Software Model Checking

Gerard J. HOLZMANN
Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

In thes notes we will review the automata-theoretic verification method and propositional linear temporal logic, with specific emphasis on their potential application to distributed software verification.
An important issue in software verification is the establishment of a formal relation between the concrete, implementation-level, software application and the abstract, derived, automata-model that is the subject of the actual verification. In principle one can either attempt to derive an implementation from a verified abstract model, using refinement techniques, or one can attempt to derive a verification model from an implementation, using systematic abstraction techniques. The former method has long been advocated, but has not received much attention in industrial practice.
The latter method, deriving abstract models from concrete implementations guided by explicitly stated correctness requirements, has recently begun to show considerable promise. We will discuss it in detail.

1. Introduction

Programming is a human activity. Because even the most conscientious human can occasionally make mistakes, a professional software design process will normally include a careful system of checks and balances that aims to intercept as many of the mistakes as possible, before a product ships to customers. It is our premise that the fraction of mistakes intercepted can be increased, specifically for distributed systems designs, if we complement traditional testing techniques with software model checking techniques.

Not all mistakes are equally easy to detect. Errors of syntax are easier to find than semantic errors, and errors in sequentially executing, deterministic programs are easier to find than errors in multi-threaded, non-deterministic systems. We focus here on the problem of detecting errors in distributed systems code: network applications, data communications protocols, multi-threaded code, client-server applications, and the like. We are particularly interested in algorithmic techniques that can be harnessed into tools, and that can be integrated seamlessly into the software design cycle.

The goal of this introduction is to give a bird's eye view of the field and place the main issues in software model checking in context. We provide a brief introduction to the automata-theoretic checking process, discuss the use of logic for the specification of program properties. In the remainder of the notes we will also discuss formal program abstraction techniques, and a methodology for extracting verification models directly from program source code.

Feasibility

First a word about the relevance of software model checking techniques in industrial practice. Formally, the problem we are trying to solve can be shown to be PSPACE hard, e.g., [BZ83],[CM81]. In practical terms this means that there is a serious problem in handling large problem sizes. There will always be such problems, no matter how powerful machines become, so there is a need for algorithms that can scale graciously from exact to approximate solutions for growing problem sizes. As we shall see, such algorithms have been developed.

Can formal verification techniques handle the type of problem sizes that occur in practice today? The perception of most practitioners is that formal verification techniques are perhaps applicable to small examples, but not to any problem of real significance. This perception was formed and validated when the field was in its infancy, in the mid seventies, but is rarely re-examined. Since we have data on the relative performance of our verification tools for the last two decades, it can be interesting to see if the perception still holds true today.

In 1980 we used a basic reachability analyzer, called **Pan**, to verify properties of a model of an experimental telephone switch developed at Bell Labs [H81]. The switch, and the model, was called **Tpc**, short for *The phone company*. Through the use of **Pan** a number of design problems were uncovered in the **Tpc** software. A fully exhaustive verification was infeasible then; the complexity of the problem vastly exceeded the constraints of the best machines available at that time.

We can recognize two major trends that have affected the feasibility of the formal verification of problems such as these in the last two decades. The first is a series of algorithmic improvements that have helped to reduce the complexity of the model checking problem. The second trend is the cumulative effect of *Moore's curve*: a surprisingly reliable predictor of increases in memory sizes and CPU performance. On average, every 18 months the speed and memory size of the best available machine doubles. The effect of these two trends on the feasibility of solving the verification problem for **Tpc** is illustrated in Figure 1.

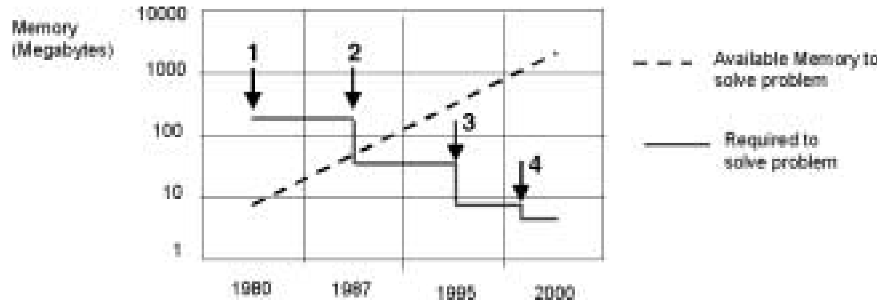


Fig. 1 — *Feasibility of Model Checking — Memory Requirements.*

Figure 1 shows the amount of memory that is available on the best available machine in each year between 1980 and 2000 (dotted), and the amount of memory that should minimally be available to solve the verification problem for our first **Tpc** model (solid). The marks indicate successive algorithmic improvements in the construction of the verifier, which itself slowly evolved from a basic reachability analyzer into a full logic model checking system named **Spin** [H97]. Mark 1 shows the memory requirements of our first algorithm from 1980. Mark 2 shows the memory requirements when the proof approximation algorithm that was introduced in 1987 [H87] is used at maximum precision (giving coverage that matches the one produced by a fully exhaustive reachability analysis). Mark 3 shows the drop in memory requirements when partial order reduction techniques are used [HP94], and mark 4 a smaller drop when some additional model reduction techniques are added [H99].

A very similar figure can be drawn for the runtime requirements of formal verification applications, as illustrated in Figure 2. A reduction in the runtime requirements for a full verification of the **Tpc** model can then be measured dropping from 7 days in 1980 to 7 seconds today, again by virtue of the combination of algorithmic improvements and the effect of Moore's curve. It should be added that the problem used to produce these data was not chosen to enhance any aspect of these trends. For more carefully selected problems, for instance, the improvements of individual algorithms can be made to look significantly better. Our purpose here is, however, not to showcase specific algorithms, but to illustrate the existence of a trend.

The nature of this trend is clear. Even if no further algorithmic improvements are made, software verification techniques will be able to handle increasingly complex problems by virtue of the exponential increase in the capabilities of available machines. This increase in power has meant that today we can perform

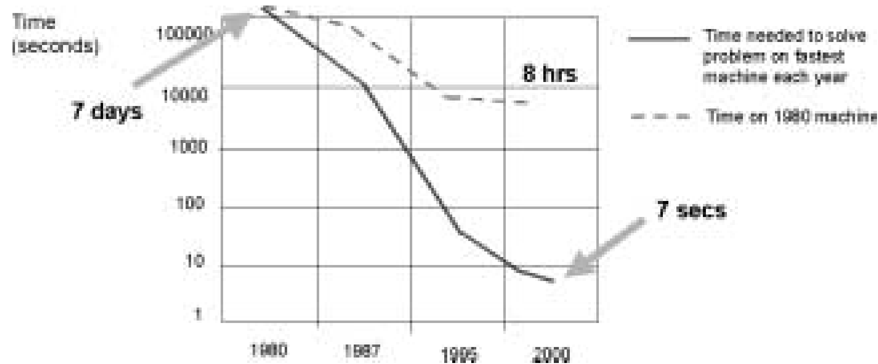


Fig. 2 — *Feasibility of Model Checking — Runtime Requirements.*

formal verification on fairly detailed models of telephone switching software [HS00]. What a continuation of the trend could mean for tomorrow can only be speculated. Suffice it to say that the prospects are good.

Modeling

For arbitrary programs with potentially unbounded capacity to store and retrieve information, no algorithmic techniques can exist for mechanically proving *all* properties of interest. In this form, the problem is undecidable [T36]. If we can put a finite bound on the possible memory use of a program, we obtain a system with a finite number of possible *states* (i.e., configurations of memory), that can in theory be enumerated. We can conceive of constructing the execution graph of such a program, to capture the successor/predecessor relation for all reachable memory configurations. This approach is not practical, though, considering the potential size of the graph and the likely computational expense of computing, storing, and analyzing it. For distributed systems the problems is still more severe. We may now have to deal with all possible combinations of the memory configurations of all concurrently executing processes. At this level of detail, the solution of the problem remains well beyond reach.

For many properties of interest, though, the fully detailed representation of an execution graph contains far more information than is needed for verification. In many cases, even a coarse abstract representation of the graph suffices. This abstract representation can be obtained by removing unwanted detail from the system description (i.e., the program) in such a way that properties of interest are preserved. The abstracted system description can be used to generate a smaller abstract execution graph, which can effectively be used in a verification process. The abstract system description is called a *model* of the original system.

The purpose of the construction of a model is to facilitate analysis: by using abstraction we can trade implementation detail for analytical power. The model could be created as a mathematical description, as a set of axioms, rules of inference, and theorems to be proven. In that case, both model and proof are most likely constructed manually, perhaps with some assistance from mechanical or human proof checkers. In some cases the model could also conceivably be created as a physical structure: a prototype device of which the basic properties may be verified by measurement.

Automata

We will focus on models that are expressed as *automata*. The automata models can in some cases be extracted from program source and analyzed mechanically. The potential automation of the verification process gives this approach an advantage over manual proof methods, although it is understood that automation may also bring limitations to the potential scope of a verification. Model construction and model extraction are based on systematic abstraction, such as slicing [T95], data hiding, and mapping [CLG94],[CD00],[V00]. We will discuss abstraction and automated model extraction techniques in part IV of these notes.

Logic

We have so far suggested that we may be able to obtain automata models from program sources, and that these models may suffice for the analysis of properties. We have not yet discussed how these properties can be expressed in such a way that automated analysis becomes possible.

Propositional linear temporal logic (LTL) allows us to make very concise statements about required causal relations between the events in a distributed system [P77],[E90]. Each LTL formulae, furthermore, can be converted mechanically [GPVW95] into an ω-automaton [T90] that can be used in the verification process. An automata-theoretic verification method [VW86] proceeds follows.

1. The property to be verified is expressed as a formula f in LTL, and then *negated* to $\neg f$. The negation reverses the meaning of the formula to capture all system behavior that deviates from the original requirement. The negated LTL formula is converted into an ω-automaton A, using the procedure outlined in [GPVW95],[EH00]. The negated property automaton is designed to accept all system behavior that satisfies the negated formula, and that therefore violates the original requirement.
2. The property is used to define an abstraction which guides the definition of an automata model for the system to be verified. The resulting system model S captures all possible system behavior at the required level of abstraction.
3. A model checker, such as **Spin** [H91], can now be used to compute the language intersection G of property automaton A and system automaton S, as illustrated in Figure 3. This language intersection contains all feasible violations of the original LTL formula f. If it is empty, no violations of the property are possible.
4. An error sequence (any violation uncovered in the last step) is interpreted at the source level of the original program (i.e., lifted) and reported to the user for action.

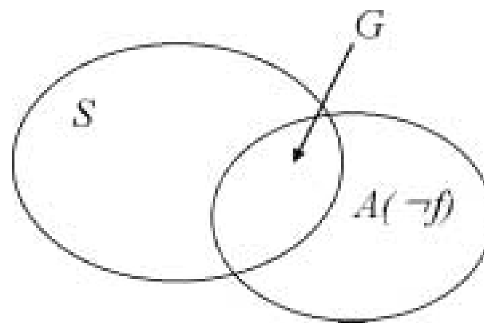


Fig. 3 — *Intersection G of System S and Automaton A Derived from LTL Formula ¬f.*

There are many issues that we have silently skipped over, but that need carefully consideration before this method can be used.

- Distributed systems often have dynamically changing numbers of active processes. In general there will be one separate automaton model for each asynchronously executing process in the system.
- The verification framework should apply both to finite, terminating, system executions, and to potentially infinite executions (ω-runs) [VW86],[T90].
- Optimization and reduction techniques must be considered to help reduce the amount of work required for the computation of language intersections [P96],[EH00]. Despite all that, the computational complexity of verification can still exceed the bounds of available resources. *Best-effort* relief strategies should be available for these cases.
- The validity of an abstraction cannot be taken for granted. An incorrect use of abstraction may produce false error reports or cause valid error reports to be missed.
- And finally, we should take into account that a system can only be verified subject to a reasonable set

of assumptions about the *environment* in which it is used. Just like the formulation of logic properties, it can be hard to derive such assumptions automatically. The validity of a verification result will always be conditional on the accuracy of these formalized assumptions. The assumptions should therefore be conservative.

Overview

In the remaining sections of these notes the details of the software verification method sketched above will be filled in. In Section 2 we begin by reviewing the automata theoretic verification method, the definition of ω-automata and ω-acceptance. We discuss the formal relation between propositional linear temporal logic and ω-automata, and consider the basic procedure for on-the-fly verification used in an existing LTL model checker **Spin**. In Section 3 we look at optimization and reduction strategies, including model reduction, partial order reduction, and proof approximation methods. In Section 4 we discuss model extraction, and systematic abstraction techniques. In Section 5 we reflect briefly on our findings.

2. Automata

We will model the actions of processes in a distributed system in terms of *states* and *transitions* (i.e., state *transformers*). These notions are captured in the definition of a finite automaton. Automata models are intuitive and have been used frequently for the description of distributed systems, also by practitioners. In particular there is a long history of their use for the definition of data communication protocols. The well-known definition of the 'alternating bit protocol' from 1969, for instance, was based on an automaton description [BSW69].

2.1. Finite automata

We begin with a standard definition of a finite automaton, defined over finite executions. We then generalize the definition to capture also infinite executions.

> A *finite automaton* is a tuple {S, s_0, L, F, T}, with S a finite set of 'states,' $s_0 \in S$, is a predefined 'initial state,' L is a finite set of labels or 'symbols,' F $\subseteq S$, is a set of 'final' states, and T $\subseteq S \times L \times S$, is the 'transition relation.'

The structure of a finite automaton can be represented by a graph, as illustrated in Figure 4, Vertices represent states, edges represent transitions, and labels appear as annotations on the edges. A path through this graph can then be interpreted as an execution, called a 'run,' of the automaton (we will define it more precisely below). A run is said to be *accepted* by the automaton if it starts in the initial state and ends in one of the final states in set F. Of course, this particular notion of acceptance applies only to finite runs.

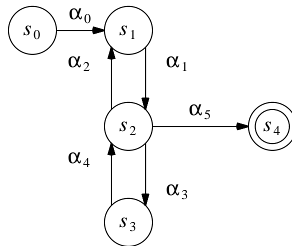


Fig. 4 — *The Structure of a Finite Automaton.*

The labels from set L can be treated as abstract representations of arbitrary program 'actions.' This would include access to data objects, to modify or to test their value (their 'state'). Any *run* of the automaton then defines a sequence of labels. For a fixed *interpretation* of the labels in a given *context* we can restrict the notion of acceptance to only those runs that would be feasible under the given interpretation. The action that is represented by a label, for instance, may only be feasible (executable) under precisely stated conditions. A strict definition of the interpretation of labels in a given context will not be needed for these notes, so we will not pursue it here.

For the example automaton in Figure 4 we have S = { s_0, s_1, s_2, s_3, s_4 }, L = { $\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5$ }, F = { s_4 }, and T = { $(s_0, \alpha_0, s_1), (s_1, \alpha_1, s_2), (s_2, \alpha_2, s_1), (s_2, \alpha_3, s_3), (s_3, \alpha_4, s_2), (s_2, \alpha_5, s_4),$

}. This automaton could be used to model the life of a user process in time-sharing system, as controlled by a process scheduler. State s_0 then represents the 'Initial' state where the process is being instantiated, s_1 is the 'Ready' state, s_2 is the 'Running' state, s_3 is the 'Suspended' state, e.g., where the process is blocked waiting for a system call to complete, and s_4 is the 'Final' state, reached if and when the process terminates. An interpretation of the symbols in set L for this system can be: α_0 is the scheduler's 'Start' action, α_1 is 'Run,' α_2 is 'Suspend,' α_3 is 'Block,' α_4 is 'Unblock,' and α_5 is 'Stop.' An acceptable finite run of this system is the state sequence $\{s_0, s_1, s_2, s_4\}$, which corresponds to the sequence of scheduler actions Start, Run, and Stop.

2.2. Runs

A more precise definition of the *run* of an automaton can be given as follows.

> $\sigma = (s_0, s_1, s_2, \cdots, s_k)$ is a *run* of finite state automaton {S, s_0, L, T, F}, if and only if (iff) $(\forall i, 0 \le i < k : \exists \alpha, \alpha \in L \wedge (s_i, \alpha, s_{i+1}) \in T)$.

We can also define a run as an ordered set of labels from L instead of an ordered set of states from S. If the automaton is non-deterministic, which is generally the case in software model checking applications, the two definitions are of course not equivalent.

