

Software Model Checking with SPIN

Gerard J. Holzmann

Laboratory for Reliable Software
NASA/JPL, Pasadena, CA 91109, USA

Abstract

The aim of this chapter is to give an overview of the theoretical foundation and the practical application of logic model checking techniques for the verification of multi-threaded *software* (rather than hardware) systems. The treatment is focused on the logic model checker SPIN, which was designed for this specific domain of application. SPIN implements an automata-theoretic method of verification. Although the tool has been available for over 15 years, it continues to evolve, adopting new optimization strategies from time to time to help it tackle larger verification problems. This chapter explains how the tool works, and which types of software verification problems it is designed to handle.

Contents

1.	Introduction	2
2.	Background	3
3.	Finite Automata	5
	Automaton Runs	7
	Omega Acceptance	7
	Asynchronous Product	8
	Automata Expansion	8
4.	Temporal Logic	9
	Standard Formulae	10
	Synchronous Product	11
5.	LTL Model Checking	13
	Depth-First Search	13
	Nested Depth-First Search	14
	Adding Fairness	18
	SPIN's On-the-Fly Implementation	19
	Partial Order Reduction	19
	Compression Techniques	19
	Bitstate Hashing	20
6.	Model Extraction and Abstraction	21
	Other Uses of Abstraction	23
7.	Perspective	23
8.	References	23

1. Introduction

This chapter is concerned with the development of automated procedures for the verification of software systems, with particular emphasis on the verification of process interactions in either logically or physically distributed software systems. Several verification tools are available today that can prove interesting facts for a significant class of such systems. An up to date overview can be found on the web.¹ In this description we will focus on SPIN [36] as one of the leading tools in this class. SPIN is distributed freely in source form.²

Two notable trends have contributed to the recent successes of the logic model checkers in the verification of distributed software systems. The first trend is the continuing improvement in algorithms and tool design in this area, which make it possible to handle ever larger and more complex verification problems. We will review the main improvements of this type in this chapter. A second significant trend is the steady increase in basic compute power, which continues to follow the curve that was tentatively suggested by Gordon Moore nearly forty years ago [42].

The trends that have turned software verification from a theoretical curiosity into a practical reality are paralleled by similar trends in hardware verification. The difference in the nature of hardware and software, though, makes that there is surprisingly little overlap in the algorithms, data structures, and specific logics that are used in these two fields. We will discuss some of the main reasons for these differences towards the end of this chapter.

The most commonly used method to validate software systems today remains testing. In a unit test, a single process or module of the system is placed in isolation and probed on its functional correctness. Once successful, a series of unit tests is followed by a system integration test. In an integration test multiple units are linked together to form part or all of the envisioned system. The limitations of this method of system validation are as well known as its benefits. For sequential software systems, where one is primarily interested in verifying the computational aspects of a system, the classical testing techniques still have few competitors, even though much could be done to improve precision and coverage by a more aggressive use of formal methods based approaches. In distributed software systems, the verification task is larger, since now we do not just need to worry about computational correctness but also about a range of concurrency related problems that can prevent proper execution. Concurrency does not just increase the obligations of the tester or verifier, it also significantly complicates the already existing obligations for demonstrating the correctness of sequential computations. Concurrency can introduce race conditions, data corruption, delay, process or thread starvation, or even system-wide deadlock.

The unpredictable nature of the interleaving of process executions in distributed systems makes that test executions are not always reproducible. Each single execution is typically only one of a virtually unimaginably large set of possible interleaved executions. What is needed to address these problems is an effective method for probing the system for conveniently defined *classes of behavior*, rather than isolated *instances of behavior*. Logic model checkers promise to provide such a technique, but they too comes with some limitations. The current limitations of model checking are of two kinds: computational

1. <http://archive.comlab.ox.ac.uk/comp/formal-methods.html>

2. <http://spinroot.com/whatispin.html>

complexity and user friendliness. In this chapter we provide a synopsis of the model checking procedure as it applies to the verification of distribute software systems, and summarize the progress that has been made in diminishing the effects of these last two limitations.

We will begin by sketching the development of automated verification systems since the late seventies. We then introduce the main building blocks of the software model checking procedure. We define what a formal model is, and how we can formally state the logic properties of a model, using a standard definition of automata on infinite words. Next we discuss the automata theoretic verification procedure, as defined by Vardi and Wolper [52], and show how it can be implemented efficiently, following the methodology that was used in the design of the SPIN model checker.

We then move to some more practical considerations: the problem of semi-automatically extracting models from software applications, to enable us to apply the model checking procedure. Systematic abstraction techniques form a central role in this area. We discuss abstraction techniques and give examples of significant applications of software model checking in practice. We conclude the chapter with a brief comparison of software model checking with techniques used in hardware verification and a perspective of likely developments in this field in the near future.

2. Background

The first attempts to built automated verification systems targeted formal definitions of communication protocols. In 1979, Jan Hajek used a graph exploration tool, called Approver [21], to formally verify basic correctness properties of the protocols in Tanenbaum's primer on computer networks [47]. The details of Hajek's system were documented only fairly recently,³ but remained unknown at the time. Unfortunately, this limited the influence that the Approver system could have on later developments. Independently, Colin West developed a different protocol validation procedure [53], triggered by a collaboration with Pitro Zafiropulo on the validation of a CCITT recommendation [54]. Initially unaware of this earlier work, I developed and implemented yet another protocol verification procedure in 1980 at Bell Labs, initially based on the definition of a process algebra [22,23].

Communications protocols were a convenient target for the early work since their behavior could fairly easily be formalized in terms of automata. A well-known example of this type, that served as a litmus test for early validation systems throughout the eighties, is the definition of the alternating bit protocol as two communicating finite state machines. The protocol is illustrated in Figure 1 as it was first defined in 1969 [3].

