Bell Labs researchers Patrice Godefroid and Doron Peled, developed theories that could be used to capture the fraction of the system executions that should minimally be considered to verify a concurrent system. In typical examples, the application of these *partial order reduction* strategies can result in a reduction in the fragment of G that must be computed by an order of magnitude or more, without loss of generality. In the verifier **Spin** we adopted an efficient variant of partial order reduction, that we named *static reduction* (**C**), in 1994 [HP94]. A small extension of this method, used to optimize the automata models, was added in 1999 (**D**).

A third method to reduce the memory requirements of a verification run is the use of alternative, highly compact, representations of the reachable state space. One such method consists of the computation of a minimized graph structure for recording all reachable states with as little redundancy as possible [HP99]. Such a storage method can trade increases in verification-time for sometimes dramatic additional reductions in the memory cost of verification. The benefits, however, are often problem dependent. For the application to the *TPC* model used in **Figure 3**, this method does not provide additional benefits.

Reducing the size of $A(\neg f)$

The size of G depends not just on S but also on the size of the automaton that is generated from a temporal logic formula f . The initial construction procedure that proved that for any formula f one can construct an equivalent automaton $A(f)$ generated prohibitively large automata. In 1995, Doron Peled, together with Rob Gerth, Pierre Wolper, and Moshe Vardi, developed a new algorithm that could produce much smaller automata [GPVW95]. This algorithm was included in **Spin** that same year. More recently, Kousha Etessami, also from Bell Labs showed that the automata could be reduced still further in size, often by a substantial margin [E00]. All improvements together translate into significant reductions in the complexity of verification, bringing what were once inconceivably large problems eligible targets for formal verification.

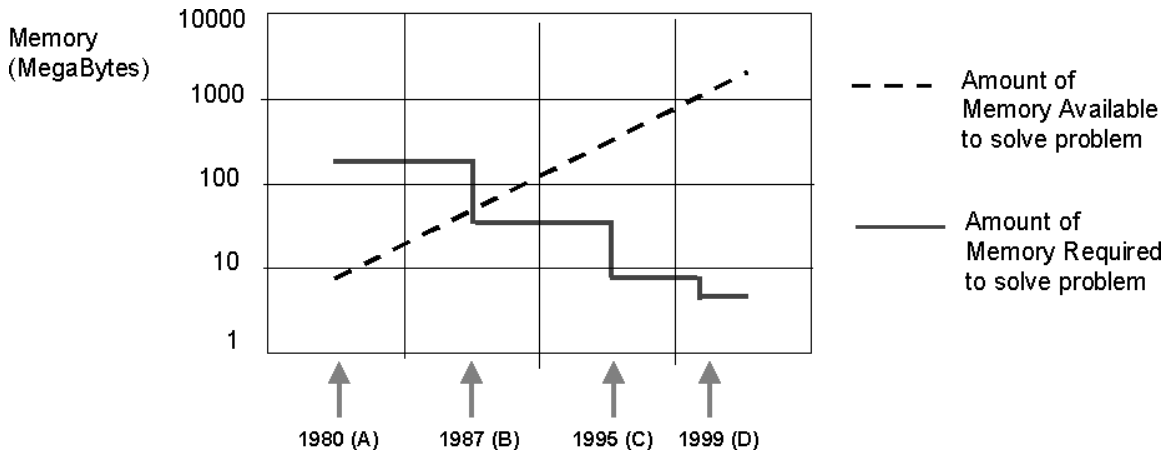


Figure 3. Performance improvements in the verification tool for distributed systems software **Spin**. Plotted is the minimum amount of memory required to verify the correctness of a sample model of a telephone, and the amount of memory available on a typical computer. The first attempt to verify the model was in 1980 (A). The effect of the supertrace algorithm is shown at (B), the effect of partial order reduction at (C), and, finally, the effect of additional model optimization techniques at (D).

Deriving Automata for Program Sources

Up to this point we have discussed how we can efficiently compute the intersection product of a system, specified as a collection of automata, and a property, expressed as a logic formula. We mentioned algorithms that can be used for extracting automata from temporal logic formula, and for reducing the amount of work that needs to be done in the product computation. The issue that we have not yet discussed is how one would go about obtaining a specification of a substantial software product in the form of automata. That is: how do we obtain the components for S in automata form when the system is implemented in a mainstream programming language, such as C.