Set L defines the 'alphabet' of label symbols. Each run of the automaton defines one or more *words* over that alphabet. (Note that adjacent states in the run may be connected by multiple symbols.) In classic finite state automata theory, a finite run is said to be *accepted* iff it terminates at a state within set F. The set of words that correspond to accepted runs is referred to as the *language* accepted by the automaton.

> A *finite* run $\sigma = (s_0, s_1, s_2, \cdots, s_k)$ of finite state automaton {S, s_0, L, T, F} is *accepted* iff $s_k \in F$.

We would like to be reason equally about terminating and non-terminating systems, though, and therefore we need a broader definition of acceptance.

2.3. Büchi-acceptance

There are several ways to extend the notion of acceptance to infinite runs [T90]. We will use a simple method called Büchi-acceptance.

> An *infinite* run σ of finite state automaton {S, s_0, L, T, F} is *accepted* iff it at least one state from set *F* appears infinitely often in σ.

For the automaton in Figure 4, for instance, we could define the Running state s_2 as a Büchi-acceptance state. In this case all infinite runs would then necessarily be accepting, since there exists no strongly connected component in the graph of Figure 4 that excludes s_2.

A simple extension of finite runs will also allow us to interpret finite runs as special cases of infinite runs, for the purpose of deciding acceptance.

> The *stutter extension* of finite run $\sigma = (s_0, s_1, s_2, \cdots, s_k)$ of finite state automaton {S, s_0, L, T, F} is the concatenation of σ with s_k^*: an infinite repetition of final state s_k.

The infinite repetition of the final state of a finite run corresponds to the addition of a dummy self-loop transition (s_k, ε, s_k) to set T, minimally for each state s_k in set F, where ε is a predefined label representing a nil action.

A slightly more general definition of Büchi-acceptance is known as *Generalized Büchi-acceptance*. In this case, we allow for more than one set of final states F, and require that at least one state from each of these final sets appears infinitely often in a run. An unfolding method, known as Choueka's flag-construction [C74], can be used to translate a Generalized Büchi-automaton into a standard one. We will see an example of a generalized Büchi-automaton below

A number of interesting properties of Büchi-automata are decidable, specifically:

- *non-emptiness*: deciding whether a given Büchi automaton accepts any runs at all, and
- *intersection*: given two Büchi automata, constructing a new automaton that accepts precisely those runs that are accepted by both of the given automata.

The model checking procedure relies on both of these methods.

2.4. Products of automata

The joint execution of two finite automata can be defined as product of automata. There are several ways to define an automata product, reflecting differences in the assumptions about the semantics of joint behavior in a distributed system.

> The *automata product* of the finite automata $\{S^A, s_0^A, L^A, F^A, T^A\}$ and $\{S^B, s_0^B, L^B, F^B, T^B\}$ is the finite automaton {S, s_0, L, F, T}, such that S = $S^A \times S^B$, $s_0 = (s_0^A, s_0^B)$, L = $(L^A \times L^B) \cup \varepsilon$, F = { (s,t) | s $\in F^A \wedge t \in F^B$ }, and T $\subseteq S \times L \times S$.

The interesting part is to provide a precise definition of the transition relation T. We can, for instance, define it as follows:

$$T = \{ ((s,t), (\alpha,\beta), (v,w)) \mid ((s,\alpha,v) \in T^A \vee (\alpha \equiv \varepsilon \wedge s \equiv v)) \wedge ((t,\beta,w) \in T^B \vee (\beta \equiv \varepsilon \wedge t \equiv w)) \}.$$

This definition allows for both joint and independent transitions, where one automaton changes state while the other performs a self-loop on ε. A 'joint' transition can be used to model synchronization conveniently, e.g. rendezvous operations. In most cases of interest we can also remove the joint operations from the definition without loss of generality. The result of a joint action can usually also be modeled with a sufficiently finely grained interleaving of atomic actions.

The product AxB differs from the product BxA only in the naming conventions for states and transitions: the graphs corresponding to these two products are isomorphic.

2.5. Logic and automata

The next important step we have to make is the establishment of a direct link between a general formalism for expressing logical requirements on a distributed system and automata representations. That link is provided by propositional linear temporal logic.

Linear temporal logic (LTL) was proposed in the late seventies by Amir Pnueli as a formalism for reasoning about concurrent systems [P77]. The main notions used in the definition of temporal logic were derived from earlier work on *Tense Logics* for tightening arguments relating to the passage of time. Curiously, this work did not originate in computer science but in philosophy [P57],[P67],[RU71].

Propositional linear temporal logic can be used to formally state properties of system executions with the help of boolean propositions, the classic boolean relational operators, and a small number of new temporal operators that we discuss next. The truth of a temporal formula is always defined over infinite runs.

If a temporal formula *f* is valid (holds) for ω-run σ, we write:

$$\sigma \models f.$$

We will write σ[i] to denote the suffix of a run σ starting at the i-th element, with σ[1] ≡ σ.

The first temporal operator we will discuss is the binary operator *until*, first introduced in [K68], and represented by the symbol U. There are two variations of this operator, a *weak* version and a *strong* version.

Weak Until (U): $\forall i, (\ \sigma[i] \models (p \mathrm{U} q) \Leftrightarrow \sigma[i] \models q \vee (\sigma[i] \models p \wedge \sigma[i+1] \models (p \mathrm{U} q)))$.

This definition does not require that sub-formula q will eventually hold. The strong until operator, written ***U***, adds that requirement:

Strong Until (*U*): $\forall i, (\ \sigma[i] \models (p \boldsymbol{U} q) \Leftrightarrow \sigma[i] \models (p \mathrm{U} q) \wedge \exists j, j \geq i, \sigma[j] \models q)$.

There are two special cases of these definitions that are important enough that two separate operators are defined for them. The first the case where the second operand of the weak until operator is *false*, which leads to the definition of the unary operator □, pronounced 'box' or *Always*.

Always (□): $\forall i, (\ \sigma[i] \models \Box p \Leftrightarrow \sigma[i] \models (p \mathrm{U} \mathit{false}))$.

This operator captures the important notion of safety or *invariance*. The second special case is when he first operand of the strong until operator is *true*, which leads to the definition of the unary operator ◊, pronounced 'diamond' or *Eventually*.

Eventually (◊): $\forall i, (\ \sigma[i] \models \Diamond q \Leftrightarrow \sigma[i] \models (\mathit{true} \boldsymbol{U} q))$.

This operator captures the important notion of inevitability or *liveness*.

There are many standard types of correctness requirements that can be expressed with the temporal operators we have defined. Two important types are defined below: recurrence and stability. A *recurrence property* is any temporal formula that can be written in the form $\Box\Diamond p$; the dual property, written $\Diamond\Box p$, is called a *stability property*. The recurrence property $\Box\Diamond p$ states that it is always true that p will be satisfied at some future point in the run. The stability property $\Diamond\Box p$ states that there is point in the run from where p is invariantly satisfied.

There are other interesting types of duality. For instance, it is not hard to prove that in any context: $\neg\ \Box p \Leftrightarrow \Diamond\neg p$ and $\neg\Diamond p \Leftrightarrow \Box\ \neg p$. Some commonly used equivalence rules are listed below, cf. [MP91].

[1]	$\neg\ \Box p$	$\Leftrightarrow$	$\Diamond\ \neg p$
[2]	$\neg\Diamond p$	$\Leftrightarrow$	$\Box\ \neg p$
[3]	$\neg(p\ \boldsymbol{U}\ q)$	$\Leftrightarrow$	$(\neg q)\mathrm{U}(\neg p \wedge \neg q)$
[4]	$\neg(p\ \mathrm{U}\ q)$	$\Leftrightarrow$	$(\neg q)\boldsymbol{U}(\neg p \wedge \neg q)$
[5]	$\Box(p \wedge q)$	$\Leftrightarrow$	$\Box p \wedge \Box q$
[6]	$\Diamond(p \vee q)$	$\Leftrightarrow$	$\Diamond p \vee \Diamond q$
[7]	$p\ \mathrm{U}\ (q \vee r)$	$\Leftrightarrow$	$(p\ \mathrm{U}\ q) \vee (p\ \mathrm{U}\ r)$
[8]	$(p \wedge q)\ \mathrm{U}\ r$	$\Leftrightarrow$	$(p\ \mathrm{U}\ r) \wedge (q\ \mathrm{U}\ r)$
[9]	$p\ \boldsymbol{U}\ (q \vee r)$	$\Leftrightarrow$	$(p\ \boldsymbol{U}\ q) \vee (p\ \boldsymbol{U}\ r)$
[10]	$(p \wedge q)\ \boldsymbol{U}\ r$	$\Leftrightarrow$	$(p\ \boldsymbol{U}\ r) \wedge (q\ \boldsymbol{U}\ r)$
[11]	$\Box\Diamond(p \vee q)$	$\Leftrightarrow$	$\Box\Diamond p \vee \Box\Diamond q$
[12]	$\Diamond\Box(p \wedge q)$	$\Leftrightarrow$	$\Diamond\Box p \wedge \Diamond\Box q$

2.6. Implication and causality

It can sometimes be hard to interpret the meaning of more complex temporal logic formulae. A common case of confusion is to mistake logical implication for temporal causality. To state, for instance, that the occurrence of event p (say, a request) will inevitably lead to the occurrence of event q (the corresponding response), one would be tempted to write

$$p \rightarrow q$$

which is incorrect. By the definition of logical implication, this formula would state merely that in the initial program state we must have $(\neg\ p \vee q)$. There is no statement on a required temporal relation between p and q. Slightly better would be to write

$$p \rightarrow \Diamond\ q$$

but also this is most likely not what was intended, since it still requires that the initial condition holds precisely in the initial state. If p does not hold initially, no check at all is implied here for any future occurrences of p. Somewhat better again is therefore to write:

$$\Box\,(p \rightarrow \Diamond\ q)$$

but even this is most likely not what the user meant. Clearly if event p never occurs, then the condition will be vacuously true. If the user went to the trouble of writing down the more complicated form of the expression that includes q, there is probably an expectation that the trivial case

$$\Box\,\neg\ p$$

does not apply (i.e., is *not* satisfied). In this case it is wise to prove the absence of the trivial case explicitly with a separate check. Note carefully that if it is acceptable that some runs contain p and some do not, it will *not* suffice to prove that

$$\neg\ \Box\,\neg\ p \equiv \Diamond\ p$$

because this states that p must eventually occur at least once in *all* runs. If p occurs in *some* but not *all* runs, neither $\Diamond$ p, nor its negation $\Box\,\neg$ p, will hold.

2.7. The next operator

There is one other standard temporal operator that we will exclude from our toolkit, even though it cannot be defined in terms of the other operators. It is usually defined as follows.

Next (X): $\forall i$, (σ[i] ⊨ **X**q ⇔ σ[i+1] ⊨ q).

We have two reasons to reject the use of the next operator.

- The precise meaning of the operator is unclear in the context of concurrent systems, cf. [L83]. A run of a concurrent system is typically given as an interleaving sequence of the runs of a number of participating processes. Whereas a 'step' in the run of a sequentially executing single process reflects the progression of a computation in a meaningful way, the same is not necessarily true for a 'step' in the run of a concurrent system. There is, for instance, no simple way to relate these steps to a global notion of time. Consider, for instance, the effect of network latency, message overtaking, message duplication, etc.
- We can define a powerful optimization of the verification process [HP94] for the stutter-invariant subset of full LTL. Any LTL formula that can be written without the use of the next operator is guaranteed to be stutter-invariant and vice-versa any stutter-invariant propositional LTL property can be written without the next operator [P97].

2.8. Verification

The most significant benefit of the use of LTL in a model checking procedure is that for every LTL formula one can construct a Büchi-automaton *A* that accepts precisely the runs that satisfy the formula [VW86,GPVW95,V96,DGV99,EH00,SB00]. By constructing this *property automaton* for a given LTL formula, we can now find all runs that *satisfy* the formula in a system *S* by intersecting the property automaton *A* with the system automaton *S*. Better still, by negating the property before the property automaton is constructed, we can similarly find all runs that *violate* the original property. Computing the intersection amounts to computing an automata product, a well understood procedure. This, in a nutshell, then is the automata-theoretic verification method. In the following we will first consider the relation between LTL formula and Büchi-automata a little more closely, and then look at the computation of the intersection product.

2.9. Construction

The essence of the procedure that can be used to construct a Büchi-automaton from an LTL formula is as follows. First we need to define the closure of a temporal formula.

The *closure* of temporal formula *f*, *Cl(f)*, is the set of all sub-formulae of *f* and their negations.

For example, if *p* is a boolean propositional symbol, then $Cl(\Diamond\Box p) \equiv \{ \Diamond\Box p, \neg\Diamond\Box p, \Box p, \neg\Box p, p, \neg p \}$.

Let *Prop(f)* be the set of all boolean propositional symbols in *f*. For the example, $Prop(\Diamond\Box p) \equiv \{ p \}$. Each state in the automaton that is constructed contains a list of those subformulae from *Cl(f)* that are satisfied in that state. We refer to that list for state *s* as *Ann(s)*.

Given a temporal formula *f*, the corresponding *Generalized* Büchi-automaton is { S , s_0, L, T, F }, where S = $2^{Cl(f)}$, s_0 is the state in S for which $Ann(s_0) \equiv f$, L = $2^{Prop(f)}$, and F = { F_1, ... F_n }. That is, each element of the L corresponds to a unique truth assignment to the propositional symbols in *Prop(f)*.

Transition relation T is now defined as follows:

$(s,\gamma,s') \in$ T if and only if the truth assignment $\gamma \in$ L satisfies all non-temporal formulae in *Ann(s)*, and
$((p\ \boldsymbol{U}\ q) \in Ann(s) \rightarrow (\ q \in Ann(s) \vee (\ p \in Ann(s) \wedge (p\ \boldsymbol{U}\ q) \in s'\ )))$

There is one sub-set in F for each sub-formula in *Cl(f)* that contains a strong until operator. Assume there are *n* such sub-formulae. For the i-th sub-formula $(p\ \boldsymbol{U}\ q)$ in *Cl(f)*, $(1 \le i \le n)$, we have:

$$F_i = \{\ s : (p\ \boldsymbol{U}\ q) \notin Ann(s) \vee q \in Ann(s)\ \}$$

Only the strong until sub-formulae contribute to the Büchi-acceptance conditions. We can use Choueka's

flag construction method [C74] to convert the generalized Büchi-automaton into a standard one. If we consider the automaton as a graph this can be done by making n copies of the graph, numbered $1..n$. We then change the edges exiting from all states in F_i in the i-th copy of the graph, $(1 \le i < n)$ to point to their successor in the (i+1)-th copy of the graph, and the edges exiting from the states in F_n in the n-th copy of the graph are redirected to their successors in the 1st copy. Finally, only the states in set F_n in the n-th copy of the graph are preserved as Büchi-accepting states. All other states are made non-accepting. To be accepted, any infinite run in the final automaton must now necessarily include at least one state from each set in F, which secures that all strong untils will be satisfied in the run.

The above construction can be improved significantly with an on-the-fly construction that avoids creating redundant states. The basic algorithm for doing so was introduced in [GPVW95]. Further improvements can be found in [DGV99],[SB00],[EH00]. As an example, the automaton that corresponds to the formula ◊□p, as computed by the **Spin** model checker [H97], is shown in Figure 5.

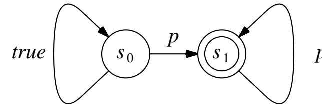


Fig. 5 — *Non-Deterministic Büchi Automaton for LTL formula ◊□p.*

2.10. Model checking

Assume we are given a finite automaton representing a system S = { S^S, s_0^S, L^S, F^S, T^S } and a property f of S expressed in LTL. S is generally defined as the product of smaller component automata representing concurrent processes.

Property f and its negation $\neg f$, contains propositional symbols as operands, where the truth-value of each proposition is defined as a boolean expression over the states of S. (That is, for any given state of S, any given propositional symbol evaluates to either true or false.)

If we convert $\neg f$ into Büchi-automaton A = { S^A, s_0^A, L^A, F^A, T^A }, the labels in L^A will always be boolean combinations of propositional symbols, without temporal operators. For each state in S we can evaluate the expression for each such label, and determine its truth value. Let *Eval*(α,*s*) be the result of evaluating label $\alpha \in L^A$ at state $s \in S^S$.