3. <http://www.mattheory.info/hajekit/approver.txt>

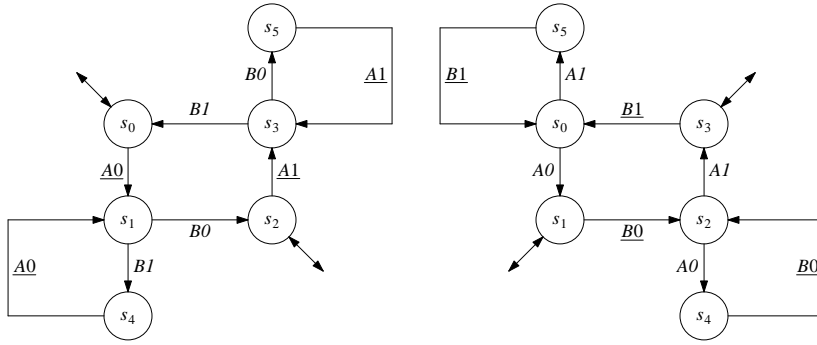


Figure 1 — Alternating Bit Protocol

Two state machines are defined in Figure 1, formalizing a sender process and a receiver process. The edge labels specify message exchanges. Each label consists of two characters: the first specifies the origin of the message being sent or received and the second specifies a sequence number for the message. This sequence number, termed the *alternation bit* in [3], is either zero or one, and is toggled between the two values on each successful transmission. Underlined names represent send actions; the other names represent receives. The double arrows, finally, indicate the states where new data is to be fetched for transmission by the sender, or received data is to be stored by the receiver. The protocol starts with sender and receiver in the states labeled s_0 . The sender will then transmit a message with sequence number zero ($\underline{A0}$) to the receiver. If all is well, the receiver will receive the message ($A0$) and both processes will move to the states labeled s_1 . The receiver will now acknowledge receipt by transmitting $\underline{B0}$, and both processes move to the states labeled s_2 . The same sequence now repeats with the sequence number toggled from zero to one. If for some reason, e.g., the loss or duplication of a message, the receiver process sees a data message with the wrong sequence number, it will reply with the matching acknowledgment but not proceed. If the sender receives an acknowledgment with a wrong sequence number it will repeat the last transmission and hope for the best.

If we abstract from the data being transmitted, we can see that each process in this system can be in no more than six distinct states. The combination of sender and receiver, therefore, can be in no more than 6×6 or 36 distinct system states. In this simple case, a brute force exhaustive enumeration of all reachable states of the system will suffice to establish most of its logical properties. The combined behavior of the system defines a new finite state automaton, and can similarly to Figure 1 be formalized as a graph. This graph can be constructed and analyzed with a standard depth-first search procedure [48] at a cost that is linear in the size of the graph.

Manual techniques for the analysis of state machine models of protocols had been pioneered in the early seventies, e.g. [5], but not surprisingly these methods quickly proved too cumbersome and too errorprone, as demonstrated in [21]. Although the first automated systems had greater potential, they were mostly restricted to proving only a small set of mostly predefined properties, and would quickly run into seemingly unsurmountable barriers of computational complexity. The types of properties that could be

demonstrated for small protocol models included absence of deadlock (i.e., the absence of reachable states in the global execution graph without successors) and the preservation of system invariants on system states (i.e., the absence of reachable states in which one or more of the required invariants would evaluate to false). In the eighties a more general framework for proving logic properties of finite state models took shape and found general acceptance.

The development of automated verification systems has taken a somewhat different path for hardware and for software applications, leading to two different sets of verification tools that are based on different logics and that exploit different types of search and optimization algorithms. The dominant techniques in formal hardware verification, e.g. [13,41], are founded on the early work of Clarke and Emerson in the U.S. [11], and of work by Queille and Sifakis in France [45]. In software model checking, the development can be traced through the early work of Pnueli [44] on temporal logic, to the development of the *automata theoretic verification method* by Vardi and Wolper in the mid eighties [52,55]. It should be noted, though, that the new theories were not immediately of practical use. It took a while for algorithms to be developed that could be implemented efficiently, and for desktop machines to provide the required compute power to execute them. It appears now generally agreed that this turning point was reached in the mid to late nineties.

With improved algorithms and ever increasing compute power, the attention in recent years has shifted from the development of the basic capability to perform logic model checking on hand-built system models towards the automated extraction of verification models from implementation level source code. Before discussing these methods, though, we will first cover the basic theoretical framework that underlies specifically the SPIN model checking system.

3. Finite Automata

In this section we introduce the notion of an automaton and of ω -acceptance, which we use to develop the automata theoretic verification method in subsequent sections. We begin with the definition of an extended finite state automaton.

Definition 3.1. An *extended finite state automaton* A is a tuple $\{S, s_0, D, L, T, F\}$, with S a finite set of 'states,' $s_0 \in S$, called the 'initial state,' D a finite set of named 'data objects,' L a finite set of named 'actions' on objects in D , $T \subseteq S \times L \times S$, called the 'transition relation,' and $F \subseteq S$, called the set of 'final' states.

We will be brief here about the definition of 'data objects' and 'actions.' Model checking languages such as PROMELA [27,30,36] give precise semantics to these notions, which guides the operation of the model checker. For our purposes here, it will suffice to assume that each data object has a unique name and finitely many possible 'values' of arbitrary type. One value in the domain of each object is always tagged as the initial value of an object of that type. Each data object also has a 'current value' that can only be changed through the application (or 'execution') of 'actions' from set L .

Definition 3.2. An *action* on set of data objects D consists of two parts: a guard and an effect. The 'guard' is a boolean expression on the values of elements in D . The 'effect' can change the values of elements in D as a function of the current values of all elements.

The intuition is that (the effect part of) an action can only be applied when the guard condition is true. Every transition in the automaton is labeled with an action, which blocks the transition until the guard condition is satisfied and applies the effect when the guard condition is true and the transition is executed.

Definition 3.3. A transition is said to be *executable* if and only if the guard expression from the corresponding action evaluates to *true*, otherwise it is said to be 'unexecutable' or 'blocking.'

We will use this notion of 'executable' and 'unexecutable' actions below in the definition of the 'runs' of a system.

As examples of useful data objects, consider the following PROMELA message channel structures.

```
chan s2r = [1] of { mtype, bit };
chan r2s = [1] of { mtype, bit };
```

According to PROMELA semantics, these channels are initially empty and can each store one message consisting of two typed fields [36]. Some actions from PROMELA on the channel `s2r` are:

```
full(s2r)
s2r!A,0
r2s?B,1
```

The first action has a guard that returns *true* only when the channel currently stores one message and is thereby filled to capacity. The effect part of the first action is *skip*, a null-operation that has no effect. That is: the actions acts as a condition without side-effects. The second action has a guard that returns *true* only when the channel is non-full. Its effect changes the value of the data object `s2r` by appending the message `A, 0`, with `A` of type `mtype`, and `0` of type `bit`. The third action has a guard that returns *true* only when the channel holds a message with the fields `B, 1`; its effect part deletes that message from the channel.

An extended finite state automaton, as defined, can be represented conveniently by a directed graph with the nodes representing states and the edges representing the transition relation T . The edges are labeled with actions from L . A system like this is therefore also known as a 'labeled transition system.' Our aim is to use extended finite state automata to represent process behavior in a distributed system. Set F can be used to mark the normal termination points of a process, or they can be used to mark special acceptance nodes in the graph that can serve to define and check the liveness properties of a system, as we shall describe shortly.