The full implementation of a software product generally contains a significant amount of detail that is outside the scope of the verification, either because it was checked before or because it is unrelated to the property being checked now. This means that an automata description can be much more compact than a full system implementation. The most commonly used method today to derive a high level description of a software product is to create one manually. The user of a verification system is then asked to construct a verification model by hand, focusing on the issues that are central to the properties to be checked. There are two basic problems with this method. First, just like the original software can contain defects, so can a manually constructed model. Second, the manual construction of a model for a substantial piece of software can be intellectually demanding and very time consuming. For safety critical systems this investment can often be justified, and not surprisingly the use of formal verification techniques has so far been mostly restricted to these types of applications.

For routine software development, the manual construction of an automata model from systems code is often too cumbersome. Several solutions to this problem have been tried over the years. In one of the first large-scale verification efforts at our company we defined a subset of the programming language that was used for switch development, in such a way that programs written in the resulting language corresponded in a natural way to the high level automata description required for verification [H94]. Although successful from a verifier's point of view, the restrictions to the programming language were not long lived. We also attempted to extract verifiable automata models from high-level system requirements, to secure correctness for the starting point of a system's development [HPR97]. Nonetheless, the correctness of the high-level system requirements cannot secure the correctness of a system's implementation.

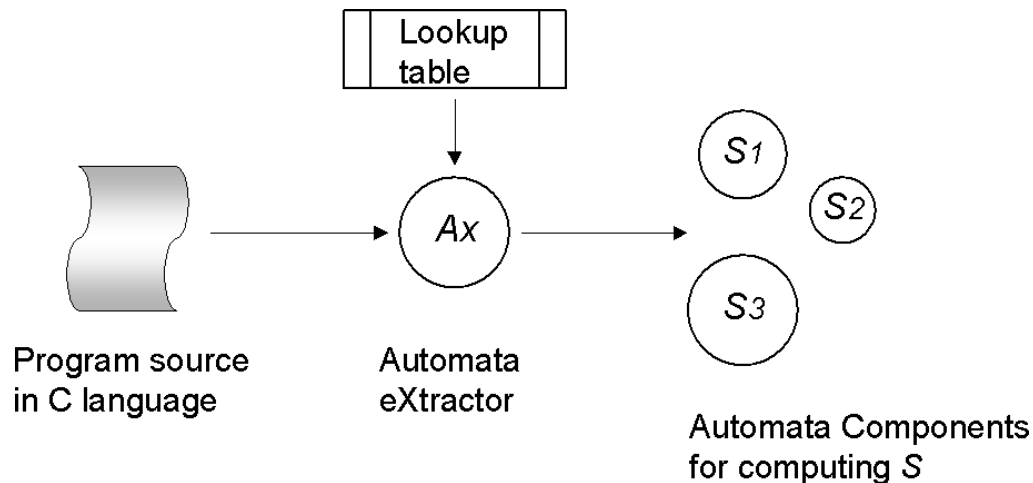


Figure 4. Automated extraction of verification models from program source, guided by a simple lookup table that contains user defined abstractions.

Mechanical Extraction of Automata

We recently developed a different method that makes it possible to extract automata models from implementation level code mechanically, as illustrated in **Figure 4**. The automaton extraction covers both control flow and data use for each asynchronous process in the system, and is guided by a user defined lookup table that restricts the model to those parts of the application that are of primary interest in the verification [HS99]. Model extraction with this method takes a fraction of a second and is guaranteed to match the product implementation. Verification results for an update of the program sources can thus be obtained within minutes, rather than in weeks or months with previous methods. A detailed description of the first application of this method to the verification of the call processing software for one of Lucent's new switching systems appears in the next issue of this journal [HS2000].

Verification and Testing

The capabilities of a verification tool like **Spin** are narrowly focused on issues of logical correctness in a distributed system. The verifier, for instance, is not designed to check purely computational aspects of a sequential program or to compute performance metrics. Its power is in checking that the interactions of the components in a distributed system satisfy functional properties. Within that domain of interest, a logic verifier like **Spin** can significantly outperform standard software testing methods, both in thoroughness and in speed. We have already mentioned the problem of reproducibility in a distributed system. We can use a verification system to formulate queries about the feasibility of system behavior that cannot be answered at all in online tests. More surprising is perhaps that a verification system can explore system executions *faster* than a system that is based on the real-time execution of the software. The reasons for this efficiency are in the nature of the search process and in the use of abstractions through the automata models.

To see how the search process helps to make verification more efficient than a straight set of system executions, consider the tree structure shown in **Figure 5**. It could represent a simple piece of software with fixed initial state, shown at the root of the tree. Each execution passes three successive conditional branches, for a total of eight possible system executions. Since each of these eight executions consists of three steps, an exhaustive test of this system, by executing each of the eight possible scenarios, would require a total of 24 steps.