To compute the intersection of the automata S and A we can compute their automata product with a restricted transition relation defined as follows:

$$T = \{ ((s,t),(\alpha,\beta),(v,w)) \mid ((s,\alpha,v) \in T^S \wedge (t,\beta,w) \in T^{T)} \wedge Eval(\beta,s) \equiv true \}.$$

All states in S^S are defined to be accepting by setting $F^S = S^S$. Any infinite run accepted by this intersection product of S and A now corresponds to a run of S for which the $\neg f$ is satisfied and therefore the original property f is violated.

2.11. Complexity

The computational complexity of the model checking procedure is clearly linear in the size of the product of S and A. This statement, however, hides two basic facts:

1. The size of A can be *exponential* in the size of the LTL formula f, measured as the number of temporal operators used.
2. The size of S can be *exponential* in the number of component automata that is used to compute it.

Although both observations seem equally pernicious, the first is far less so than the second. Temporal formulae of practical value contain only few temporal operators, and the property automata generated from these formula typically contain very few states, typically between two and five (cf. Figure 5). The precise meaning of formulae that generate larger automata can be hard to determine, and they are therefore of limited value. The number of components in a large concurrent system, however, can only be restricted with abstraction techniques, which in themselves need justification before they can be relied upon in verification. The typical size of a system automaton S can easily exceed millions of states, and is the true source of complexity in model checking applications. Optimization techniques can be used to restrict the size of both A

and S.

2.12. On-the-fly verification

The model checking procedure we have outlined lends itself well to an implementation that allows on-the-fly verification. This means that we can instrument the verification system in such a way that it can detect the presence of a violation of the property before the intersection product of system and property automata is fully computed, and in many cases even without construction the full system S. First observe that violating runs are always infinite runs that contain at least one accepting state infinitely often. This means that there must exist at least one accepting state in the intersection product of S and A that is both reachable from the initial state of the product and reachable from itself.

Proving that such an accepting state either exists or does not exist can be done with two basic depth-first search procedures: first to detect all accepting states that are reachable from the initial state, and second to identify the accepting states from this set that are reachable from themselves. The second part of the problem amounts to detecting cycles in a finite graph, and as such would be a good fit for Tarjan's algorithm for constructing the strongly connected components of a graph in linear time [T72]. In practice we can do slightly better.

2.13. Cycle detection

The problem is to efficiently detect the existence of a cycle through an accepting state in a finite graph. In the worst case the algorithm we use to solve this problem will visit every node in the graph, and therefore the complexity cannot be less than linear in the size of the graph. But if the construction of the strongly connected components can be avoided, this problem may be solved with lower overhead than Tarjan's algorithm.

Tarjan's algorithm stores the nodes of a graph in a single depth-first traversal. Each node is typically annotated with two integer numbers, a *lowlink* and a *depth-first* number, e.g. [AHU74]. This requires storing with each node 2xlog(R) additional bits of information, to represent the lowlink and the depth-first number of a node, if R is the number of nodes in the graph. In practice, with R unknown, one typically uses two 32-bit integers to store this information. We will explore an alternative method that allows us to solve the cycle detection problem while adding just two bits of information to each node.

We begin by discussing a simple algorithm for a restricted class of ω-properties, i.e., proving the absence or existence of non-progress cycles in a finite graph [H90],[H91]. The algorithm works by splitting the depth-first search into two phases with the help of a two-state demon automaton. Next we discuss a stronger two-phase search algorithm that can be used to prove the absence or existence of acceptance cycles, as required for LTL model checking [CVWY92],[HPY96].

2.14. Non-progress cycle detection

We begin by taking the product of system automaton S with the two-state automaton D illustrated in Figure 6. Automaton D can non-deterministically decide to move from its initial state s_0 into an alternate state s_1, where it will then stay forever. The label on this transition is ε, the nil-action. We assume that zero or more states in the automaton S have been identified as *progress* states. We will be interested in finding any infinite run that contains only finitely many such progress states. This corresponds to solving the model checking problem for LTL properties of the type $\Diamond\Box$ *np*, with *np* a predefined state property that is *true* if and only if the system is not in a progress state.

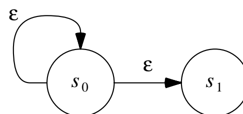


Figure 6 — *Two-State Non-deterministic Automaton for Detecting Non-Progress Cycles.*

We compute the asynchronous product of S and D, and perform a slightly modified depth-first search in the reachability graph for that product. Naturally, the product will be at most twice the size of S, containing one

copy of each state with D in state s_0, and possibly one more copy with D in s_1.

Each state s in the product is a tuple consisting of a state of D and a state of S. Let $Dm(s) \equiv true$ if D is in state s_0, and let $Np(s) \equiv true$ if s is not a progress state. The search starts from the initial state of the product of S and D, with D in state s_0. The non-progress cycle detection algorithm maintains a set *visited* of all states it has encountered in the search, and a search *stack* of states currently being explored.

```
dfs_A(s)
{
        add s to visited

        if Dm(s) or Np(s)
        {       push s onto stack
                for each successor s' of s
                {       if s' not in visited
                        {       dfs_A(s')
                        } else if s' in stack and ¬ Dm(s')
                        {       report non-progress cycle
                                stop
                }       }
                pop s from stack
        }
}
```

The algorithm ignores all successors of progress states as soon as D reaches state s_1. Every cycle that exists with D in state s_1 is therefore necessarily a non-progress cycle.

Property. If non-progress cycles exist, `dfs_A()` will report at least one of them.
Proof. Suppose there exists a reachable state that is part of a non-progress cycle, i.e., it can be reached from itself without passing through progress states. Consider the *first* such state that is entered into the second state space (upon the transition of D to its alternate state), and call it *r*.
State *r* is reachable from itself in the second state space and must find itself in the depth-first search below *r* unless that search truncates at a previously visited state outside the current search stack. Call that previously visited state *v*. We know that *r* is reachable from *v* (or else it would not block *r* from reaching itself) and also that *v* is reachable from *r*. This means that also *v* is reachable from itself in the second statespace via *r*. This, however, contradicts the assumption that *r* was the first such state entered into the second state space. This means that *r* either revisits itself or a successor of *r* revisits itself before that happens. In both cases the existence of a non-progress cycle is reported. □

Whenever a cycle is detected, the corresponding run can be reproduced from the contents of the stack: it will contain a finite prefix of non-repeated states, and a finite suffix, starting at the state within the stack that was revisited, with only non-progress states. This capability to produce exact counter-examples that demonstrate the violation of a property is critical in a model checking system.

To implement the algorithm it is not necessary to store two full copies of each reachable state. It suffices to store the states once with the addition of just two bits of memory [GH93]. The first of the two bits records if the state was encountered in the first statespace, and the second bit records if the state was encountered in the second statespace. Initially both bits are off. We can encounter only the bit combinations 01, 10, and 11, but not 00. (The state is neither present in the first nor in the second statespace for bit combination 00.) States may be either encountered first in the second statespace, and later in the first statespace, or vice versa. One bit, e.g. to record only the state of D, therefore would not suffice.

This non-progress cycle detection algorithm was first implemented in 1988 and later incorporated in **Spin** [H90],[H91]. An stronger version of this type of two-phase search algorithm was introduced in [CVWY92]. This algorithm is known as the *nested depth-first search*.

2.15. Nested depth-first search

This time the transitions of D are placed under the control of the search algorithm. The call dfs_B(s,d) performs a depth-first search from state s in S and state d in D. Let *Acc(s)* ≡ *true* if and only if state s is accepting. The search starts with the call dfs_B(s_0, s_0)

```
dfs_B(s, d)
{
        add s to visited

        push s onto stack
        for each successor s' of s
        {       if s' not in visited
                {       dfs_B(s', d)
                } else if s' ≡ seed and d≡s1
                {       report acceptance cycle
                        stop
        }       }
        if d≡s0 and Acc(s)
        {       // remember the root of the second search
                seed = s
                // perform second search in postorder
                // with demon moved to state s1
                dfs_B(s,s1)
        }
        pop s from stack
}
```

The search tries to locate at least one accepting state that is reachable from itself. Automaton D can move from its initial state on at accepting states in S and the move is explored only after all successors of the accepting state have been explored (i.e., in postorder). It is now no longer sufficient for the second search to find any state within the depth-first search stack, we must require that the seed state from which the second search was initiated itself is revisited. The proof of correctness for this version of the algorithm is as follows [CVWY92].

Property. If acceptance cycles exist, dfs_B() will report at least one of these.

Proof. Let r be the first accepting state reachable from itself for which the second search is initiated. State r cannot be reachable from any state that was previously entered into the second state space. Suppose there was such a state w. To be in the second state space w either is an accepting state, or it is reachable from an accepting state. Call that accepting state v. If r is reachable from w in the second state space it is also reachable from v. But, if r is reachable from v in the second state space, it is also reachable from v in the first state space. There are now two cases to consider. Either (a) r is reachable from v in the first state space without visiting states on the depth first search stack, or (b) it is reachable only by traversing at least one state x that is on the depth first search stack (cf. Figure 7). In case (a), r would have been entered into the second state space before v, due to the postorder discipline, contradicting the assumption that v is entered before r. In case (b), v is itself an accepting state reachable from itself, which contradicts the assumption that r is the first such state entered into the second state space.

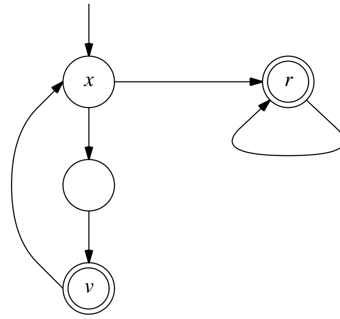


Fig. 7 — *States x, v, and r.*

State r is reachable from all states on the path from r back to itself, and therefore none of those states can already be in the second statespace when this search begins. The path therefore cannot be truncated and r is guaranteed to find itself in the successor tree. □

Like `dfs_A`, this algorithm requires no more than two bits to be added to every reachable state in S, so the overhead remains minimal. A significant advantage of this method of model checking is also that the entire verification procedure can be performed *on-the-fly*: errors are detected during the exploration of the search space, and the search process can be cut short as soon as the first error is found. It is not necessary to first construct an annotated search space before the analysis itself can begin.

We can check non-progress properties with algorithm `dfs_B` by defining the temporal logic formula $\Diamond\Box\, np$, with np equal to *true* if and only if the system is in a non-progress state. The automaton that corresponds to this formula is a two-state automaton shown in Figure 8.

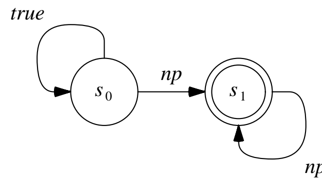


Fig. 8 — *Two-State automaton for* $\Diamond\Box\, np$.

To perform model checking we can now take the intersection product of the automaton in Figure 8 with system S, and use algorithm `dfs_B` to detect the accepting runs. We thus potentially incur two doublings of the search space: one due to the nested search inherent in `dfs_B` and one due to the product with the property automaton from Figure 8. The earlier algorithm `dfs_A` solves this specific problem more efficiently by incurring only the doubling from D. The advantage of `dfs_B` is nonetheless that it can handle any type of LTL property, not just non-progress properties.

3. Reduction

A simple and useful type of data object in distributed systems is the fifo (first-in first-out) message queue. The queue can be used to store data, or 'messages,' exchanged between processes in the order in which they were received. In a typical distributed system there is at least one message queue per asynchronous process, with a capacity to store hundreds of messages, each selected from a fairly broad set of possible messages. How feasible would it be to analyze such systems directly, without any form of reduction or abstraction?

Suppose we have q such queues, each with enough capacity to hold up to s messages, with m distinct types of messages. In how many 'states' can this set of data objects be? Each queue can hold between zero and s messages, with each message being a choice of one out of m, therefore, the number of states R_Q is

$$R_Q = \left[\sum_{i=0}^{s} m^i \right]^q .$$

Figure 9 shows how the number of states varies for different choices of the parameters q, s, and m.

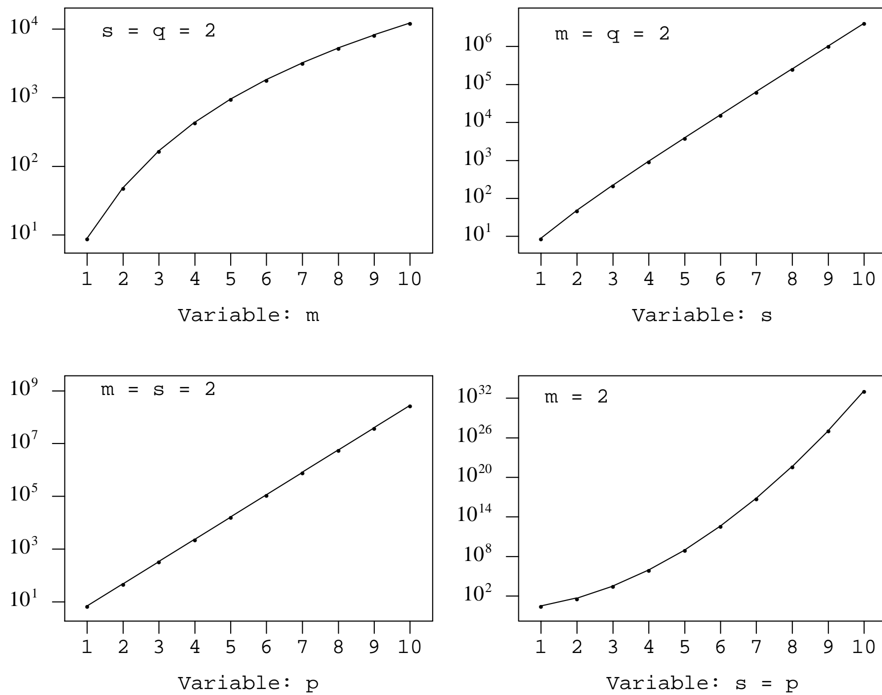


Fig. 9 — *Number of Possible States for q Message Buffers. With s Buffer Slots and m Message Types*

In the top-left graph of Figure 9, the parameters *s* and *q* are fixed to a value of 2, and the number of message types is varied from 1 to 10. There is a geometric increase in the number of states, but clearly not an exponential one. In the top-right graph, the parameters *m* and *q* are fixed to a value of 2, and the number of queue slots *s* is varied. This time there is an exponential increase in the number of states. Similarly, in the bottom-left graph, the parameters *m* and *s* are fixed, and the number of queues is varied. Again, we see an exponential increase in the number of states. Worse still, in the bottom-right graph of the figure, only the number of message types is fixed and the parameters *s* and *q* are equal and varied from 1 to 10. As can be expected, the increase is now doubly exponential. The number of possible states quickly reaches astronomical values.

3.1. Modeling

The example illustrates the importance of abstraction, reduction, and intelligent model construction. Exponential effects can quickly make simple the properties of an uncarefully constructed model intractable, but inversely they can also help the model builder to prove subtle properties of complex systems by adjusting carefully chosen parameters. It is the objective of design verification to find ways to construct tractable models for software applications, so that their properties can be verified formally.

Call E the set of all possible runs of a given system. A model checking algorithm will attempt to demonstrate that E does not contain any run that violates a correctness requirement. Now consider a different system E' that contains all the runs contained in set E, and many more that are not contained in E. Now clearly, if E contains a violating run, than so will E', but not vice versa. This means that a reduction or abstraction method that extends the number of runs of a system, but that provably cannot remove any, has the desirable property that it is fail-safe. Abstractions of this type can dramatically reduce the number of reachable states of a system. Note that we can generalize a problem by removing constraints from it. The behavior of a model that is less specific often can be represented with fewer states.

3.2. Example – a file server

Assume our task is to verify the correctness of a transfer protocol that is used to access a remote file server. Our first obligation is to determine precisely which correctness properties the transfer protocol must have, and what may be assumed about the behavior of the file server and of the transmission channel.

Consider first the transmission channel. Assume the channel is an optical fiber link. The verifier's job is *not* to reproduce the behavior of this fiber link at the finest level of detail. The quality of a verification does *not* improve when we attempt to do so.

The model we construct should represent only those behaviors that are relevant to the verification task at hand. It need not contain information about the causes of those behaviors. If the fiber link has a non-zero probability of errors, than the possibility of errors must be present in our model, but little more. The types of errors modeled could include disconnection, message-loss, duplication, insertion or distortion. If all these types of error are present, and relevant to the verification task at hand, it should suffice to model the link as a one-state demon that can randomly disconnect, lose, duplicate, insert, or distort messages.