The two state machines in Figure 1 can also be defined as extended finite state automata. The automaton for the sender (on the left side in Figure 1), for instance, can be defined as follows, using the two data objects that we introduced in the example above and PROMELA syntax for the actions

$$\begin{aligned}
 S &= \{s_0, s_1, s_2, s_3, s_4, s_5\}, \\
 s &= s_0, \\
 D &= \{s2r, r2s\}, \\
 L &= \{r2s?B,0, r2s?B,1, s2r!A,0, s2r!A,1\}, \\
 T &= \{(s_0, s2r!A,0, s_1), (s_1, r2s?B,1, s_4), (s_1, r2s?B,0, s_2), (s_4, s2r!A,0, s_1), (s_2, s2r!A,1,
 \end{aligned}$$

$$s_3), (s_3, r2s?B,1, s_0), (s_3, r2s?B,0, s_5), (s_5, s2r!A,1, s_3)\}, \\ F = \{s_0, s_2\}.$$

In general, the finite state automata that we will consider can be non-deterministic, e.g., we allow transitions such that: $(v, a, w) \in T, (v, a, w') \in T$, with $w \neq w'$. Non-determinism is an important mechanism for building an abstract model of a distributed system. It can be used to generalize a model and to remove implementation level detail [32,36].

Automaton Runs

A run $\sigma = t_0, t_1, t_2, \dots, t_k$ of automaton A is a sequence of transitions that satisfies the following conditions:

the source state for t_0 , the first transition in σ , is always s_0 , i.e., the initial state of the automaton,

$$\forall i, 0 \leq i \leq k : t_i \in T,$$

$$\forall i, 0 \leq i < k : t_i \equiv \{ a, b, c \}, \text{ and } t_{i+1} \equiv \{ d, e, f \} \rightarrow c \equiv d.$$

that is, the run defines a path in the graph of A. Note that a 'run' only defines uninterpreted potential executions of a system, it does not take the manipulation of data objects through actions into account just yet. We will distinguish between 'valid' and 'invalid' runs in an expanded finite state automaton shortly.

According to the classic definition of acceptance a *finite* run is said to be *accepted* by A if its final state is in set F, i.e., for run σ with final transition $t_k \equiv \{ a, b, c \}$ if $c \in F$. If set F is used to mark the normal termination points of a process then a run will not be accepted by the automaton unless it terminates at such a marked state.

Omega Acceptance

The classic notion of acceptance given above applies only to finite runs, i.e., to terminating executions. Looking at the automata in Figure 1, though, it is unclear if termination should be considered proper behavior or an error. As long as data is available from the unspecified source, the sender process should continue to transmit it to the receiver. If the protocol terminates, we would like it to terminate in either state s_0 or s_2 , with the last data message properly acknowledged, but it need not terminate at all. We will therefore define a notion of ω -acceptance that can be applied to both the *infinite* and the *finite* runs of an automaton. An infinite run of an automaton is called an ω -run.

Definition 3.4. An ω -run σ is *accepted* by extended finite state automaton A if it contains at least one state from set F infinitely often.

The above notion of acceptance is known as Büchi acceptance [8,50]. For the automata definition we gave for the processes in Figure 1 it would suffice to limit the set of accepting states to one of the two states s_0 and s_2 , since clearly neither can be visited infinitely often unless the other is too. It is also clear that any ω -run for a finite state automaton will have to repeat states, i.e., it will necessarily be cyclic.

We now define the 'stutter-extension' of a finite run to make sure that the rules of ω -acceptance can be applied equally to infinite and finite runs of an automaton.

Definition 3.5. The *stutter-extension* of a finite run σ of finite state automaton A is the ω -run that is derived from σ by appending an infinite number of nil-actions $\{s_k, nil, s_k\}$ to it, where s_k is the final state that is reached in σ , and *nil* is an action with guard *true* and effect *skip*.

Asynchronous Product

The combined behavior of asynchronously executing processes in a distributed system can be formalized as a simple product of automata.

Definition 3.6. The *asynchronous product* of the extended finite automata $A = \{S, s, D, L, T, F\}$ and $B = \{S', s', D', L', T', F'\}$ is another extended finite automaton $\{S'', s'', D'', L'', T'', F''\}$ such that $S'' = S \times S'$, $s'' = (s, s')$, $D'' = D \cup D'$, $L'' = L \cup L'$, $T'' \subseteq S'' \times L'' \times S''$, $F'' = F \times F'$, and $\forall ((n, n'), l, (m, m')) \in T'' : (l \in L \wedge (n, l, m)) \vee (l \in L' \wedge (n', l, m'))$

That is, the states of the asynchronous product define combinations of states in the individual automata, but the edges correspond to the individual transitions of the two automata: the transitions are interleaved.

As an example, the asynchronous product of the two automata from Figure 1 has 6×6 or 36 states. The initial state of that automaton is (s_0, s_0) . Set F has four states. Set D contains two data objects $\{s2r, r2s\}$, and set L contains the eight actions $\{r2s!B,0, r2s!B,1, r2s?B,0, r2s?B,1, s2r!A,0, s2r!A,1, s2r?A,0, s2r?A,1\}$.

Automata Expansion

Clearly the data objects in an extended finite state automaton also carry state information. We can map an extended finite state automaton to a *pure* finite state automaton by moving the state information from set D into set S. In effect, this expansion multiplies set S with the set of values of all data objects. To construct a pure automaton we can replicate each state in S, except the initial state, as many times as there are distinct combinations of values for all data objects in D. For the initial state, the initial value for each data object is used. Each copy of $s \in S$ has a copy of all incoming and outgoing transitions of s in the original automaton.

Next, we can mark the transitions in this new automaton as either *valid* or *invalid*, depending on whether the corresponding action from L is executable in that state. Since data values are now explicit, the validity of each transition can be determined unambiguously.

Let $\mu(n)$ be the valuation of all data objects in state n , i.e., a finite and ordered set of values, and let $\gamma(l, n)$ be the valuation of all data objects in state n *after* the effect part of action l is applied.

Definition 3.7. A transition $\{n, l, m\}$ from the set of transitions of an expanded finite state automaton is *valid* if the guard of action l is *true* for $\mu(n)$, and $\mu(m) \equiv \gamma(l, n)$.

The expansion process of an automaton is completed by first omitting all transitions that are not valid, and next omitting all states that are no longer reachable from the initial state.

For the automata in Figure 1 the full set of actions is, assuming *ideal* full-duplex communication between sender and receiver:

$\underline{A0}$:	$s2r!A, 0$	$A0$:	$s2r?A, 0$
$\underline{A1}$:	$s2r!A, 1$	$A1$:	$s2r?A, 1$
$\underline{B0}$:	$r2s!B, 0$	$B0$:	$r2s?B, 0$
$\underline{B1}$:	$r2s!B, 1$	$B1$:	$r2s?B, 1$

The complete expansion of the asynchronous product of the two automata, after deleting invalid transitions and unreachable states, has eight states, and permits just one ω -run, as illustrated in Figure 2. In the product automaton in Figure 2 we can now easily annotate each state n with its valuation $\mu(n)$, giving the explicit value of each data object.

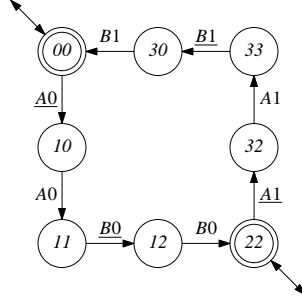


Figure 2 — Expanded asynchronous product of automata in Figure 1. Accepting states have double circles.