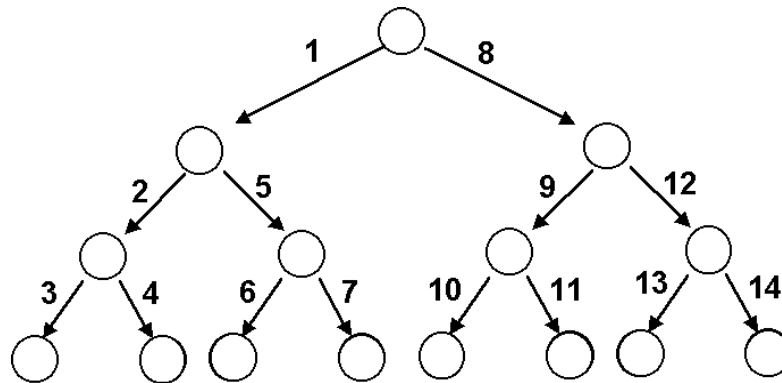


Figure 5.
Depth-First Search Order for Binary Expanding Tree

The execution steps in **Figure 5** are numbered in the order in which a depth-first search algorithm would explore them. Note that there are only 14 such steps. The exploration starts at the initial state of the automaton, here shown as the root of the tree, and recursively explores all successor states. At each newly visited state we select the left-most unexplored successor, move there, and repeat. When there are no more unexplored successor states, the search moves back one step to a previous state. If there are no more previous states (i.e., the search has returned to the root state, with all successor states explored), the search terminates and the full extent of the automaton is known. The gain in efficiency comes from the fact that if we want to check the path **1; 2; 4**, after having checked the path **1; 2; 3**, we do not need to repeat steps **1** and **2**. In general, if we are n steps deep in the tree, and a new execution path to be tested differs only in the last m steps from the previously tested paths, we can avoid having to repeat the first $n-m$ steps, and check only the steps that differ.

For the tree in **Figure 5**, a depth-first search explores 14 steps instead of 24. More generally, to check a binary expanding tree structure of n levels requires $(n \times 2^n)$ steps with a conventional test and $(2^{n+1} - 2)$ steps with a depth-first search. The gain in efficiency for $n > 1$ increases linearly with n . For an average length of an execution sequence of 200 steps, for instance, the gain would be a factor of 100.

The remaining issues that distinguish a conventional system test from a run with a logic verifier have to do with *controllability* and *observability*. In a concurrent system the executions of individual processes can be arbitrarily interleaved in time, unless otherwise constrained by process synchronization mechanisms. The eight full paths in **Figure 5**, for instance, could represent the possible interleavings of two executions steps in one process with a single execution step in a second. How precisely these steps would be interleaved in time during a real system execution is typically beyond the control of a tester. No matter how many times the test would be run, there is no guarantee that all interleavings are actually tested. By virtue of the extracted automata model a verification can indeed check each relevant interleaving. The partial order reduction strategy reduces the executions that are inspected to only those that differ in relevant ways, for instance because of access to shared data. The same argument goes for observability: not all aspects of the concurrent execution of asynchronous processes can be observable in a conventional system test. The details of process interleavings are difficult, if not impossible, to record, and, similarly, large classes of data may not be visible at all from the testing interface. Because the way in which automata are extracted from program sources, the observability of all events and conditions can be guaranteed in a logic verification system.

Conclusion

Software verification techniques have long remained relatively obscure, seemingly incomprehensible and largely irrelevant to industrial practice. The cumulative effect of both algorithmic improvements and improved tool design is slowly changing this. Today the use of software verification systems is already considered non-negotiable on safety critical systems. The verifier **Spin**, for instance, has been used to intercept software defects in the code for applications that range from railway signaling protocols to the control software for a new billion dollar flood control system that was recently built near Rotterdam in The Netherlands [H97]. At NASA, **Spin** was used to check the software for a dually redundant control system for one of the larger spacecraft [SECH98], and the control software for the Deep Space 1 mission [HLP98]. In all these applications significant software defects were found, that had escaped detection in thorough conventional testing.

Even in the unlikely case where no further algorithmic improvements are made in this area, the steady increase in available memory sizes and the general compute power of machines virtually guarantees a continued growth in the size of problems where software verification methods can be applied.