A fully detailed model of the link could require thousands of states, representing, for instance, the clustering of errors, or the nature of distortions. For a design verification of the protocol, however, it not only suffices to represent the link by a one-state demon: doing so guarantees a stronger verification result that is *independent* of clustering or distortion effects. A model that randomly produces *all* relevant events that can be part of the real link behavior satisfies the requirements for a fail-safe reduction strategy. It might add error runs, but it cannot remove them.

Next, consider the file server. It can receive requests to create and delete, open and close, read and write distinct files. Each such request can either succeed or fail. A read request on a closed file, for instance, will fail. Similarly, a create or write request will fail if the file server runs out of space. Again, for the verification of the interactions with the file server, we need not model in detail under what circumstances each request may succeed or fail. Our model of the server can again be a one-state demon that randomly accepts or rejects requests for service, without even looking at the specifics of the request.

Our one-state server would be able to exhibit behaviors that the real system would not allow, e.g., by rejecting valid requests. All behaviors of the real server, however, are represented in the abstract model. If the transfer protocol can be proven correct, despite the fact that our model server may behave worse than the real one, the result is stronger than it would have been if we had represented the server in more detail. By generalizing the model of the file server, we separate the correctness of the transfer protocol from detailed assumptions about on the server. The model that randomly produces *all* relevant events, is a fail-safe generalization of the server.

Finally, let us consider the number of message types and message queues that are needed to represent the interaction of user processes with the remote file server. If no single user can ever have more than one request outstanding, we need minimally three distinct types of messages, independent of how many distinct services the remote system actually offers. The three message types are *request*, *accept*, and *reject*.

If there are q users and only one server, the server must of course know which response corresponds to which request. Suppose that we use a single queue for incoming requests at the server, and mark each request with a parameter that identifies the user. This gives q distinct types of messages that could arrive at the server. If $q \times s$ is the total number of slots in that queue, the number of distinct states will be:

$$\sum_{i=0}^{q \times s} q^i .$$

What if we replaced the single queue with q distinct queues, each of s slots, one for each user? Now we need only one type of request, and the number of queue states is now $(s+1)^q$. Which is better? Note that every feasible state of the multiple queues can be mapped to a specific state of the single queue, for instance by simply concatenating all s slots of all q queues, in numerical order, into the $q \times s$ slots of the single queue. But the single queue has many more states, i.e., all those states that correspond to arbitrary interleavings of the contents of the multiple queues. With these parameters, then, it can make a large difference in complexity if we replace a single queue with a set of queues. To get an idea of the difference, assume $s = 5$ and $q = 3$, then the total number of states of all multiple queues combined is $(s+1)^q = 6^3 = 216$, and the total number of states of the single queue is

$$\sum_{i=0}^{q \times s} q^i \ = \ \sum_{i=0}^{15} 3^i \ = \ 21{,}523{,}360$$

or about five orders of magnitude larger. If the relative order of messages between queues is irrelevant, this can be a significant win. The choice of a model, then, and the level of detail that it represents, can have a very substantial impact on the feasibility of verification.

Assuming that we have the smallest possible model that still captures the essential features of a system, is there anything more we can do to reduce the complexity of the verification task? Fortunately, the answer is yes. We will briefly sketch the intuition behind one such technique: partial order reduction. We will also look in somewhat more detail at a different approach to the complexity problem: proof approximation.

3.3. Partial order reduction

Consider the automata $T1$ and $T2$ shown in Figure 10. In this representation the symbols that label the transitions are used to represent assignment statements in a simple C-like programming language. In this interpretation the two automata share access to a single integer data object named g, and they each have non-shared access to a private data object, named x and y respectively. Assume the initial value of all data objects is zero.

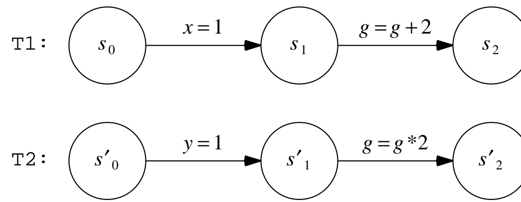


Fig. 10 — *Automata $T1$ and $T2$.*

The interleaving product of $T1$ and $T2$ is illustrated in Figure 11, where we have restricted ourselves to the proper interleaving of transitions (i.e., excluding simultaneous transitions). The state labels in Figure 11 are used to represent the values of the data objects, in the order: x, y, g.

The graph in Figure 11 represents all basic interleavings of the four statements in the systems $T1$ and $T2$. Clearly, the two interleavings of the transitions labeled $x = 1$ and $y = 1$ lead to the same result $x \equiv y \equiv 1$. The two interleavings of the transitions labeled $g = g + 2$ and $g = g * 2$, on the other hand, lead to two different values for g.

The system is small enough that we can exhaustively write down all possible runs. There are only six:

$$\sigma_1 = \{(0,0,0),(1,0,0),(1,0,2),(1,1,2),(1,1,4)\}$$
$$\sigma_2 = \{(0,0,0),(1,0,0),(1,1,0),(1,1,2),(1,1,4)\}$$
$$\sigma_3 = \{(0,0,0),(1,0,0),(1,1,0),(1,1,0),(1,1,2)\}$$
$$\sigma_4 = \{(0,0,0),(0,1,0),(0,1,0),(1,1,0),(1,1,2)\}$$
$$\sigma_5 = \{(0,0,0),(0,1,0),(1,1,0),(1,1,0),(1,1,2)\}$$
$$\sigma_6 = \{(0,0,0),(0,1,0),(1,1,0),(1,1,2),(1,1,4)\}$$

or, if we write them down in a more familiar form, as sequences of transition symbols:

1: *x = 1; g = g+2; y = 1; g = g*2;*
2: *x = 1; y = 1; g = g+2; g = g*2;*
3: *x = 1; y = 1; g = g*2; g = g+2;*
4: *y = 1; g = g*2; x = 1; g = g+2;*
5: *y = 1; x = 1; g = g*2; g = g+2;*
6: *y = 1; x = 1; g = g+2; g = g*2;*

Sequences 1 and 2 differ only in the relative order of execution of *y = 1* and *g = g+2*, which are independent operations. Similarly, sequences 4 and 5 differ in the relative order of execution of the independent operations *x = 1* and *g = g*2*, By a process of elimination, we can reduce the number of distinct runs to just two, for instance to:

2: *x = 1; y = 1; g = g+2; g = g*2;*
3: *x = 1; y = 1; g = g*2; g = g+2;*

All other runs can be obtained from these two by one or more permutations of adjacent independent operations. We have the following mutual dependencies:

*g = g*2* and *g = g+2*	because they touch the same data object,
x = 1 and *g = g+2*	because they are both part of T1,
y = 1 and *g = g*2*	because they are both part of T2.

The following operations are mutually independent:

x = 1 and *y = 1*,
x = 1 and *g = g*2*,
y = 1 and *g = g+2*.

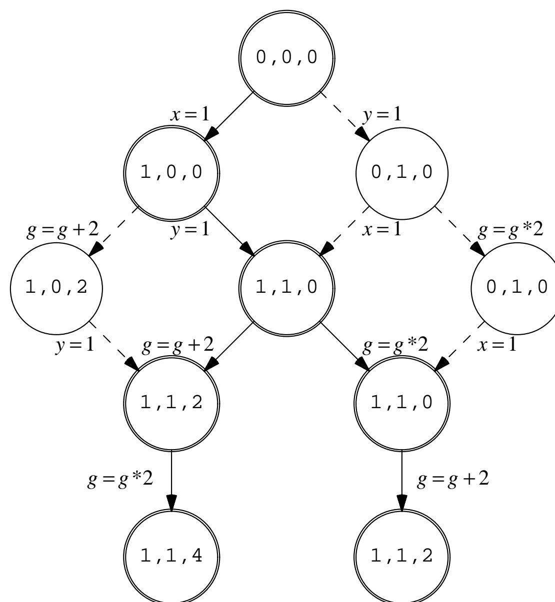


Fig. 11 — Full and Reduced Depth-First Search for $T1 \times T2$.

Using this classification of dependent and independent operations, and transitions, we can partition the runs of the system into two equivalence classes: {1,2,6} and {3,4,5}. Within each class, each run can be obtained from the other runs by one or more permutations of adjacent independent transitions. The eventual outcome of a computation remains unchanged under such permutations. For verification it therefore would suffice to consider just one run from each equivalence class.

For the system from Figure 11 it would suffice, for instance, to consider only runs 2 and 3. In effect this restriction amounts to a reduction of the graph in Figure 11 to the portion spanned by the *solid* arrows, and including only the states indicated in *bold*. There are three states fewer in this graph and only half the number of transitions, yet it would suffice to accurately prove LTL formulae such as:

$\Box\,(g \equiv 0 \ \vee \ g > x)$,

$\Diamond\,(g \geq 2)$,

$(g \equiv 0)\ \boldsymbol{U}\ (x \equiv 1)$,

3.4. Visibility

Would it be possible to formulate LTL properties that hold in the reduced graph, but that are violated in the full graph? To answer this question, consider formula

$$\Box (x \geq y).$$

This formula indeed has this unfortunate property. So what is different? The formula secretly introduces a dependence that was assumed not to exist: it relates the values of the data objects x and y, while we earlier used the assumption that operations on these two data objects were always independent. The dependence of operations, therefore, does not just depend on automata structure and access to data, but also on the logical properties that we are interested in proving about a system. If we remove the pair $x = 1$ and $y = 1$ from the set of mutually independent operations, the number of equivalence classes of runs that we can deduce increases to four, and the reduced graph gains one extra state and two extra transitions. The new graph will now correctly expose the last LTL formula as invalid.

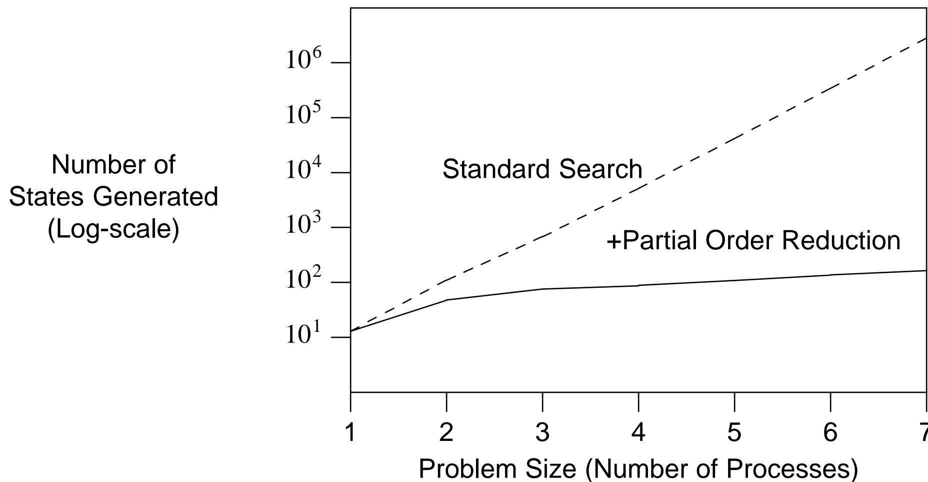


Fig. 12 — *Effect of Partial Order Reduction.*
Best Case Performance. Leader Election Protocol with N Processes.

The potential benefits of partial order reduction are illustrated in Figure 12. Shown is the reduction in the number of states in the product graph that needs to be explored to perform model checking when partial order reduction is either enabled (solid line) or disabled (dashed line). In this case, the improvement increases exponentially with the problem size. It is not hard to construct cases where partial order reduction cannot contribute any improvement (e.g., if all operations are dependent). The challenge in implementing this strategy in a model checker is therefore to secure that in the worst case the graph construction will not suffer any noticeable overhead. This was done in the **Spin** model checker with a *static reduction method*. In this case, the dependency relations are computed offline, before a model checking run is initiated, so that no noticeable runtime overhead is incurred.

To formalize the many notions we have casually introduced above, we need to introduce the formal framework of an *extended finite automaton*, which includes the definition of a context of data objects and an interpretation of transition symbols within that context. We must also formalize the notion of *dependence* of operations, *equivalence* of runs, and *equivalence robustness* of properties. For detailed treatments of these notions we refer to [Ma87, Kw89, P94]. A description of the implementation of partial order reduction techniques within the **Spin** model checker can be found in [HP94], with a small adjustment that is explained in [HPY96]. A formal proof of correctness of the algorithm is given in [CP99].

3.5. Proof approximation

The first automated verification systems based in graph analysis were developed about twenty years ago today. Since then, the computational complexity has been the single most dominant issue that is being addressed in this field. We have better algorithms today, smarter tools, and significantly more powerful

machines, but the problems we attempt to solve have also grown in size. The problem of managing computational complexity is still the single most dominant issue in this field, and given the nature of the problem we are attempting to solve, it is likely to remain that way.

With proper abstraction and modeling techniques, with reduction and minimization algorithms, and with access to the largest computers impressive results have been achieved. Where twenty years ago it could be a challenge to verify the toy alternating bit protocol, today we can verify complex software spanning several thousands of lines of code. Yet, one does not have to look far to find examples of applications where the computational resources that would be required to rigorously verify the simplest model of the smallest separable piece of the application still exceed practical constraints. The question is now what we can do in these situations. We can blame the model builder, and wait for a better model. We can blame the machine, and wait for more a powerful one. We can blame the verification tool, and wait for better algorithms. Or, we can try to design a different type of search algorithm, that attempts to approximate the result of a verification as closely as possible within the currently available constraints, whatever those constraints may be. We will investigate such techniques here.

To begin, let us look at the memory requirements of model checking. The depth-first search algorithm, discussed in part II of these notes, constructs a set of states. Each state in the intersection of the property automaton and the interleaving product of the component automata is stored in a *statespace*. Since the model checking problem for all practical purposes is reduced to the solution of a reachability problem, all the model checker does is to construct states and to check whether they were previously visited or new. The performance of a model checker is determined by how fast we can do this.

The statespace structure serves to prevent the re-exploration of previously visited states during the search: it turns what would otherwise be an exponential algorithm into a linear one, that visits every reachable state in the graph at most once. To enable fast lookup of states, the states are normally stored in a hash-table, as illustrated in Figure 13.

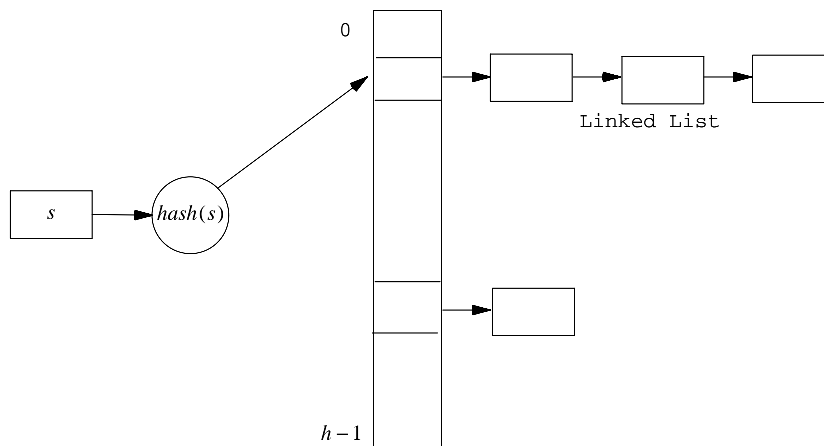


Fig. 13 — *Standard Hash Table Lookup.*

Assume we have a hash-table with h slots. Each slot contains a list of zero or more states. To determine in which list we store a new state s, we compute a hash-value $hash(s)$, unique to s and randomly chosen in the range 0..h-1. We check the states stored in the list in hash-table slot $hash(s)$ for a possible match with s. If a match is found, the state was previously visited and need not be explored again. If no match is found, state s is added to the list, and the search continues.

> Each state is represented in memory as a sequence of S bits. A simple (but poor) hashing method would be to consider the array of bits as one large unsigned integer, and to calculate the remainder of its division by h, with h a prime number. A more efficient method, used in the model checker **Spin**, is to use a *checksum polynomial* to compute the hash values. We now chose h as a power of 2 and use the polynomial to compute a checksum of $log(h)$ bits. This checksum is then used as the hash value.

Let r be the number of states stored in the hash-table and h the number of slots in that table. When $h >> r$, each state can be stored in a different slot, provided that the hash function is of sufficiently good quality.