4. Temporal Logic

The correctness properties of a distributed system can be formalized in linear temporal logic (LTL), as first proposed by Pnueli in [44]. Any boolean expression over the state of a system and its associated data values will be called a *state formula*. Every guard from a action in an automaton definition, for instance, is defined by a state formula. In the following, the lower case symbols p, q, r represent state formulae and f, g, h represent temporal formulae, which are defined as follows.

Definition 4.1. Every state formula p is also a *temporal formula*.

If f is a temporal formula, then so are $\neg f$, (f) , and Xf .

If f and g are temporal formulae, then so are $f \wedge g$, $f \vee g$, and fUg .

The temporal operator X is pronounced 'next,' and the temporal operator U is pronounced 'until.'

We write $v(f, s_i) \equiv true$ to express that temporal formula f holds at state s_i . We can then define the standard Boolean operators as follows:

$$\begin{aligned} v(f \vee g, s_i) &\Leftrightarrow v(f, s_i) \vee v(g, s_i) \\ v(f \wedge g, s_i) &\Leftrightarrow v(f, s_i) \wedge v(g, s_i) \\ v(\neg f, s_i) &\Leftrightarrow \neg v(f, s_i) \end{aligned}$$

The semantics of X and U are defined over an ω -run σ . Let $s_0, s_1, s_2, \dots, s_i, s_{i+1}, \dots$, be the set of states that is traversed in σ . We can then define:

$$v(Xf, s_i) \Leftrightarrow v(f, s_{i+1})$$

$$v(f\cup g, s_i) \Leftrightarrow v(g, s_i) \vee (v(f, s_i) \wedge v(f\cup g, s_{i+1}))$$

The definition of \cup requires that either g is *true* now or that f remain *true* until g becomes *true*. If, however, f remains *true* invariantly then g is not required to become *true*. The operator \cup is therefore called a ‘weak until’ operator. There is also a ‘strong until’ operator \mathbf{U} , which can be defined as follows.

$$v(f\mathbf{U}g, s_i) \Leftrightarrow v(f\cup g, s_i) \wedge \exists j, j \geq i: v(g, s_j).$$

Two other frequently used temporal operators can be defined in terms of the operators we have defined so far. They are the \square , or ‘always’ operator, and the \diamond , or ‘eventually’ operator:

$$\begin{aligned} v(\square f, s_i) &\Leftrightarrow (f \cup \text{false}) \\ v(\diamond f, s_i) &\Leftrightarrow (\text{true } \mathbf{U} f). \end{aligned}$$

Standard Formulae

Many standard types of correctness requirements can be expressed with the temporal operators we have defined here. We give two examples of commonly used patterns.

A *progress* property is a temporal formula that can be written in the form $\square\diamond p$. This formula states that at any point in an execution the state formula p is either true or it will become true at some point in the future.

A *guarantee* property is a temporal formula that can be written in the form $\diamond\square p$. This formula states that the state formula p is guaranteed to become invariantly true at some point in the future.

Progress and guarantee are in many ways dual properties. It is, for instance, not hard to show that $\neg\square\diamond f \Leftrightarrow \diamond\square\neg f$.

Some other equivalences [40] are:

$$\begin{array}{ll} \neg\square f & \Leftrightarrow \diamond\neg f \\ \neg\diamond f & \Leftrightarrow \square\neg f. \\ \square(f \wedge g) & \Leftrightarrow \square f \wedge \square g \\ \diamond(f \vee g) & \Leftrightarrow \diamond f \vee \diamond g \\ \square\diamond(f \vee g) & \Leftrightarrow \square\diamond f \vee \square\diamond g \\ \diamond\square(f \wedge g) & \Leftrightarrow \diamond\square f \wedge \diamond\square g \\ f \cup (g \vee h) & \Leftrightarrow (f \cup g) \vee (f \cup h) \\ (f \wedge g) \cup h & \Leftrightarrow (f \cup h) \wedge (g \cup h) \\ f\mathbf{U}(g \vee h) & \Leftrightarrow (f \mathbf{U} g) \vee (f \mathbf{U} h) \\ (f \wedge g) \mathbf{U} h & \Leftrightarrow (f \mathbf{U} h) \wedge (g \mathbf{U} h) \\ \neg(f \mathbf{U} g) & \Leftrightarrow (\neg g) \cup (\neg f \wedge \neg g) \\ \neg(f \cup g) & \Leftrightarrow (\neg g) \mathbf{U} (\neg f \wedge \neg g) \end{array}$$

So far we have defined the evaluation of temporal formulae for specific ω -runs. We will be interested in proving properties of a system for *all* possible executions starting from its initial system state. When we say that f holds for finite state automaton A we mean that it holds for *all* ω -runs that start from A 's initial state. We may, for instance, want to prove that $\square(p \rightarrow \diamond q)$ for the automaton in Figure 2, with p and q defined as state properties:

$$\begin{aligned} p &\equiv \text{empty}(s2r) \\ q &\equiv \text{empty}(r2s) \end{aligned}$$

Equivalently, we may want to prove that the negation of this formula is *not* satisfied. Using the equivalences and the definition of logical implication ($p \rightarrow q \Leftrightarrow \neg p \vee q$) we can write the negation as:

$$\begin{aligned} \neg \square (p \rightarrow \diamond q) &\Leftrightarrow \\ \diamond \neg (\neg p \vee \diamond q) &\Leftrightarrow \\ \diamond (p \wedge \neg \diamond q) &\Leftrightarrow \\ \diamond (p \wedge \square \neg q) & \end{aligned}$$

This formula is satisfied if at some point in an execution the state formula p becomes *true* while q is *false* and remains *false* forever thereafter. Note that this indeed captures the violation of the formula $\square (p \rightarrow \diamond q)$ that we started with. Now consider the automaton in Figure 3.

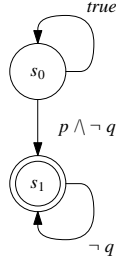


Figure 3 — Non-deterministic automaton for $\diamond (p \wedge \square \neg q)$, with initial state s_0 and accepting state s_1 .

This automaton has two states s_0 and s_1 , with s_0 the initial state. Set D is identical to those of the automata in Figures 1 and 2, $D = \{s2r, r2s\}$. Set L has three elements:

- $true$ which is an action with guard ($true$) and effect *skip*,
- $p \wedge \neg q$ which is an action with guard ($p \wedge \neg q$) and effect *skip*, and
- $\neg q$ which is an action with guard ($\neg q$) and effect *skip*.

Set F, finally, has one element: s_1 .