With automated model extraction techniques, much of the overhead traditionally associated with software verification techniques disappears. The ideal of creating a seemingly omniscient box that can magically point out the problem areas in a piece of concurrent code, virtually as quickly as the code is written, may not be very far away.

References

[B69] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson, A note on reliable full-duplex transmission over half-duplex lines, *Comm. of the ACM*, Vol. 12, No. 5, pp. 260-261, May 1969.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis, Memory efficient algorithms for the verification of temporal properties. *Formal Methods in Systems Design*, Vol. 1, pp. 275-288, 1992.

[E00] K. Etesami, *Optimizing B^{*}-chi Automata*, Bell Labs Technical Report, February 2000, submitted for publication.

[FMGS97] A.R. Flora-Holmquist, E. Morton, J.D. O'Grady, and M. Staskauskas, The Virtual Finite-State Machine Design and Implementation Paradigm, *Bell Labs Technical Journal*, pp. 96-113, Winter 1997.

[GPVW95] R. Gerth, D. Peled, M. Vardi, and P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, *Proc. Symp. on Protocol Specification, Testing, and Verification*, Warsaw, Poland 1995, Chapman and Hall, pp. 3-18.

[GHI98] P. Godefroid, R.S. Hammer, L. Jagadeesan, Systematic Software Testing using VeriSoft, *Bell Labs Technical Journal*, Volume 3, Number 2, April-June 1998.

[HLP98] K. Havelund, M. Lowry, and J. Penix, Formal Analysis of a Space Craft Controller using Spin, *Proc. 4th International Spin Workshop*, Paris, France, 1998.

[H87] G.J. Holzmann, On Limits and Possibilities of Automated Protocol Analysis, *Proc. 6th Int. Conf on Protocol Specification Testing and Verification*, IFIP, Zurich Switzerland, May 1987, 13.

[HP94] G.J. Holzmann, and D. Peled, An Improvement in Formal Verification, *Proc. Formal Description Technique*, Chapman Hall, Berne Switzerland, October 1994, pp. 197-211.

[H94] G.J. Holzmann, The Theory and Practice of a Formal Method: NewCoRe, *Proc. IFIP World Computer Congress*, Vol. I, pp. 35-44, North-Holland Publ., August 1994.

[H97] G. J. Holzmann, The Model Checker Spin, *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997.

- [HPR97] G.J. Holzmann D.A. Peled and M.H. Redberg, *Design Tools for Requirements Engineering*, Bell Labs Technical Journal, pp. 86-95, Winter 1997.
- [HS99] G. J. Holzmann, and M.H. Smith, Software model checking - Extracting verification models from source code, *Formal Methods for Protocol Engineering and Distributed Systems*, Oct. 1999, Kluwer Academic Publ., pp. 481-497.
- [HP99] G.J. Holzmann and A. Puri, A Minimized Automaton Representation of Reachable States, *Software Tools for Technology Transfer*, Springer Verlag, Vol. 2, 3, pp. 270-278, November 1999.
- [HS2000] G.J. Holzmann, and M.H. Smith, Automating Software Feature Verification, *Bell Labs Technical Journal*, Special Issue on Software Complexity, April 2000.
- [K94] R. Kurshan, *Computer-Aided Verification of Coordinating Processes*, Princeton Univ. Press, 1994.
- [L69] W.C. Lynch, Commentary on [B69], *Comm. of the ACM*, Vol. 12, No. 5, p. 261, and 265, May 1969.
- [P77] A. Pnueli, The temporal logic of programs, *Proc. 18th IEEE Symposium on Foundations of Computer Science*, 1977, Providence, R.I., pp. 46-57.
- [SECH98] F. Schneider, S.M. Easterbrook, J.R. Callahan, and G.J. Holzmann, Validating Requirements for Fault Tolerant Systems using Model Checking, April 1998, *Proc. International Conference on Requirements Engineering*, Colorado Springs, Co IEEE, pp. 4-14.
- [T72] R.E. Tarjan, Depth first search and linear graph algorithms, *SIAM J. Computing*, 1:2, pp. 146-160, 1972.
- [WVS83] P. Wolper, M.Y. Vardi, and A.P. Sistla, Reasoning about Infinite Computation Paths, *Proc. 24th IEEE Symposium on Foundations of Computer Science*, 1983, pp. 185-194, Tucson, Az.
- [VW86] M.Y. Vardi, and P. Wolper, An automata-theoretic approach to automatic program verification, *Proc. Symp. on Logic in Computer Science*, Cambridge, June 1986, pp. 322-331.