The lists stored in each slot of the hash-table will either be empty or contain one single state. State storage has only a constant overhead in this case, carrying virtually no time penalty.

When $h < r$, there will be cases for which the hash function computes the same hash value for different states. These *hash collisions* are resolved by placing all states that hash to the same value in a linked list at the corresponding slot in the hash-table. In this case we may have to do multiple state comparisons for each new state that is checked against the hash-table: towards the end of the search on average r/h comparisons will be required per state. The overhead incurred increases linearly with growing r/h, once the number of stored states r exceeds h.

Clearly, we would like to be in the situation where $h >> r$. In this case, a hash-value uniquely identifies a state, with low probability of collision. The only information that is contained in the hash-table is now primarily whether or not the state that corresponds to the hash-value has been visited. This is one single bit of information. A rash proposal is now to indeed store only this one bit of information, instead of the S bits of the state itself. This leads to the following trade-offs.

Given m bits of memory to store the hash-table, S bits of data in each state descriptor, r reachable states, and a hash-table with h slots. Clearly, fewer than m/S states will fit in memory, since the hash-table itself will also take some memory. If $r > m/S$ the search will exhaust the available resources (and stop) after exploring a fraction of $m/(r.S)$ of the statespace. Typical values for these parameters are: $m = 10^9$, $S = 10^3$, and $r = 10^7$, which gives a ratio $m/(r.S) = 10^{-2}$, or a coverage of the problem size of only 1%.

If we configure the hash-table as an array of $8.m$ bits, using it as a hash-table with $h = 8.m$ 1-bit slots, we now have $h >> r$, since $8.10^9 >> 10^7$, which should give us an expected coverage close to 100%. When, with low probability, a hash-collision happens, our model checking algorithm will conclude incorrectly that a state that was visited before, and it will skip it. It may now miss other states that can only be reached via a path in the reachability graph that passes through this state. This, therefore, would lead to loss of coverage, but it cannot lead to false error reports. We will see below that in almost all cases where this method would be used (i.e., when normal state storage is impossible due to limited resources available) coverage will increase far more due to the increased capacity to store states than it is reduced due to hash collisions.

This storage discipline was referred to by former Bell Labs colleague Robert Morris in 1968 as follows:

> ''A curious possible use of virtual scatter tables arises when a hash address can be computed with more than about three times as many bits as are actually needed for a calculated address. The possibility that two different keys have the same virtual hash address becomes so remote that the keys might not need to be examined at all. If a new key has the same virtual hash address as an existing entry, then the keys could be assumed to be the same. Then, of course, there is no longer any need to keep the keys in the entry; unless they are needed for some other purpose, they can just be thrown away. Typically, years could go by without encountering two keys in the same program with the same virtual address. Of course, one would have to be quite certain that the hash addresses were uniformly spread over the available addresses.
> No one, to the author's knowledge, has ever implemented this idea, and if anyone has, he might well not admit it.'' [M68]

To reduce the probability of collision, we can use multiple independent hash-functions, and set more than one bit per state. Using more bits can increase the precision but reduce the number of available slots in the bit hash-table. The trade-offs are delicate and deserve a more careful study.

3.6. Bloom filters

The method we have described was first proposed for use in a verification tool in [H88]. It is closely related to a method known as a Bloom Filter, described by Burton Bloom in 1970 [B70].

Let again m be the size of the hash-table in bits, r is the number of states stored, and k the number of hash-functions used. (That is, we store k bits for each state stored, with each of the k bit-positions computed with an independent hash-function that uses the S bits of the state descriptor as the key.)

Initially the hash-table will contain only zero bits. When r states have been stored, the probability that any one specific bit is still 0 will be:

$$\left[1 - \frac{1}{m}\right]^{k.r}$$

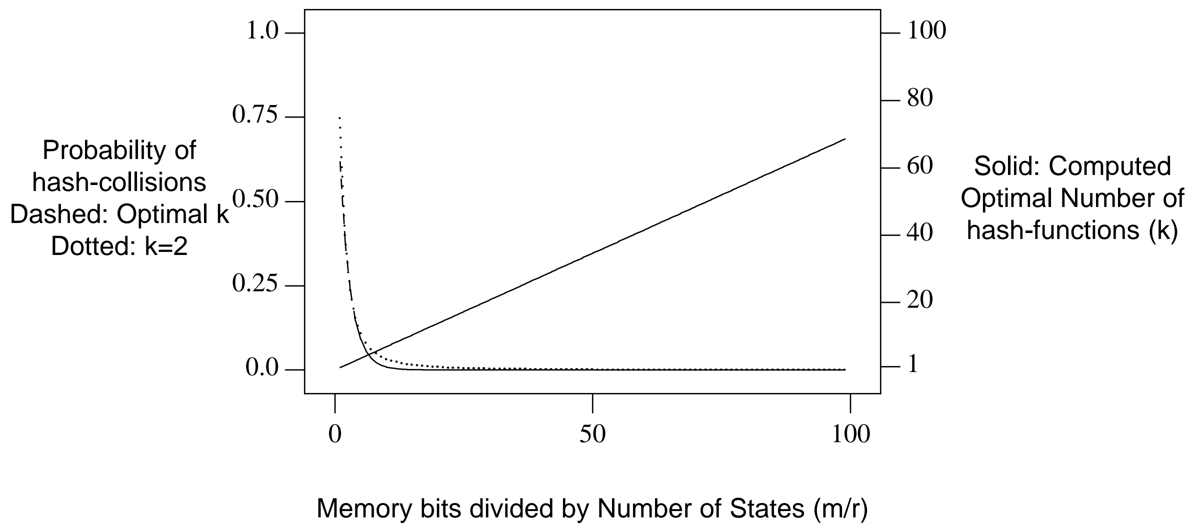
The probability of a hash-collision on the $(r+1)$st state entered would then be

$$\left[1-\left[1-\frac{1}{m}\right]^{k.r}\right]^{k} \approx \left[1-e^{-k.r/m}\right]^{k}$$

which gives us an upper-bound for the probability of hash-collisions on the first r states entered. (E.g., the probability of a hash-collision is trivially 0 for the first state entered.) The probability of hash-collisions is minimized for the value of $k = log(2).m/r$, which gives

$$\left[\frac{1}{2}\right]^{k} = 0.6185^{m/r}$$

For $m = 10^9$ and $r = 10^7$ this gives us an upper-bound on the probability of collision in the order 10^{-21}, for a value of $k = 89.315$.



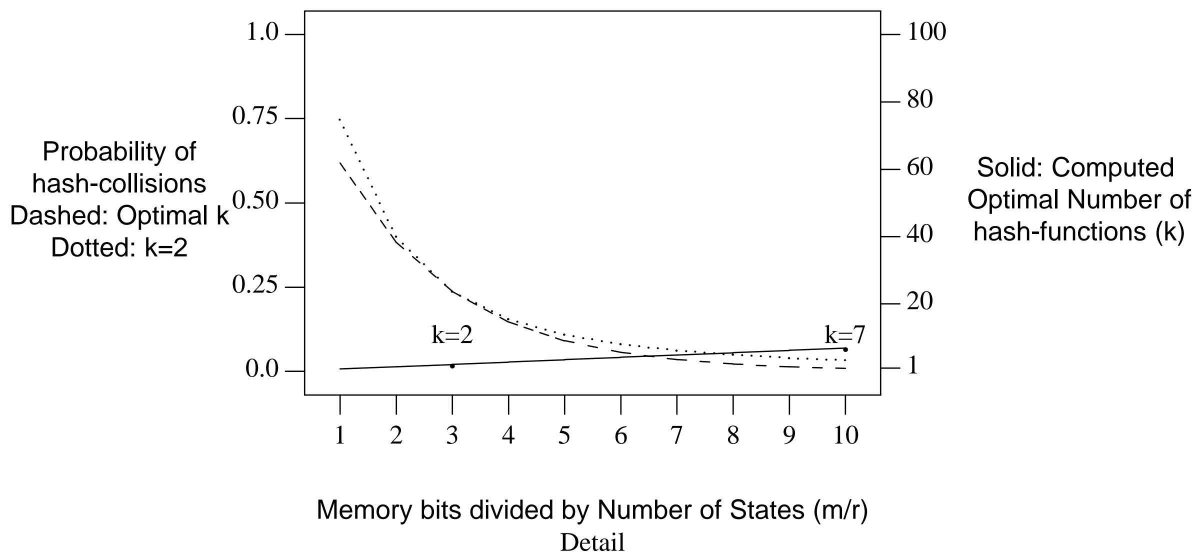


Fig. 14 — *Optimal Number of Hash-Functions and Probability of Hash-Collision.*

Figure 14 illustrates these dependencies.

In practice k must be an integer (e.g., 90). In a well-tuned model checker, the runtime requirements of the search depend linearly on k: computing hash-values is the single most expensive operation that the model checker must perform. The larger the value of k, therefore, the longer the search for errors will take. In the model checker **Spin**, for instance, a run with $k = 90$ would take approximately 45 times longer than a run with $k = 2$. Although time is a more flexible commodity than memory, the difference is significant. The question is then how much quality we sacrifice if we select a smaller than optimal value of k. The trade-off is illustrated in Figure 14.

For the suboptimal value $k = 2$, the value used in the model checker **Spin** [H97], the upper-bound on the collision probability becomes 4.10^{-4}, which reduces the expected coverage of the search from 100% to near 99%, still two orders of magnitude greater than realized by a hash-table lookup method for this case. We can also see in Figure 14 that the hashing method starts getting very reliable for m/r ratios over 100. To be compatible with traditional storage methods, this means that for state descriptors of less than 100 bits (about 12 bytes) this method is not competitive. In practice, state descriptors exceed this lower-bound by a significant margin (one or two orders of magnitude).

An interesting variant of this strategy was proposed in [W93], and named *hash-compact*. In this case we try to increase the size of m far beyond what would be available on a normal machine, e.g. to 2^{64} bits. We now compute a single hash-value within the range $0..(2^{64}$-1), as a 64-bit number, and store this number in a regular hash-table, as shown in Figure 13, instead of state s. We have $k = 1$ in this case, effectively $m = 2^{64} \approx 10^{19}$. For the value of $r = 10^7$ we then get a probability of collision near 10^{-57}, giving an expected coverage of 100%. To store 10^7 64-bit numbers takes less than $m = 10^9$ bits, so also this method works. The maximum value of r for which we could get the superior performance of the hash-compact method is of course m/r.

Both the hash-compact method and double-bit hashing have been implemented in **Spin** [H97].) A measurement of the performance of these two methods for a fixed problem size $r = 427,567$ and the amount of available memory m varying from 0 to $m > r.S$ is shown in Figure 15, which is taken from [H98].

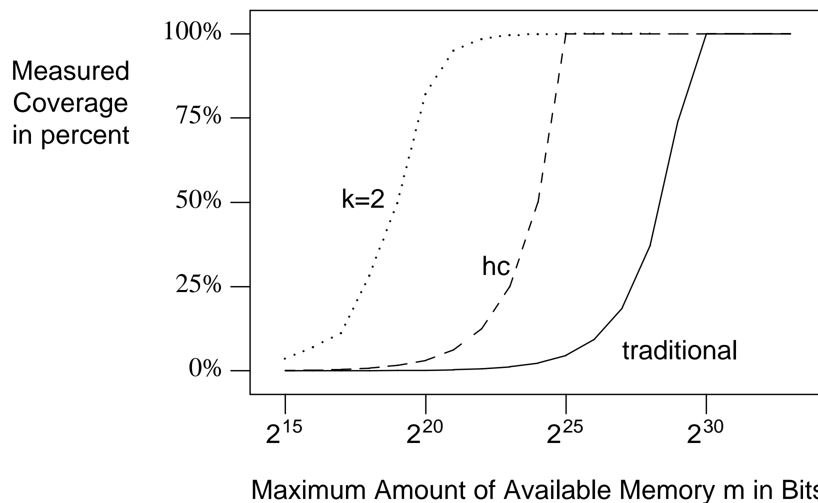


Fig. 15 — *Measured Coverage of Hash-Compact (hc) and Double Bitstate Hashing (k=2), for varying m, and fixed r=427567 states and S=1376 bits.*

When sufficient memory is available, traditional state storage is preferred. Barring this, if sufficient memory is available for the hash-compact method, then this is the preferred method. Beyond that the double-bit hashing method is superior. The latter method, for instance, still achieves a problem coverage of 50% when only 0.1% of the memory resources required for a full traditional search are available.

The coverage of both the hash-compact and the double-bit hashing method can be increased further by performing multiple searches, each time with an independent set of hash-functions [H98]. If each search

misses a fraction p of the state space, t independent searches could reduce this to p^t. Though expensive in runtime, this gives us a capability to increase the quality of a verification under adverse constraints.

4. Model extraction

The most powerful tool we have in our arsenal for the verification of software applications is abstraction. By capturing the essence of a design in a mathematical model, we can often demonstrate that the design must have certain inevitable properties. The very purpose of a model is to enable proof. If it fails to do so, with the tools that are available to the prover, the model should be considered inadequate.

We could stop here, and merely illustrate the point by presenting some examples of poorly constructed and well-constructed models [H98b], sketching the types of abstraction techniques that are useful in building verifiable models of software applications. There are some problems with this approach though. First, finding the right abstraction can be hard. It takes time to develop the insight that is needed to capture the essence of a software design at the proper level of abstraction. More often than not, one only realizes what the proper level of abstraction was some time after a verification attempt is completed. Software projects, especially in industry, face strict deadlines, leaving little room for reflection or detailed consultation with software designers. A hurried, and inadequate choice of an abstraction can trigger long an frustrating battles with run-away complexity. Worse, the choice of an invalid abstraction can give a false sense of security by causing the verifier to miss design errors alltogether.

An alternative method, that we shall explore here, is to use abstraction techniques in a systematic manner to extract verification models mechanically from software artifacts (source code). As an example, we will consider programs written in a relatively low-level imperative programming language such as C. The model extraction process then proceeds in four steps: parsing, interpretation and abstraction, simplification, and finally conversion into the format accepted by the model checker.

Parsing.

The program source text is converted into a finite automaton structure (also known as a *control flow graph*, a *parse tree* or an *abstract syntax tree*). The states in the automaton are the control flow points of the program, and the transitions are labeled with the declarations, conditions, and basic statements from the program text. The finite automaton structure is constructed such that it preserves all information necessary to reconstruct the original program source text, no more and no less.

Interpretation and Abstraction.

The program is now in a standard form where abstraction techniques can be applied, e.g., [AL91],[CGL94]. We can also apply program slicing techniques [T95],[CD00], with the slice criteria derived from the program properties to be proven. Slicing algorithms allow us to construct the smallest program fragment that preserves all access to all data objects mentioned in the properties, and all entries to an exits from the corresponding program locations. We can also postpone slicing until after a base model has been generated from the program source, and use model-based slicing techniques, e.g., as supported in the **Spin** model checker (version 3.4.0 and later). We will discuss these and other types of abstraction in more detail below.

Simplification.

Next, the abstracted program can be simplified and, optionally, optimized by using standard techniques used in compiler construction. This includes rewriting, dead variable elimination, dead code elimination, constant propagation, loop unfolding etc.

Conversion.

The final step is to translate the abstracted and simplified program model into the syntax of the model checker used, and to write it out. This stage is similar to the final code-generation phase of a compiler, but since the target is high-level, rather than low-level code, the step is relatively straightforward here. We can benefit from the fact that the control flow of the application is usually trivial to convert from one format into another, and simple abstraction techniques can be used to bridge any syntactic gap between the source implementation language and the target modeling language.

The main types of abstraction can be used in the model extraction process are as follows.

- *Slicing* [T95] can be used to reduce a program source to a smaller fragment, of lower complexity, that contains only part of its functionality. The slice point can be given as a reference to a a specific set of

data objects, e.g. the data objects that are referred to in the property to be proven or in a particular statement of interest. All code that is directly relevant to the manipulation of these data objects is preserved in the slice and the rest is hidden. Slicing algorithms are based on data and control dependency analysis of the program text. The objective of the slicing algorithms is to identify those parts of the program that are irrelevant with respect to the properties to be proven. Since all properties of interest are necessarily preserved under this abstraction we can guarantee that if the original program can violate a property of interest, then so can the sliced version and vice versa. Property-based slicing has the desirable characteristics of being both sound and complete; it permits neither false negatives nor false positives during the final verification. As we shall see, this is not necessarily true for other abstraction and reduction techniques.