The accepting runs of this automaton have the following form, written as a sequence of transitions:

$$(true)^+ ; (p \wedge \neg q) ; (\neg q)^\omega$$

where ; indicates concatenation, + indicates finitely many repetitions, and ω indicates infinitely many repetitions. Note that this matches the semantics of $\diamond (p \wedge \square \neg q)$, and could be useful in automating the verification process. The automaton in Figure 3 need not be discovered by trial and error: there are efficient algorithms for constructing it mechanically from the LTL formula [19,17,18,52].

Synchronous Product

How can we use the automaton from Figure 3 to prove our sample property for the alternating bit protocol, i.e., for the automaton from Figure 2? Somehow we must 'match' the runs of the automaton in Figure 3 with the runs of the automaton in Figure 2. We can do precisely this by computing the synchronous product of these two automata.

Definition 4.2. The *synchronous product* of the extended finite automata $A = \{S, s, D, L, T, F\}$ and $B = \{S', s', D', L', T', F'\}$ is an extended finite automaton $\{S'', s'', D'', L'', T'', F''\}$ identical to the asynchronous product except for the definitions of L'' and T'' . L'' is a set of *ordered* pairs $L \times L'$, and $T'' \subseteq S'' \times L'' \times S''$ with $\forall ((n, n'), (l, l'), (m, m')) \in T'' : (l \in L \wedge (n, l, m)) \wedge (l \in L' \wedge (n', l', m'))$.

That is, the edges of the synchronous product of the automata correspond to *joint* transitions of the automata. Since every transition now carries two actions, the guards of *both* actions must evaluate to true for the transition to be valid. For a property automaton that is derived from a temporal formula (like the one in Figure 3) the effect part on each action will always be *skip*, so the order in which the effects are executed is unimportant. (In general this order will matter, so that the synchronous product $A \times B$ may be different from $B \times A$.)

The synchronous product of the automata in Figures 2 and 3 is shown in Figure 4. Valid transitions are drawn solid and invalid transitions are dashed. The only valid ω -run in the automaton from Figure 4 contains no accepting states. We can conclude that formula $\diamond (p \wedge \square \neg q)$ cannot be satisfied in the automaton from Figure 2, and therefore that formula $\square (p \rightarrow \diamond q)$ cannot be violated.

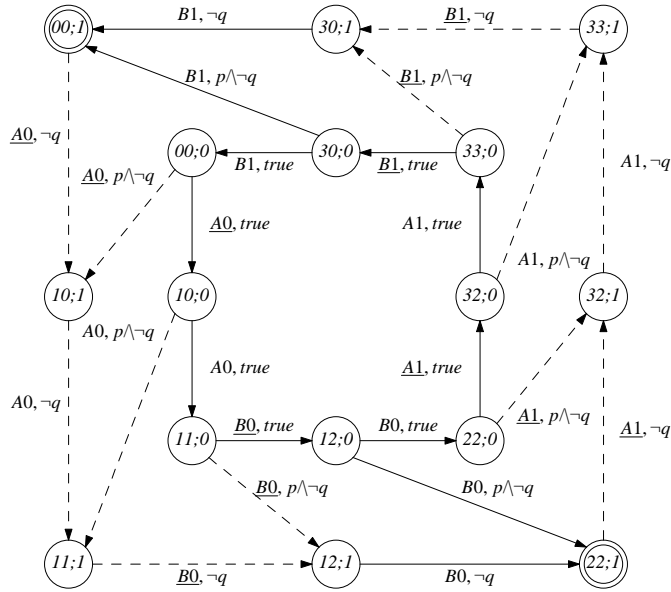


Figure 4 — Synchronous product of the automata in Figures 2 and 3.

5. LTL Model Checking

The set of all ω -runs that an automaton accepts is often referred to as the *language* that is recognized by the automaton. Let $L(M)$ be the language recognized by the automaton that represents the system behavior we are studying, and let f be a temporal logic formula that is required to be satisfied by the system. The verification proceeds in four steps:

1. Mark all states in M as accepting states, to make sure that all ω -runs of M are considered.
2. Compute a Büchi automaton B for $\neg f$ (the negation of f , capturing all possible ways in which f might be violated).
3. Compute the language intersection of $L(B)$ and $L(M)$, by computing the synchronous product of B and M .
4. If the intersection is empty, i.e., if the product automaton accepts no ω -runs at all, M cannot violate f and therefore property f is satisfied.
If the intersection is non-empty, there is at least one ω -run that is accepted by both M and B . Because it is accepted by B it constitutes a violation of property f . The run can be used as concrete evidence that f is not satisfied by M .

In this section we will consider how this automata theoretic method for the verification of LTL formulae can be implemented efficiently.

Depth-First Search

For an accepting ω -run to exist, there must be at least one execution of the product automaton defined above that traverses an accepting state infinitely often. This means that there must exist at least one accepting state in the product automaton that is both reachable from the initial state of that automaton *and* that is reachable from itself. For this to be true the reachability graph for the product automaton must have at least one strongly connected component with an accepting state. The strongly connected components in a graph can be computed in linear time with Tarjan's algorithm [48]. The product of M and B also depends linearly on the numbers of states in the automata M and B .

More problematic is, though, that the sizes of M and B can depend exponentially on the problem size. M is generally given as an asynchronous interleaving product of automata. This means that the size of M can increase exponentially with the number of automata (asynchronous processes) that we consider. B is extracted from an LTL automata, and in the worst case the size of B can also be exponentially larger than the size of the formula, measured by the number of state subformulae in the formula [52].

Fortunately, in practice things are not quite this bad. LTL formulae of practical interest rarely contain more than two or three temporal operators, and the automata generated from them have rarely more than five or six states [18]. The reason is simple: the precise meaning of formulae with more than three temporal operators can be hard to determine. The chains of reasoning required to interpret such a requirement quickly becomes too long to be meaningful in systems verification. In almost all cases of interest, a more complex system requirement can be broken down into smaller steps of just a few basic types: invariance (expressed as $\Box p$), inevitability ($\Diamond p$), progress ($\Box\Diamond p$), and conditional response ($\Box(p \rightarrow (q \cup r))$) [39,43].

The size of the synchronous product of M and B is almost completely determined by the

size of M , which can indeed be large. Contributing factors to the size of M can be the number of asynchronous processes, and the number and the value ranges of data objects used. A number of techniques have been developed to reduce the size of M , and the cost (in time and memory) of analyzing it. We will review the most important of these here. The most frequently used techniques include model reduction and abstraction, partial order reduction, symmetry reduction, on-the-fly verification, state compression, machine minimization, and proof approximation.

Nested Depth-First Search

First let us briefly revisit the central problem in LTL model checking: detecting the existence of at least one cycle through an accepting state, in a finite graph. In the worst case the algorithm must visit every node in the graph, 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. [2]. This requires storing with each node $2x\log(R)$ 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 [26,27]. The algorithm works by splitting the depth-first search into two phases with the help of a two-state demon automaton. We then continue with a discussion of a similar but stronger two-phase search algorithm that can be used to prove the absence or existence of acceptance cycles (accepted ω -runs), so that it can be used to perform LTL model checking [14,29].