- *Predicate Abstraction* [GS97], [DDP99] and *Mapping* [CGL94], can be used to reduce the value ranges of data objects, e.g., from integer to Boolean values. If, for instance, the correctness property requires us to determine if a specific timer is running or not, but does not require knowledge of it's precise value, then we can map the integer data object that holds the timer value to a Boolean object, with an appropriate mapping function. We can use the assistance of a theorem prover or of specialized decision procedures to prove that mappings are applied consistently, and together define a sound abstraction. In general, this type of abstraction can guarantee that if the program allows a property violation then so will the model, but not necessarily vice versa: it is sound (cannot produce false positives) but not necessarily complete (it may produce false negatives). A false negative is counter-example that shows that a property can be violated in the abstract model, that cannot be reconstructed for the concrete model. It means that information was lost in the abstract that turns out to be relevant for distinguishing incorrect from correct runs. In most cases the counter-example contains sufficient information to allow the user to remove the false negative by revising the abstraction that was applied.
- *Generalization* is a method by which we introduce non-determinism to remove irrelevant detail from a model. The generalization is defined in such a way that the number of runs of the system of the whole strictly increases. The system can still perform *all* the executions that it could before the generalization was applied, but it now permits also additional executions. And execution that violates the property must therefore still be present, so the method is sound. The added executions, however, could themselves violate the property, and thereby introduce false negatives, so like predicate abstraction, this method is not necessarily complete. An example of generalization is to replace a process with a random demon that can generate all externally visible events that the original process can generate non-deterministically. (E.g., to model a subscriber in a telephone system with a demon that randomly generates on-hook, off-hook, and digit events.)
- *Restriction* can be used to restrict the scope of the verification to a subset of the problem. We can for instance restrict the capacity of buffers, the number of active processes, the dimensions of arrays, etc. In general, there will be no guarantee that essential correctness properties are preserved under these abstractions. In formal terms, this abstraction method is neither sound nor complete: it can introduce both false negatives and false positives. Nonetheless, the method can be useful in an exploratory phase of a verification effort, to study problem variants with possibly lower complexity than the full problem that is to be solved.

We next discuss three main types of abstraction methods in more detail below: program and model slicing, predicate abstraction, and tabled abstraction [HS99a],[HS99b].

5. Program and model slicing

As an example we will consider a simple wordcount program, written in Promela, the input language of the **Spin** model checker. The program receives characters, encoded as integers, over the channel `stdin`, and simply counts the number of newlines, characters, and white-space separated words, up to an end-of-file marker which is encoded as the number `-1`.

If we wish to verify that this program maintains the invariant $\Box\,(nc \geq nl)$, then clearly all manipulation of the variables `nw` and `inword` are irrelevant. In a first step of a slicing algorithm, only the variables `nc` and `nl` are marked as relevant. These two variable become the slice criteria for deriving a reduced model that will suffice to prove, or as we shall discover disprove, the property.

The slicing algorithm now performs a data-flow analysis, marking all statements where the relevant variables are either used (i.e., read) or defined (i.e., assigned a value). These statements appear on lines 11 and 12. Next, we perform a control-dependency analysis for each of the three currently marked statements. The marked statements are *control-dependent* on every statement in the model that can affect their execution (e.g., preventing it by blocking). For our wordcount program this applies to the conditionals `c == '\n'` on line 11, and `c == -1` on line 10. Note, for instance, that if the latter condition evaluates to *true*, the relevant statements cannot be reached. These two conditionals are now marked as relevant.

```
1  chan stdin = [1] of { int };
2  int c, nl, nw, nc;
3  bool inword = false;
4
5  active proctype wordcount()
6  {      /* count number of lines, words, and chars received */
7         do
8         :: stdin?c ->
9                 if
10                :: c == -1 -> break        /* end of input */
11                :: c == '\n' -> nc++; nl++
12                :: else -> nc++
13                fi;
14                if
15                :: c == ' ' || c == '\t' || c == '\n' ->
16                        inword = false
17                :: else ->
18                        if
19                        :: !inword ->
20                                nw++; inword = true
21                        :: else /* do nothing */
22                        fi
23                fi
24        od;
25        printf("%d\t%d\t%d\n", nl, nw, nc)
26 }
```

The data-objects referred to in the newly marked statements become data-dependent on the property, and we repeat the control dependency analysis. In the second phase we now discover the input statement on line 8 as both data-dependent (it assigned a value to the data-dependent variable `c`) and control-dependent (if it blocks, none of the other relevant statements can be reached). This marks the channel `stdin` as data dependent. We continue this process of performing alternately a data dependency analysis and a control dependency analysis until a fixedpoint is reached.

In the final slice, the program fragment on lines 14-23, and the one statement on line 25 is marked as irrelevant to the proof of the invariant property. Note for instance that even though the statements on line 15 refers to the relevant variable `c`, it cannot change the value of that variable, and therefore properly remains outside the slice.

We can now perform model checking on the reduced model. Before we can do so, however, we must close the model to its environment. That is, we must encapsulate inside the model all the essential assumptions that have to be made about external processes that the process considered can interact with. These assumptions are essential for the proof to be performed. In the case of the wordcount program we must formalize our assumptions about the external source of the characters that are being counted. Using a *generalization* technique, we can think of a first approximation of an external process that sends randomly selected symbols from the ASCII character set, plus the specially designated end-of-input marker.

We can do better though, by applying another simple form of abstraction. If we collect all the uses of the input variable `c` in the (remaining) text of the wordcount program we see that only three ranges of values of the variable are of interest: (1) newlines, (2) end-of-input markers, and (3) any other symbol. It suffices therefore to restrict the input stream to just three abstract symbols, representing the three relevant value ranges. The following environment definition suffices for the sliced model.

```
#define newline          '\n'
#define eof              -1
#define anythingelse     0

active proctype input()
{
        do
        :: stdin!newline
        :: stdin!eof
        :: stdin!anythingelse
        od
}
```

Note that for the non-sliced program, we would have had to add two extra symbols, for space and tab, to capture the close the system to its environment. We can now combine this with the sliced model derived before, to complete the verification model.

```
chan stdin = [1] of { int };
int c, nl, nc;

active proctype wordcount()
 /* sound and complete slice for □ (nc ≥ nl) */
{
        do
        :: stdin?c ->
                if
                :: c == eof -> break
                :: c == newline -> nc++; nl++
                :: else -> nc++
                fi
        od
}
```

If we perform the verification of the invariant $\Box$ (nc $\geq$ nl), we discover that this property can be violated. Because we have only performed sound and complete abstractions, this is necessarily a valid counter-example and cannot be a false negative.

The counter-example shows that when the value of variable `c` wraps around its maximal value (exceeding the range of `int`) it can become smaller than the value of `nl`: an obvious consequence of the fact that all value ranges are necessarily finite. It is not a problem we are likely to run into in practice, except for exceedingly large inputs.

5.1. Slicing algorithm

We will describe the core of the slicing algorithm that is included in the **Spin** model checker. For model checking applications the slicing algorithm has to be slightly different from the traditional methods, as for instance described in [T95]. We cannot, for instance, safely remove a cyclic component from the control-flow graph, even if each individual statement in such a cycle is independent of the property to be proven. Removal of such a cyclic component would require a proof of termination, which in general cannot be done by static analysis. Observe that absence of termination would affect the liveness properties of a program. Our approach to slicing, therefore, proceeds in three phases. In the first phase, we identify all statements that are relevant to the property to be proven. In the second phase, all non-relevant statements are replaced by the null-statement *skip*, but the structure of the control-flow graph is not modified.

Only in the last phase of the slicing algorithm do we simplify the control-flow graph, while taking care to preserve all liveness properties. A non-cyclic subgraph in the control-flow graph consisting only of *skip* statements, for instance, can be collapsed down to a single statement, as illustrated in Figure 16. Note that cycles are preserved under this transformation.

We will now look at the slicing algorithm that was implemented in **Spin** Version 3.4.0 in more detail. The input language for **Spin** defines three basic types of objects: processes, local and global variables, and message channels. Processes can interact via synchronous or asynchronous message passing, or via unrestricted access to global variables. In particular, there are no pointers or recursive functions in **Spin**

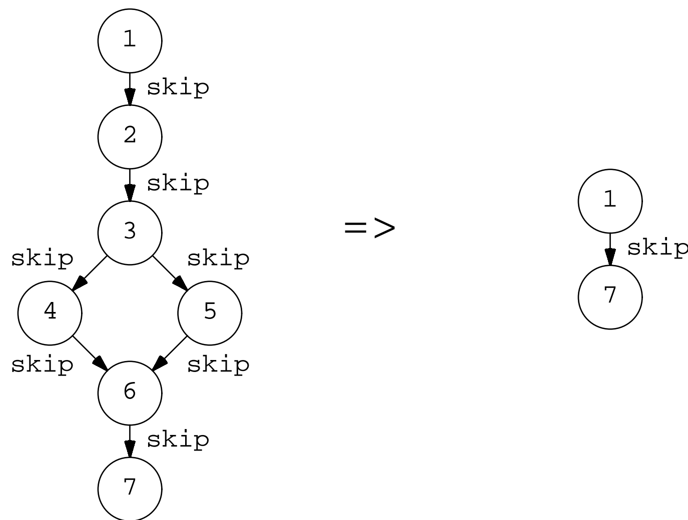


Fig. 16 — *Simplification of the Control-Flow Graph After Slicing.*

verification models: the two features that can complicate the design of a slicing algorithm for a general purpose programming language. Assume the following input to the slicing algorithm.

`S` is the set of all basic statements in the program (like assignments, conditions, send operations, receive operations, process instantiations, etc.). The set of basic statements is identical to the set of labels on the transitions in the control-flow graph of the program considered.

`B` is the subset of S that contains all basic statements that can block, that is all those statements of which the execution is conditional on the system state. The execution of an assignment statement, for instance, is always unconditional. The execution of a receive operation on message channel *m* is conditional on the non-emptyness of *m*.

`T(s,t)` or as we shall write `s ~> t`, is a function on the elements of `S` that returns true if and only if statement `t` is reachable from statement `s` within the control flow graph of the program. If `s` and `t` are statements from different processes, then `s ~> t` will always be false.

`D` is the set of all local and global data objects in the program.

`C` is a subset of `D` that contains all pending slice criteria. The initial contents of `C` is derived from the program properties that are to be proven: each data object that is referred to in at least one property becomes a slice criterion.

`P` is a subset of `D` that contains all processed slice criteria. This set is initially empty.

`Def(s)` with `s` ∈ `S`. `Def(s)` contains all those elements of `D` that are *defined* (i.e., can be assigned a new value) in statement `s`.

`Use(s)` with `s` ∈ `S`. `Use(s)` contains all those elements of `D` that are *used* (i.e., evaluated) in statement `s`.

The objective of the slicing algorithm is to compute a set `R`, which is the subset of `S` that contains all statements in `S` that are relevant with respect to the initial set of slice criteria, and hence relevant to the verification of the property. All statements in `S\R` can be replaced by a null operation *skip* in the model without altering the outcome of a verification attempt in any way.

In the description of the algorithm below we will use the following three operations on sets.

`Empty(x)` is a boolean function that returns trueif set `x` is empty, and otherwise false.

`Get(x)` removes an element from set `x` and returns it as the result of the operation. And finally,

`Put(y,x)` adds element `x` into the set `y`.

Initially, set `P` is empty. The algorithm computes the set of statements that must be contained in the slice in set `R`.

```
while (!Empty(C))
{
        d = Get(C);       # start processing slice criterion d
        Put(P, d);        # mark it as processed

        X = { s | s ∈ S ∧ d ∈ Def(s) }
        R = R ∪ X         # add data dependent statements

        U = { u | ∃ s, s ∈ X ∧ u ∈ Use(s) ∧ u ∉ (P ∪ C) }
        C = C ∪ U         # add new slice criteria

        Z  = { s | s ∈ B ∧ ∃ t, t ∈ X ∧ s ~> t }
        W = { u | ∃ s, s ∈ Z ∧ u ∈ Use(s) }
        C = C ∪ W         # add control-dependencies
}
```

Termination: The algorithm presented above will terminate. First observe that for **Spin** models input set `D` is guaranteed to be finite. In each cycle of the algorithm precisely one element from set `C` moves to set `P`, and only elements from set `D` that are not already contained in either `C` or `P` can be added as new elements to `C`. In maximally $|D|$ iterations of the algorithm, then, all elements of `D` will be present in `P` and no further elements can be added to `C`. Since `C` shrinks by one element in each iteration, set `C` will reduce to empty within a finite number of iterations, at which point the algorithm terminates.□

There are two special cases that must be dealt when the algorithm is implemented, e.g., as it is in **Spin** Version 3.4.0. The first is that without precautions the control-dependency analysis would be too strict. For the computation of set `Z` we can exclude statements from consideration if those statements are part of *bubble* in the control-flow graph. A *bubble* is a subgraph of the control-flow graph that satisfies the following three conditions.

- The subgraph has one unique entry point `e` and one unique exit point `x`, both elements of set `S`: no statement inside the subgraph can be reached other than by passing through `e`, and no statement outside the subgraph can be reached other than by passing through `x`.
- All *decisions* in the subgraph are non-blocking. This means that either all statements in the subgraph are either non-blocking, or the immediately preceding statements in the control-flow graph also have `else` as one of their successors.
- The subgraph contains no single statement that is currently contained in set `R`.

The subgraph defined by the program fragment from lines 14-23 in the wordcount example at the start of this section satisfies these three conditions, and therefore need not add any control dependencies.

Bubble subgraphs can be found by computing the dominators for each node in the control-flow graph, cf. [ASU86]. The dominators are computed twice: once for the control-flow graph as given and once for the same graph with the direction of all edges reversed.

The second special case has to do with the use of channels in **Spin** models. The question has to do with the initial computation of the sets `Def(s)` and `Use(s)` for each statement in `S`. Consider the simple send and receive statements:

```
s1:      q1!w
s2:      q2?v
```

Statement `s1` sends the value of variable `w` over channel `q1`, and statement `s2` receives a value from channel `q2` and assigns it to variable `v`. We must have

```
Def(s1) = { }
Use(s1) = { w, q1 }
Def(s2) = { v }
Use(s2) = { ?, q2 }
```

What should be the missing entry in `Use(s2)`? Clearly, the value assigned to the variable `v` came from somewhere. If `q2` and `q1` refer to the same channel object, then we can fill in:

```
Use(s2) = { w, q2 }
```

In **Spin** models there are only three ways in which a channel name can be aliased: by assignment, by message passing (channels can be passed as parameters through channels), and by process instantiation (by passing a channel object as a parameter to the newly created process).

This means that it is relatively straightforward to perform a channel alias analysis that associates with each channel object all instantiated channels that the object could point to. To determine, then, which data objects must be entered into the definition of `Use(s2)`, we inspect the alias list of `q2`, and locate all statements in the model that can perform send operations on the channels that appear in this list. All variables used as parameters in these send operations, that appear in their `Use` definitions, are now added to `Use(s2)`. For the example, with `q1` and `q2` pointing to the same channel, this identifies `w` as the missing element, as intended.

6. Predicate abstraction

Another well-understood abstraction technique is predicate abstraction [GS97],[DDP99]. If, for instance, in the property we are interested in the sign of a data object, but not in it's absolute value, we can replace every occurrence of this data object with a new variable that captures only its sign, but not its value. For example, if the property is:

$$\Box((x < 0) \rightarrow \Diamond(x \geq 0))$$

and the program contains statements such as

$$x = 0;$$
$$x{+}{+};$$

and conditional such as

$$(x > 5)$$
$$!(x > 5)$$

we can replace all occurrences of the variable *x* with a new boolean variable *neg_x*. The property is rewritten as:

$$\Box((neg_x) \rightarrow \Diamond(\neg neg_x))$$

the assignments and conditions are now mapped as shown in Table 1, using *ND(a,b)* to indicate a non-deterministic choice of *a* or *b*.

Table 1 — Predicate Abstraction.

Concrete	Abstract	
	neg_x	¬ *neg_x*
x = 0;	*neg_x = false;*	*skip;*
x++;	*neg_x = ND(true, false);*	*skip;*
(x > 5)	*false*	*ND(true, false);*

Under the abstraction, precise information about the value of variable *x* is replaced with non-deterministic guesses about the possible new values of the boolean *neg_x*. For instance, when *neg_x* is currently *true*, and the value of *x* is incremented, the new value of *x* could be either positive or remain negative. This is reflected in a non-deterministic choice in the assignment of either *true* or *false* to *neg_x*. If, however, *x* is known to be non-negative, it will remain so after the increment, and the value of *neg_x* remains *false* in this case.