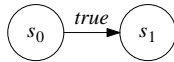


Figure 5 — Two-state non-deterministic demon automaton for detecting non-progress cycles.

This is done by the addition of a two-state demon D , as illustrated in Figure 5. The demon 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 has guard *true* and effect *skip*. We assume that some of the states in the automaton M to be analyzed are marked 'progress' states. We will be interested in finding any ω -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.

We compute the asynchronous product of M and D , and perform a slightly modified depth-first search in the reachability graph for that product. The product machine will be at most twice the size of the original M , containing one copy of each state with the demon in state s_0 , and possibly one more copy with the demon in state s_1 .

Recall that each state s is a tuple consisting of a state of the demon and a state of M . Let $dm(s)$ be *true* if the demon machine is in state s_0 , and let $np(s)$ be *true* if s is not marked as a progress state. The non-progress cycle detection algorithm is then as follows. The search starts from the initial state of the product of M and D , with the demon in state s_0 .

```

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
    }
}

```

Note that we do not consider any successors of progress states when the demon is in state s_1 . Every cycle in the second state space (with the demon in state s_1) is therefore necessarily a non-progress cycle.

Property 5.1. If non-progress cycles exist, $dfs_A()$ will report at least one of these.

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 the demon automaton into 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 state v . We know that r is reachable from v (or else it would not block r from reaching itself) and that v is reachable from r . This means that v is reachable from itself in the second statespace via r . This, however, contradicts the assumption that r was the first state 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. q.e.d.

Whenever a cycle is detected, the corresponding ω -run can be reproduced exactly 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.

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 two bits [20]. 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. (Note that the state is neither present in the first nor the second statespace when the bit combination is 00.) Note that 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 the demon

automaton, therefore would not suffice. The second of the two bits is always equal to the state of the demon automaton, which therefore need not be stored separately.

This non-progress cycle detection algorithm was first implemented in 1988 in the tool `sd1valid`, the immediate predecessor of the SPIN model checker [25], and later incorporated also in SPIN [26,27]. A stronger version of this type of two-phase search algorithm was introduced in [14], and can be used to solve the general LTL model checking problem. This algorithm is known as the *nested depth-first search*.