Given a data object with domain V. Call *M* the function that maps values from the concrete domain V to an abstract domain A, i.e., $\forall v \in V,\ M(v) \subseteq A$. A requirement on the validity of the abstraction is that we can define a reverse function R that lifts abstract values back into the concrete domain, in such a way that [CW00],[CC76],[DGG97]:

$$\forall v \in V,\ v \subseteq R(M(v)) \ \wedge \ \forall m \in A,\ m \equiv M(R(m))$$

i.e., such that *M* and *A* form a Galois connection. The relations are illustrated in Figure 17.

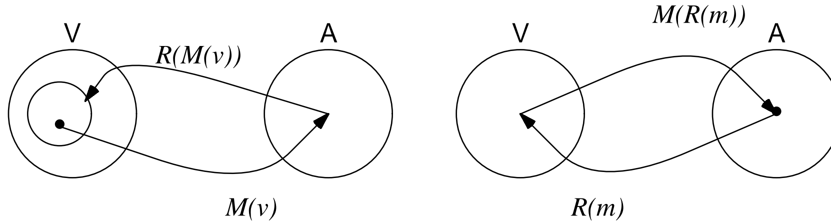


Fig. 17 — *Abstraction and Concretization.*

These relations hold for the sample abstraction mapping from the integer variable *x* to the boolean variable *neg_x*. Note that $R(M(v))$ is a set. Depending on the original value of v, this set includes either all values $v \geq 0$ or all values $v < 0$, as intended.

Predicate abstractions can in some cases be computed mechanically for restricted types of statements and conditions, e.g., when we restrict to Pressburger arithmetic. In this case, one can use a mechanized decision procedure for the necessary computations, e.g. the Stanford Validity Checker SVC [L98]. In general, especially for applications written in unrestricted C, a manual process to define the abstractions seems unavoidable. This leads to the next method, called *tabled abstraction*.

6.1. Tabled abstraction

Once a program is parsed, all control-flow constructs have been interpreted and what remains are only the basic statements and conditions from the source language. We can sort this list, remove duplicates, and place the entries into a table. For each entry into the table we can now define an translation from the source language to the target modeling language. The translation allows us to specify simple syntactical conversions but also higher-level abstractions. The table can be filled in to a large extent with automated techniques, e.g. slicing and predicate abstractions based on the property to be proven. It seems unavoidable, though, that some of the abstractions that are currently beyond the reach of automated techniques have to be provided manually.

This tabled abstraction method has the advantage that it is intuitive, and imposes minimal overhead on the verifier (both the human and the mechanized versions). It allows us to apply *all* abstraction techniques in our toolset, including manually chosen generalization and restriction techniques. It is relatively easy to keep an abstraction table up to date, as the source program that is the subject of verification evolves. A model extractor can track the evolving source mostly automatically, alerting the user only to changes that cannot be handled mechanically (e.g., extensions of functionality in the source).

6.2. Abstraction rules

Each entry into the abstraction table contains a left-hand side entry with a canonicalized representation of a basic statement or conditional expression from the source text of the application, and a right-hand side that specifies its desired interpretation in the abstract model. In many cases, a pre-defined interpretation, or *mapping*, can be applied by the model extractor. Simple predefined types of rules for either hiding or literally preserving specific types of statements from the program source are listed in Table 2.

Table 2 — Predefined Mappings.

Type	Meaning
print	Embed source statement into a print action in the model
comment	Include in the model as a comment only
hide	Do not represent in the model
keep	Preserve in the model, subject to global Substitute rules

A mapping to print, for instance, signifies that we can abstract from the source statement, but that we still are interested in seeing a witness of its appearance in the run of a model (e.g., in simulation runs or when reproducing error trails). A mapping to comment preserves the source text of the statement as a comment in the model, but without any semantics. A mapping to hide strips the statement completely from the model.

An example of an abstraction table with three of these mapping rules, plus two global Substitute rules, is shown as Table 3.

Table 3 — A Sample Abstraction Table.

Substitute FALSE	false
Substitute BOOL	bit
D: int pData=GetDataPointer();	hide
D: BOOL m_bConnected	keep
A: *((int *)pData)=(int)nStatus	print
A: m_bConnected=FALSE	keep

Declarations from the source text are prefixed (by the model extractor) with a designation "D:" and assignments are prefixed with "A:". Assume that it can be determined that the use of variable pData is irrelevant to the property to be proven. We suppress the variable declaration in the verification model with a mapping to hide, but can nonetheless preserve visibility of access to the variable by mapping all assignments to print. The print mapping means that whenever this statement is encountered the verification model will not execute but print the source text of the statement.

If a particular statement does not appear in the abstraction table the model extractor applies a default mapping rule, which can be chosen by the user. For assignments, the default rule could be print, and in that case the above entry can be omitted from the abstraction table. The user can specify a default mapping for each basic type of statement (e.g., declarations, assignments, function calls, conditions).

All branch conditions, e.g. those used in iteration and selection statements to effect control flow, are entered twice into the abstraction table by the model extractor: once in the form found in the source text, and once in negated form. The reason for this apparent redundancy is that in the abstract model we have the option of mapping *both* versions to true, and thus introduce non-determinism. Consider, for instance, the following case:

```
C: (device_busy(x->line))     true
C: !(device_busy(x->line))    true
```

The precise determination if a given device is idle or busy is considered to be beyond the scope of the verification here. For verification purposes it suffices to state that both cases can occur, and the results of the verification should hold no matter what the outcome of the call is. In a similar vain, though, we can use a mapping to false as a constraint, to restrict the verification attempt to just one case:

```
F: (device_busy(x->line))     true
F: !(device_busy(x->line))    false
```

Here the verification would check correct operation of the system when the device polled is always busy.

6.3. Explicit mapping

In some cases, the predefined interpretations from Table 2 are not adequate to cover the specifics of a verification. For the applications of model extraction that we have considered so far, this applied to fewer than 20% of the entries in an abstraction table. The following example illustrates a typical use.

```
F: m_pMon->SendEvent(dest_Id,etype)     destq!etype
```

Here the sending of a message is preserved in the verification model, much like a keep, after by casting it into a specific, standardized, format. Note that within a programming language the send statement can take

any form whatsoever, since there is no generally accepted standard library for such operations. The abstraction table here serves to standardize the format for these types of statements, without impeding the freedom of the programmer to chose an arbitrary representation.

How a particular program statement should be abstracted in the model can also depend on the data objects that are used in that statement. The tabled abstraction method allows us to identify the data objects that should be considered relevant to the verification and those that can be elided without harm. A statement that refers to an irrelevant data object will then be hidden from the verification model.

If no explicit mapping is defined and no data restrictions apply, then the model extractor will apply a set of default type rules to define the conversion from program to model. Each source fragment is classified as one of four types: an assignment (A), a condition (C), a declaration (D), or a function call (F). For each of these types the model extractor has a default abstraction rule, based on the entries from Table 2.

6.4. Abbreviations

The abstraction table is generally much smaller than the program text from which it is derived. The user can shorten it still further by exploiting some features of the model extractor. First, any entry that maintains it's default mapping can be omitted from a user-maintained table: the model extractor can fill in these missing entries as needed. Second, the user can use patterns to assign the same mapping to larger groups of entries that match the patter. For instance, suppose that all calls of the C library-functions `memcpy` and `strcpy` are to be hidden. We can avoid having to list all different calls by using ellipses, as follows:

```
F: memcpy(...    hide
F: strcpy(...    hide
```

This method could be expanded into a more general pattern matching method based on regular expressions. The above prefix match, however, suffices to cover most cases encountered in practice.

The second method for introducing abbreviations uses the Substitute rule that was shown earlier. Substitute rules take effect only on mappings of type keep, and they are applied in the order in which they are defined in the abstraction table.

6.5. Example

The tabled abstraction method was first described in [HS99a], [HS99b] and used at Bell Labs to prove the correctness of the call processing software for a new commercial switching system. We'll illustrate the use of the tabled abstraction method here with a much smaller example: an implementation in ANSI-C [KR88] of the well-known alternating bit protocol from [BSW69]. The source text for this program is shown below.

```
#include <stdio.h>

/*
 * C version of alternating bit protocol
 */

typedef char uchar;

typedef struct Buffer {
         int size;         /* current size of buffer */
         uchar *cont;      /* buffer contents */
} Buffer;

extern int get_data(Buffer *);
extern int put_data(Buffer *);
```

```
int
abp_sender(int N)
{       Buffer   Bufinp, Bufout;
        short    s, S=0, cnt=0;

        Bufout.size = 1;
        Bufout.cont = "M";
        while (get_data(&Bufout))
        {       cnt++;
                send(&Bufout, S);
                if (!recv(&Bufinp, &s))
                        break;
                if (s == S)
                        S = 1 - S;
        }
        return cnt;
}

int
abp_receiver(void)
{       Buffer  Bufinp, Bufout;
        short    s, E=0, cnt=0;

        Bufout.size = 1;
        Bufout.cont = "A";
        while (recv(&Bufinp, &s))
        {       cnt++;
                send(&Bufout, s);
                if (s == E)
                {       E = 1 - E;
                        if (!put_data(&Bufinp))
                                break;
        }       }
        return cnt;
}
```

The program defines the behavior of the sender and the receiver in the protocol. To run it, one can instantiate two independent processes (asynchronous threads of execution): one process to execute the sender's code and one process to execute the receiver's code. Two external routines are assumed to be available in the execution environment. The function `get_data()` is used at the sender side to obtain data to be transmitted, and the function `put_data()` is used to deliver data to its ultimate destination at the receiver. The details of the code are of less interest here than the process of converting it into an abstract model, guided by a user defined abstraction table.

Using the program as input, we can extract a verification model in **Spin**'s input language (Promela) with a model extraction tool. The tool we use is the Bell Labs Automata Extractor for C code called `AX`. The tool can generate a default abstraction table, that can be based on slicing and predicate abstraction techniques. The abstraction is conservative in the sense that language constructs that cannot be handled are generalized. For instance conditional tests on data objects that cannot be represented in the specification language of the model checker (e.g., pointers) are non-deterministically mapped to the values *true* and *false*. The table can be adjusted manually for more targeted model extraction.

The two parts of the model of the alternating bit protocol, one part for the sender and one part for the receiver, are extracted separately as follows.

```
$ ax -a abp_receiver abp.c
$ ax -a abp_sender  abp.c
```

The two parts of the model are extracted into the files `abp_receiver.spn` and `abp_sender.spn`, and the two default abstraction tables are written into the files `abp_receiver.lut` and `abp_sender.lut`. The tables we will use are shown in Figures 18 and 19.

The model extractor classifies statements as a declaration (prefix "D:"), a condition (prefix "C:"), an assignment (prefix "A:"), a function call (prefix "F:"), a return statement (prefix "R:"), or an expression (prefix

```
D: Buffer Bufinp,Bufout;   keep              /* literal */
D: short s,E=0,cnt=0;      keep
A: Bufout.size=1           keep
A: Bufout.cont="A"         keep
C: (s==E)                  keep
C: !(s==E)                 keep
A: E=(1-E)                 keep
E: cnt++                   keep

F: send(&(Bufout),s)       sq!Bufout,s       /* syntax conversion */
C: recv(&(Bufinp),&(s))    rq?Bufinp,s       /* syntax conversion */
C: !recv(&(Bufinp),&(s))   timeout           /* restriction */

C: (!put_data(&(Bufinp)))  false             /* restriction */
C: (put_data(&(Bufinp)))   print             /* slicing */
R: return cnt              hide              /* slicing */
```

Fig. 18 — *Abstraction Rules for Receiver.*

```
D: Buffer Bufinp,Bufout;   keep              /* literal */
D: short s,S=0,cnt=0;      keep
C: (s==S)                  keep
C: !(s==S)                 keep
A: S=(1-S)                 keep
E: cnt++                   keep

F: send(&(Bufout),S)       rq!Bufout,S       /* syntax conversion */
F: recv(&(Bufinp),&(s))    sq?Bufinp,s       /* syntax conversion */
F: !recv(&(Bufinp),&(s))   timeout           /* restriction */

C: (!get_data(&(Bufout)))  false             /* restriction */
C: (get_data(&(Bufout)))   print             /* slicing */
R: return cnt              hide              /* slicing */
```

Fig. 19 — *Abstraction Rules for Sender.*

"E:").

The main restriction we have defined for this verification attempt is the assumption that the `put_data` and `get_data` functions do not fail. We can abstract from the effect of these functions for almost all properties of interest of this protocol (the working of the protocol depends only on the sequence numbers, but not on the actual data that is being transferred). Because the `get_data` is assumed to always succeed, we restrict to the case where there is an infinite stream of messages from sender to receiver, tempered only by the flow of acknowledgements in the opposite direction. The `recv()` statements, therefore, also cannot fail. We have mapped these to `timeout`, but under the given assumptions they could of course also have been mapped to *false*.

The model extraction based on these tables produces the following result. The two parts of the abstract **Spin** model shown in Figure 20 and 21 are generated by the model extractor. Once a property is added to this model, slicing techniques could eliminate still more statements, e.g., the manipulation of the `cnt` variables in sender and receiver, and the access to the `Bufout` data structure in the receiver. The model is fairly close to the one that one would construct manually based on the description in [BSW69].

If the source program is revised for any reason, we can reuse the abstraction tables from above to re-extract a model from the modified code. If new statements were introduced, the model extractor will add default entries for them in the abstraction table and warn the user about their presence, so that they can be adjusted to conform to the abstraction focus that was chosen. If statements were omitted, the model extractor will comment them out of the abstraction table. For even significant revisions of the source, taking days for a programmer to make, an update of the abstraction table to match it to the new version of the code typically takes no more than a few minutes. The alternative of rebuilding a complete verification model for each new version of the source program by hand would more likely approach the investment of time that the programmer made.

```
active proctype abp_receiver()
{	Buffer Bufinp,Bufout;
	short s, E=0, cnt=0;

	Bufout.size = 1
	Bufout.cont = A;

	do
	:: rq?Bufinp,s ->
		cnt++;
		sq!Bufout,s;
		if
		:: (s==E) ->
			E = (1-E);
			printf("C: !(!put_data(&(Bufinp)))0)
		:: !(s==E)
		fi
	:: timeout -> break
	od
}
```

Fig. 20 — *Abstract Model for the Receiver.*

```
active proctype abp_sender()
{	Buffer Bufinp, Bufout;
	short s, S=0, cnt=0;

	do
	::  printf("C: !(!get_data(&(Bufout)))0);
		cnt++;
		rq!Bufout,S;
		if
		:: sq?Bufinp,s
		:: timeout -> break
		fi;
		if
		:: s==S -> S=(1-S)
		:: else
		fi
	od
}
```

Fig. 21 — *Abstract Model for the Sender.*

We can inspect the behavior of the abstracted implementation with **Spin**. First we join the two parts of the model in a simple Promela wrapper that defines minimal context for the two processes. The wrapper below defines two abstract channels via which the processes can exchange their messages, and includes the text of the two processes. The text of this wrapper, stored in a file called `abp`, is shown in Figure 22.

```
mtype = { A, M }; /* acknowledgements and data messages */

typedef Buffer {
	int size;		/* size of buffer */
	mtype data;		/* abstracted buffer contents */
};

chan rq = [2] of { Buffer, bit };	/* data and sequence number */
chan sq = [2] of { Buffer, bit };

#include "abp_receiver.spn"
#include "abp_sender.spn"
```

Fig. 22 — *Context Definition for Alternating Bit Protocol.*

Now we can run **Spin** on this model. First, we can look at the first 20 steps in a simulation run, looking

only at message exchanges:

```
$ spin -c abp | sed 20q
proc 0 = abp_receiver
proc 1 = abp_sender
                                    C: !(!get_data(&(Bufout)))
q\p   0   1
  1   .   rq!0
  1   rq?0
  2   sq!0
  2   .   sq?0
                                    C: !(!put_data(&(Bufinp)))
                                    C: !(!get_data(&(Bufout)))
  1   .   rq!1
  1   rq?1
  2   sq!1
  2   .   sq?1
                                    C: !(!put_data(&(Bufinp)))
                                    C: !(!get_data(&(Bufout)))
  1   .   rq!0
  1   rq?0
  2   sq!0
                                    C: !(!put_data(&(Bufinp)))
```

This shows the two processes exchanging the sequence numbers and correctly retrieving and depositing data during the run. A verification run can be more illuminating, checking the system for possible deadlocks, and answering any other logical query that the user can formulate about the operation of the system.

```
$ spin -a abp
$ cc -o pan pan.c
$ pan
(Spin Version 3.4.0 -- 15 August 2000)
         + Partial Order Reduction

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

State-vector 36 byte, depth reached 13, errors: 0
      14 states, stored
       2 states, matched
      16 transitions (= stored+matched)
       0 atomic steps

hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493    memory usage (Mbyte)

unreached in proctype abp_receiver
         (0 of 12 states)
unreached in proctype abp_sender
         (0 of 12 states)
```

The verification run confirms two simple default properties of this *implementation* of the alternating bit protocol, under the stated restrictions: absence of deadlock and absence of unreachable code. Does this prove the protocol correct? No it does not. To prove that the implemented version transfers messages without loss and without reordering requires us to state and prove these more specific properties. Suffice it to note here that the implemented version of the protocol discussed her deviates in a subtle way from the original proposed in [BSW69], and does not have any of these desirable properties.