This time the transitions of the demon automaton 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 M and state d in D . Let `acc(s)` be *true* if and only if state s is accepting. The search starts with the call `dfs_B(s0, s0)`

```

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. The demon machine moves only from accepting states 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 [14].

Property 5.2. 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 is 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 (cf. Figure 6) that is on the depth first search stack. 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 necessarily an accepting state that is reachable from itself, which contradicts the assumption that r is the first such state entered into the second state space.

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. *q.e.d.*

Like `dfs_A`, this algorithm requires no more than two bits to be added to every reachable state in M , 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.

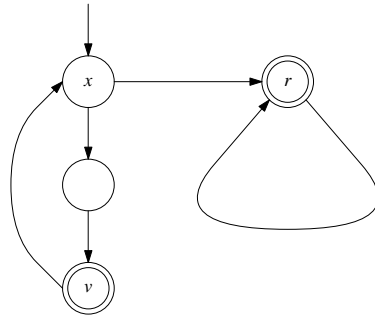


Figure 6 — States v , w and r .

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 7.

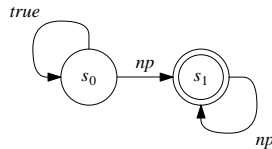


Figure 7 — Two-state automaton for $\Diamond\Box np$.

To perform model checking we can now take the synchronous product of the automaton in Figure 7 with a system M , and use algorithm `dfs_B` to detect the accepting ω -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 7. The earlier algorithm `dfs_A` solves this specific problem more efficiently by

incurring only the doubling from the demon automaton. The advantage of `dfs_B` is that it can handle any type of LTL property, not just non-progress properties.

Adding Fairness

LTL is rich enough to express many fairness constraints directly, e.g., in properties of the form $(\Box \textit{trigger}) \rightarrow (\Diamond \textit{response})$ or $(\Box \textit{guard}(t) \equiv \textit{true}) \rightarrow (\Diamond \textit{effect}(t))$, where t is a transition. More specific types of process fairness can also be predefined and incorporated into a model checking algorithm. Recall that the asynchronous product of finite automata that is the ultimate subject of LTL model checking is built as an interleaving of transitions from smaller automata, $M = M_1 \times M_2 \cdots M_k$. Each of the automata $M_1 \times M_2 \cdots M_k$ contributes transitions to the runs of M . Component automaton M_i is said to be 'enabled' at state s of the global automaton M if s has at least one valid outgoing transition from M_i . We can now define two fairly standard notions of fairness.

Definition 5.1. *Weak fairness:* An ω -run σ is weakly fair if every component automaton that is enabled infinitely often contributes at least one transition infinitely often to σ .

Definition 5.2. *Strong fairness:* An ω -run σ is strongly fair if every component automaton that is enabled infinitely long contributes at least one transition infinitely often to σ .

We can include weak fairness into the nested depth-first search algorithm by using Choueka's flag construction method [10]. The following informally describes the method that is implemented in the SPIN system [30,36]. We multiply the state space $k + 1$ times, with k the number of component automata in the asynchronous product. Only the first copy of the state space retains its acceptance states; the corresponding states in the k additional copies are made non-accepting. Next we change every transition contributed by component i in the original product into a transition from the source state of that transition in copy i of the new product to the destination state of the transition in copy $i + 1$. For the last copy, $k + 1$ all transitions move back to the first copy of the state space. Any accepted ω -run in the new unfolded state space now necessarily includes transitions from each of the k component automata. We have to make one further adjustment to this procedure to account for the fact that a component automaton that is permanently not enabled after some point in the run (strong fairness) or a component automaton that is repeatedly not enabled (weak fairness) need not participate in the run. To implement weak fairness, for instance, we can add a null transition from every state s in copy i to state s in copy $i + 1$ if component i is not enabled at s .

Unfolding the state space k times can be costly, but we can reduce the memory cost to a minimum by storing each copy of a state just once, and annotating it with $k + 1$ bits to record in which copy of the state space the state has been encountered. If we use the cycle detection method from algorithm `dfs_A` or `dfs_B` the memory overhead per reachable state remains limited to $2(k + 1)$ bits.

SPIN's On-the-Fly Implementation

The model checker SPIN performs the LTL model checking procedure on-the-fly, applying the nested depth first search algorithm `dfs_B` during the construction in a single pass of the product $Bx(M_1 \times M_2 \cdots M_k)$ where B is the property automaton for the negation of an LTL formula that should be satisfied, and where x indicates synchronous product, and \times asynchronous product. The construction is optionally modified for Choueka's flag construction to enforce weak fairness. SPIN derives the automaton B from an LTL formula using the algorithm from [19] with some optimizations from [18]. Optionally, the user can also specify B manually, and thus gain an increase in expressive power to the full range of ω regular properties. Alternatively, SPIN also allows the use of the conversion procedure from [18], which adds existential quantification to LTL and thereby also extends the expressive power to the ω regular properties.

The advantage of the on-the-fly procedure is that the construction of the product automaton can stop as soon as an accepting ω -run is found, having delivered proof that the system can violate the requirement. If a system contains an error it usually suffices to construct only a small portion of the product automaton. If the system satisfies the requirement the complete product must be computed. In many cases, though, the property automaton B acts as a constraint on the system, limiting the synchronous product to the executions that are relevant to the property being proven. Therefore, in those cases computing the product $Bx(M_1 \times M_2 \cdots M_k)$ will be cheaper than computing $(M_1 \times M_2 \cdots M_k)$.

Partial Order Reduction

The validity of an LTL formula is insensitive to the precise order in which independent transitions from different component automata are interleaved in any given ω -run of the global automaton. SPIN uses partial order reduction to exploit this fact and to reduce the cost of a typical verification. Instead of generating a full asynchronous product that captures all possible interleavings of transitions, the model checker generates a reduced product, with only a few representatives from each class of ω runs that are indistinguishable for a given LTL formula [28,29]. This reduction can, in the best case, reduce the cost of a verification by a factor that grows exponentially with the number of component automata that are used to construct the asynchronous product. In effect, by applying partial order reduction rules one can achieve that every ω -run that is inspected by the model checker represents a large class of equivalent runs. If at least one run from each equivalence class is considered, all other runs can be ignored. The correctness of the reduction algorithm used in SPIN was verified independently with a theorem prover [9].

Partial order reduction can also be combined with other types of reduction to increase the benefits in some cases, for instance by exploiting possible symmetries in a model, e.g. [16].

Compression Techniques

Memory and time are bounded resources. The challenge in the construction of practical model checking tools is to economize the memory requirements *without* incurring unrealistic increases in runtime requirements. The model checker must be able to determine at each newly generated state from the global product automaton whether or not the state already appears in the state space (named the set `visited` in algorithms `dfs_A` and `dfs_B` above). To do so one typically stores the states in a hash-table and compares the memory image of the new state against that of previously visited states with the same

hash-value. We can reduce memory use by storing all states in compressed form. The comparisons can similarly be done on the compressed memory images of the state. Better still, the compression need not be reversible. The model checker SPIN includes a number of optional lossless compression techniques, allowing for user defined trade-offs between reducing running time and increasing memory savings.

The most effective compression method SPIN supports avoids storing the set `visited` completely, and instead computes a minimized finite automaton that can recognize (optionally compressed) memory images of states as finite words over a predefined alphabet. To add a state, the automaton is updated in a way that secures its continued minimality [35]. The technique is comparable to techniques based on the use of binary decision diagrams that have proven effective in applications of model checking in hardware circuit verification, e.g. [7,13].

Bitstate Hashing

Model checking can be computationally expensive, even with aggressive use of compression and reduction techniques. Large problem sizes can easily defeat the available bounds on memory use and compute time. In cases like these it can be of great value to be able to approximate the answer to a verification problem with an accuracy depends on the ratio by which the problem size exceeds the available resources. We can use lossy compression methods to address this problem. A good example of such a method is the bitstate hashing, or *supertrace* algorithm [24,31,36,56]. This algorithm uses a fixed number of bits of memory per reachable state. The addresses for each of these bits are computed as hash values from the full memory image of a state, with statistically independent hash functions. In the current versions of SPIN the number of bits used can be chosen arbitrarily by the user, but it defaults to two. An elegant theoretical explanation of the working of bitstate hashing can be based on the theory of Bloom filters [4]. A short description will suffice for the purposes of this chapter. A more detailed account can be found in [15,36].

Suppose M bytes of main memory is available to store the set of `visited` states. On average workstations M is typically 2^{30} bytes, and likely to increase in coming years. We now compute N independent hash values of $\log_2 8M$ bits for each state (assuming 8 bits per byte). Instead of storing $N \log_2 8M$ bits, though, we interpret the N hash-values as a bit-address in M , and store only one single bit at each address (by changing that bit from 0 to 1). If the (compressed) memory image of a state is longer than $N \log_2 8M$ bits, this method will lose information. It is now possible that two different states generate the same N bit addresses. The model checker will then assume that a newly generated state matches a previously visited state and fail to generate the successors of the new state. Because only *visited* states are stored in this way, and not state information from the depth-first search stack (sometimes called *open* states or *stack* states), the omissions due to hash-collisions can cause the model checker to miss error states, but it cannot cause it to generate false error reports.

By virtue of the accuracy of all information saved on the depth-first search stack, the depth-first search is guaranteed to proceed correctly, generating only accurate complete execution sequences, though perhaps not all of them in the presence of bitstate hash collisions. The coverage of the search is truncated *randomly*, by a factor that depends on the amount of information that is lost in the hashing.

Because it is impossible to predict systematically which execution sequences of a model might lead to error, an unbiased random truncation of the search space turns out to be a