6.6. Industrial application

Automated model extraction from source code based on slicing, predicate abstraction, and the tabled abstraction method were applied successfully in at least one significant industrial project to date: the design of the call processing software for a new telephone switching system at Lucent Technologies. A detailed description of this project can be found in [HS00].

7. In conclusion

The techniques that are used in practice today to secure the quality of software were developed in the late sixties and early seventies and have changed little since then. This is a rather remarkable phenomenon. Within the same period software applications have changed significantly in size and complexity. The largest applications of the early seventies would be considered relatively small if produced today. For example, an early version of Unix® from 1973 counted just 6,600 lines of C. Today even a wordprocessing application is orders of magnitude larger, and, for that matter, the source code for the model checker **Spin** is about three times larger too. Similarly, in the early seventies most applications executed standalone and sequentially, while most applications today execute in a distributed environment. To test them fully one would need to consider sets of related and possibly interacting threads of execution.

Despite all these changes, and despite valid critique about the fundamental flaws of the traditional approach to testing, these methods are relatively effective. The best testament of this is that even though almost all computer controlled devices and services of today were checked with only these techniques, overall they do work as advertised. The phone system, for instance, is designed with these techniques to meet exceptionally stringent reliability requirements (less than 3 minutes downtime per year per switch).

On the other hand, there is also a slowly growing number of examples of spectacular failures of software controlled systems. The examples are known well enough that we need not to repeat them here. (And the odds are that better examples will occur between the writing of these notes and the time that you read them.)

The reality of industrial software development is that today it is not economically feasible to develop fault-free products. Software testing continues only until the *rate of discovery* for new software defects drops below a preset level. At this point, continuing testing becomes increasingly ineffective. The effect is illustrated in Figure 23.

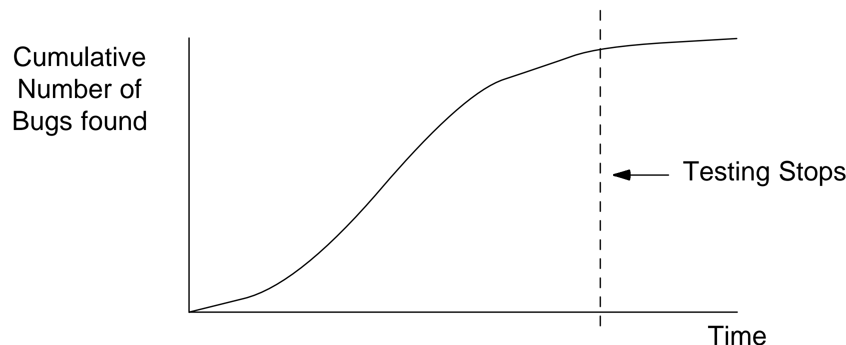


Fig. 23 — *The S-Curve of Test Effectiveness.*

After an initial startup period, where relatively few bugs are found, the testing process starts uncovering errors at a rate that is proportional to the number of tests performed. At some point the rate at which new problems are discovered drops. The most likely bugs that are within the range of the tests have now been found. Even if the amount of time spent on testing would be doubled, the number of problems found would increase only marginally. Hence, it is no longer cost-effective to continue this process. The bugs with a lower probability of occurrence, in the given test suite at least, will remain either dormant or they will be repaired only when a customer steps on one and reports the problem. We can express *risk* as the product of the probability of occurrence of an error and the damage that can be caused by that occurrence. Clearly, not all undiscovered software defects carry the same level of risk. Figure 24 illustrates this.

Traditional testing techniques cover areas 1 and 3 in Figure 24 well: they find the most likely errors in a

software application. Discovering the errors in areas 3 and 4 is critically important to software quality, while errors in areas 1 and 2 are of little practical interest. Areas 1 and 3 are important for the first impression of quality by the users of a software product. The errors in area 4, however, contribute to the infrequent and sometimes spectacular failures. When software is used infrequently, by a small group of users, the likelihood of these types of errors occurring remains small. The probability goes up, though, for successful products that are used frequently by large numbers of users, which is a relatively recent phenomenon. Traditional testing techniques cannot hope to reach these types of error. But they can reliably be found with formal software verification techniques of the type we have described in Part IV of these notes. Model checkers such as **Spin** do not distinguish between likely and unlikely scenarios, they consider all *possible* scenarios. Thus, they perhaps are still over-qualified for the job. This could be addressed by developing a new class of reduction techniques that can focus the attention of the model checker exclusively on area 4 in Figure 24, considering that the other areas are either uninteresting or are already sufficiently covered by traditional techniques.

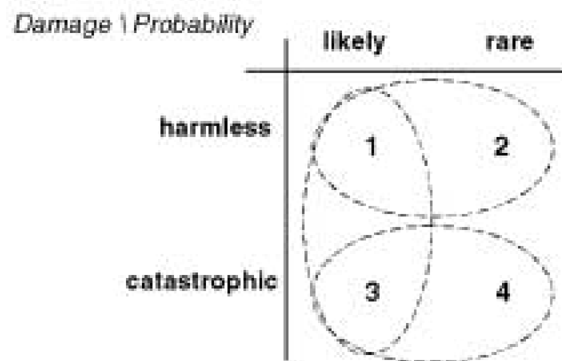


Fig. 24 — *Risk and Damage.*

Changing paradigms

It is perhaps interesting to note that the method we have outlined in these notes, based on the mechanical extraction of automata models from software implementations, is the reverse of the theoretically more attractive method of top-down stepwise refinement of code, proceeding from an abstract model towards a concrete implementation. The latter technique, based on prevention rather than detection, is easier to justify, but has clearly resisted practical adoption so far. The method outlined here proposes a more distant approach that imposes no new constraints on the software development process, but merely enables the designer to detect efficiently when design objectives are jeopardized.

References

[AL91] Abadi, M., Lamport, L., The existence of refinement mappings. *Theoretical Computer Science*, Vol. 82, No. 2, May 1991, pp. 253-284.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

[ASU86] A.V. Aho, R. Sethi, & J.D. Ullman, *Compilers - principles, techniques, and tools*, Addison-Wesley, 1986, p.671.

[BSW69] 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, 260-265, 1969.

[B70] B.H. Bloom, Spacetime trade-offs in hash coding with allowable errors. *Communications of the*

ACM, 13(7), July 1970, pp. 422--426.

[BZ83] D. Brand, and P. Zafiropulo, On communicating finite state machines, *Journal of the ACM*, Vol. 30, No. 2, pp. 323-342.

[CW00] M. Chechik and W. Ding, Lightweight reasoning about program correctness, CSRG Technical Report 396, University of Toronto, March 2000.

[CP99] Chint-Tsun Chou and D. Peled, Formal verification of a partial order reduction technique for model checking, *Automated Reasoning*, Vol. 23, No. 3, Nov. 1999, pp. 265-298.

[C74] Y. Choueka, Theories of automata on $omega#-tapes: a simplified approach, *Journal of Computer and System Science*, Vol. 8, 1974, pp. 117-141.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long, Model checking and abstraction. *ACM-TOPLAS*, Vol. 16, No. 5, Sept. 1994, pp. 1512-1542.

[CD00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, et al., Bandera: Extracting finite-state models from Java source code, *Proc. Int. Conf. on Software Engineering*, Limerick, Ireland, June 2000, to appear.

[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. I, 1992, pp. 275-288. First published in June 1990 in *Proc. 2nd Conference on Computer Aided Verification*, Rutgers University, New Jersey.

[CC76] P. Cousot and R. Cousot, Static determination of dynamic properties of programs, *Proc. Colloque sur la Programmation*, April 1976.

[CM81] P.R.F. Cunha, and T.S.E. Maibaum, A synchronization calculus for message oriented programming, *Proc. Int. Conf. on Distributed Systems*, IEEE, 1981, pp. 433-445.

[DGG97] D. Dams, R. Gerth, and O. Grumberg, Abstract interpretation of reactive systems, *ACM Trans. on Programming Languages and Systems*, Vol. 2, No. 19, pp. 253-291, March 1997.

[DGV99] M. Daniele, F. Giunchiglia, and M.Y. Vardi, Improved automata generation for linear temporal logic. *Proc. 11th Int. Conf. on Computer Aided Verification*, LNCS 1633, pp. 249-260, 1999.

[DDP99] S. Das and D.L. Dill, and S. Park, Experience with Predicate Abstraction, *Conf. on Computer-Aided Verification*, Trento, Italy, 1999, Springer Verlag.

[EH00] K. Etessami, and G.J. Holzmann, Optimizing B$roman u dotdot#chi Automata, *Proc. CONCUR 2000*, to appear.

[E90] E.A. Emerson, Temporal and modal logic, *Handbook on Theoretical Computer Science*, Volume B, Elsevier Science, 1990, pp. 995-1072.

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

[GH93] P. Godefroid and G.J. Holzmann, On the verification of temporal properties, *Proc. Int. Conf on Protocol Specification, Testing, and Verification*, Liege, Belgium, May, 1993, pp. 109-124.

[GS97] S. Graf, H. Saidi, Construction of abstract state graphs with PVS. In: O. Grumberg, Ed., *Conf. on Computer Aided Verification*, Haifa, Israel, Springer Verlag, LNCS 1254, pp. 72-83. 1997.

[H81] G.J. Holzmann, Pan — A protocol specification analyzer, AT&T Bell Laboratories Technical Memorandum, TM81-11271-5, 1981.

[H88] G.J. Holzmann, An improved reachability analysis technique, *Software Practice and Experience*, Vol. 18, No. 2, pp. 137-161, Feb. 1988. An early version appeared in *Proc. Int. Symposium on Protocol Specification, Testing, and Verification*, Zurich, Switzerland, North-Holland Publ., Amsterdam, 1987, pp. 339-344.

[H90] G.J. Holzmann, **Spin** — A protocol analyzer, *Unix Research System*, Tenth Edition, Volume II, Papers, Saunders College Publ., pp. 423-429. January 1990.

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

NJ, 1991.

[HP94] G.J. Holzmann and D. Peled, An improvement in formal verification, *Proc. 7th Int. Conf. on Formal Description Techniques*, FORTE94, Berne, Switzerland. October 1994.

[HPY96] G.J. Holzmann, D. Peled, and M. Yannakakis, On nested depth-first search, *Proc. 2nd Spin Workshop*, Rutgers Univ., New Brunswick, New Jersey, August 1996, American Mathematical Society, DIMACS/32, 1996.

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

[H98] G.J. Holzmann, An analysis of bitstate hashing, *Formal methods in system design*, Vol. 13, No. 3, Nov. 1998, pp. 287-307.

[H98b] G.J. Holzmann, Designing executable abstractions, *Proc. Formal Methods in Software Practice*, Clearwater Beach, Fl., ACM Press, 1998.

[H99] G.J. Holzmann The engineering of a model checker. *Proc. 6th Spin Workshop*, Toulouse, France, Sept. 1999, Springer Verlag, LNCS 1680.

[HS99a] G.J. Holzmann, and M.H. Smith, A practical method for the verification of event driven systems. *Proc. Int. Conf. on Software Engineering*, Los Angeles, May 1999, pp. 597-608.

[HS99b] G.J. Holzmann, and M.H. Smith, Software model checking: extracting verification models from source code. *Formal Methods for Protocol Engineering and Distributed Systems*, Kluwer Publ., London, Oct. 1999, pp. 481-497.

[HS00] G.J. Holzmann, and M.H. Smith, Automating software feature verification. *Bell Labs Technical Journal*, Vol. 5, No. 2, April-June 2000, pp. 72-87.

[K68] J.A.W. Kamp, *Tense Logic and the Theory of Linear Order*, Ph.D. thesis, University of California at Los Angeles, 1968.

[KR88] B.W. Kernighan, and D.M. Ritchie, *The C Programming Language, 2nd Edition*, Prentice Hall, Englewood Cliffs, N.J., 1988.

[Kw89] M. Kwiatkowska, ‘‘Event fairness and non-interleaving concurrency,’’ *Formal Aspects of Computing*, 1989, Vol. 1, pp. 213--228.

[L83] L. Lamport, What good is temporal logic, in: R.E.A. Mason, ed., Information Processing 1983: *Proc. of the IFIP 9th World Computer Congress*, Paris, France, North-Holland Pub., Amsterdam, 1983, pp. 657-668.

[L98] J.R. Levitt, *Formal verification techniques for digital systems*, PhD Thesis, Stanford University, Stanford, CA., Dec. 1998.

[MP91] Z. Manna, and A. Pnueli, *The temporal logic of reactive and concurrent systems: Specification*, Springer-Verlag, 1991.

[Ma87] A. Mazurkiewicz, Trace Theory, In: *Advances in Petri Nets*, 1986, LNCS, Vol. 255, Springer Verlag 1987, pp. 279-324.

[M68] R. Morris, Scatter storage techniques, *Comm. of the ACM*, Vol 11, No. 1, Jan. 1968, pp. 38-44.

[P94] D. Peled, Combining partial order reductions with on-the-fly model checking, *Proc. 6th Int. Conf. on Computer Aided Verification*, Stanford, Ca., June 1994.

[P96] D. Peled, Combining Partial Order Reductions with On-the-fly Model-Checking, *Journal of Formal Methods in Systems Design*, Vol. 8, No. 1, 1996, pp. 39-64.

[P97] D. Peled, and T. Wilke, Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 1997, 63:5, pp. 243-246.

[P77] A. Pnueli, The temporal logic of programs. *Proc. 18th IEEE Symposium on Foundations of Computer Science*, 1977, Providence, R.I., pp. 46-57.

[P57] A.N. Prior, *Time and Modality*, Oxford: Clarendon Press, 1957.

[P67] A.N. Prior, *Past, Present, and Future*, Oxford: Clarendon Press, 1967.

[RU71] N. Rescher, and A. Urquhart, Temporal Logic, 1971, Springer Verlag, Library of Exact Philosophy, ISBN 0-387-80995-3, 273 pgs.

[SB00] F. Somenzi, and R. Bloem, Efficient B$roman u dotdot#chi-automata from LTL formula, *Proc. 11th Int. Conf. on Computer Aided Verification*, 2000, to appear.

[T72] R.E. Tarjan, Depth first search and linear graph algorithms,'' *SIAM J. Computing*, 1:2, pp. 146-160, 1972.

[T90] W. Thomas, Automata on infinite words. *Handbook on Theoretical Computer Science*, Volume B, Elsevier Science, 1990, pp. 135-165.

[T95] F. Tip, A survey of program slicing techniques. *Journal of Programming Languages*, Vol. 3, No. 3, Sept. 1995, pp. 121-189.

[T36] A.M. Turing, On computable numbers, with an application to the Entscheidungs problem. *Proc. London Mathematical Soc.*, Ser. 2-42, pp. 230-265 (see p. 247), 1936.

[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.

[V96] M.Y. Vardi, An automata-theoretic approach to linear temporal logic. In: *Logics for Concurrency: Structure versus Automata*, pp. 238-265. Springer Verlag, LNCS 1043, 1996.

[V00] W. Visser, S. Park, and J. Penix, Applying predicate abstraction to model checking object-oriented programs. *Proc. 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, August 2000.

[W93] P. Wolper, D. Leroy, Reliable hashing without collision detection, *Proc. 5th Int. Conf. on Computer Aided Verification*, Elounda, Greece, Springer Verlag, LNCS 697, pp. 59-70.