desirable feature of this search process.

Trivially, if the minimum amount of memory to store one reachable system state without loss of information requires S bytes of memory, and if our machine has M bytes of memory available, the model checker exhausts memory after generating M/S states. If the true number of reachable states R exceeds M/S , then the *problem coverage* of that verification run is $M/(R \times S)$. If, for example, M is 10^8 bytes, S is 10^3 bytes, and R is 10^6 states, then the problem coverage can be no more than 0.1, meaning that no more than 10% of the reachable states are visited. Under the same system constraints, the bitstate hashing algorithm, storing 2 bits per state, can record up to 4 states per byte and could still achieve close to 100% coverage, given that $M/R \gg 4$. In general, when $M < R \times S$, a bitstate hashing technique almost always realizes greater problem coverage than a standard model checking run [31]. Since its first introduction in 1987 the bitstate hashing method has become a trusted technique that was adopted in almost all academic and commercial verification tools to deal with problems that exceed the normal bounds for exhaustive verification.

The bitstate hashing technique allows the user to set the range of bit addresses that can be used, typically matching the maximum amount of memory that is available for a verification run. Clearly, the larger this hash-array, the more states can be stored in it, the larger the coverage will be, and the longer it may take to complete the verification. This gives the user additional control over the search process: by artificially limiting the available hash-array, the user can obtain a very fast and very coarse approximation. By slowly increasing the size of the hash-array, the coverage and the runtime expense can be increased in a controlled manner. Each increase in coverage that fails to locate errors also increases our confidence in the likely correctness of the system. When the system contains errors, it usually takes only a small number of approximations to locate representative samples. Only when the system is correct, in the last phase of a design, more significant resources need to be invested to prove it. This *iterative search refinement* technique was used in the verification of the call processing software for a telephone switch [34].

6. Model Extraction and Abstraction

The construction of models of real-world applications for the purposes of verification is hardly novel and assuredly not restricted to the field of distributed software. There is a long tradition of the use of physical and mathematical models in civil engineering, and scientific disciplines like physics or chemistry would be almost unthinkable without theoretical models that attempt to capture aspects of nature. A model is always an abstraction: by abstracting from detail deemed immaterial to properties of interest we lose scope but gain analytical power.

It is well known that even simple properties of arbitrary software are undecidable or only semi-decidable [51]. This means that model construction for the verification of distributed software is not just an option: it is a necessary step. By defining an abstraction we can reduce a given software application to a finite model, consisting of finite state automata, that can be analyzed with the procedures outlined in this chapter. This reduction will bring a loss of information, so it has to be chosen in such a way that relevant information is preserved and irrelevant detail removed. What is *relevant* and what is not depends on the properties that we are interested in proving.

We can remove the detail in such a way that the soundness of the model checking procedure itself is not endangered [1,6,12,38]. This means that if the model checker indicates that a model satisfies a property, the original software necessarily also satisfies the

property. Conversely, if the model checker generates an error, the error sequence can be checked against the original software to determine its validity. If it is valid, an error in the application has been exposed. If it is not valid, we have obtained proof that the abstraction was chosen incorrectly, and the error sequence itself can be used to determine unambiguously how the abstraction should be revised.

A simple abstraction method of this type was used in the application of the SPIN model checker to complex call processing software for a commercial telephone switch, e.g., [33,34]. With this method, a parser automatically *extracts* an annotated control flow skeleton from the source code of the application. A lookup table defines precisely which statements from the program should be omitted from the model (i.e., replaced with *skip* statements), which should be abstracted with user-defined functions, and which should be preserved within the model. Within the model, every statement from the original source code of the application is mapped into a finite domain and represented as a transition in an extended finite state automaton, expressed in guards and effects that operate on finite data objects, often with a reduced range of possible values.

Function calls have to be treated with some care, in the interest of controlling the complexity of a model. The very presence of a function call, however, can be taken as a hint from the programmer that an abstraction can be made. In the call processing application function calls were treated like statements: they were either omitted if the functionality provided was outside the scope of the verification, or they were abstracted, either with a small inline routine or with a non-deterministic choice of the possible return values. As an example, a routine that determines the availability of a resource, like a tone circuit, is best abstracted with a non-deterministic choice between the two possible result values: available or not-available. No useful gain is made if we were to include more detail than this. As another example, the call of a routine that issues billing records can be omitted from the model if billing is not the focus of the verification.

In the call processing application the focus was on the verification of correct feature behavior for the telephone switch. Requirements for over twenty different feature packages, such as call waiting, call forwarding, call screening, conference calling, etc., are specified in Telcordia standards for call processing [49]. Each relevant property was formalized in linear temporal logic. Aspects of the system that were outside the scope of our verification effort (e.g., billing, process management, memory management, device driver code) was mechanically omitted from the model, and helped to reduce the cost of the verification of the remaining aspects of the code.

The advantage of this method is that it can be almost completely automated. When a new version of the source code is prepared, the model extraction program can prompt the user to provide missing and redundant entries in the lookup table. Once the lookup table has been updated, the model checking process can be repeated with a new accurate model being extracted from the source code of the application mechanically, typically in a fraction of a second.

The verification method allowed us to track the evolution of the call processing code for this application over a period of 18 months. The source code for the application grew fivefold in size in this period, and went through approximately 300 different versions, often changing daily. Approximately 75 critical errors were intercepted with the model checking technique we have outlined, at an early stage of the design, giving a clear indication of the considerable power and value of software model checking techniques. Many of the errors found involved subtle race conditions in the code that could disturb

required functionality. Such errors are virtually impossible to find with conventional testing techniques.

Other Uses of Abstraction

The model extraction method sketched above was greatly facilitated by a relatively recent extension of the SPIN model checker that allows for the inclusion of *embedded C code* inside higher level verification models [36]. This capability to use verify embedded C code fragments can be used in a number of other ways to increase the power of the model checking approach. In [37] a method is described that allow SPIN to verify a code module at implementation level, by compiling it with, and linking it to the model checker. The model checker now generates the non-deterministic input sequences for the code module and keeps track of the code's state. To achieve this, the user identifies the concrete data objects inside the code module that contain state information. The user can at this point also define an abstraction function, in C code, that takes the concrete representation of the state information and abstracts it for use by the model checker. In this way we can use SPIN to combine model abstraction without model extraction, which may prove to be a very effective technique for handling large verification problems in years to come.

7. Perspective

The software model checking techniques that we have reviewed in this chapter are based on finite automata, linear temporal logic, depth-first search, partial order reduction, and explicit state representation combined with powerful memory management techniques. It has been successfully applied in many domains, but typically to verification problems that involve asynchronous threads of computation from software systems e.g., [46]. An overview of applications can also be found in [30].

It is interesting to compare the general framework used in SPIN with the one that has been developed for hardware circuit verification. The most commonly used logic in hardware verification is the branching time logic CTL [11], the search strategy is often breadth-first, instead of partial order reduction techniques one uses BDD based algorithms [7], and instead of explicit state representation one uses symbolic model checking [41]. These differences in approach to the verification problem can be understood better if we look at some of the differences between the two domains of application. Hardware is typically clock-driven, operating in a synchronous fashion, while the processes in a distributed system are necessarily asynchronous. At the hardware level information travels as signals, in software applications the information is represented, manipulated, and moved in composite data structures. A bit level representation is clearly not helpful for these types of objects. The structure of a hardware system, finally, can often be defined statically, while in a software system one must deal with dynamically growing and shrinking numbers of asynchronous processes and data objects. These differences mean that few of the reduction techniques that work well in software model checkers show benefit when used in hardware model checkers, and vice versa.

Acknowledgements

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

8. References

- [1] M. Abadi and L. Lamport, The existence of refinement mappings. *Theoretical Computer Science*, Vol. 82, No. 2, May 1991, pp. 253-284.
- [2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1974.
- [3] 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.
- [4] B.H. Bloom, Spacetime trade-offs in hash coding with allowable errors. *Comm. of the ACM*, Vol. 13, No. 7, pp. 422-426.
- [5] G.V. Bochmann, *Finite state description of communications protocols*, Publication NO. 236, Departement d'informatique, Universite de Montreal, July 1976.
- [6] D.M. Bozga, *Vérification symbolique pour les protocoles de communication*. PhD Thesis (in French), University of Grenoble, France, December 1999, Chapter 4.
- [7] E. Bryant, Graph-based algorithms for boolean function manipulation, *IEEE Trans. on Computers*, Vol. C-35, No. 8, August 1986, pp. 677-691.
- [8] J.R. Büchi, On a decision method in restricted second order arithmetic. *Proc. Intern. Congr. on Logic, Methodology and Philosophy of Science*, Stanford Univ. Press, Stanford, CA, 1960, pp. 1-11.
- [9] C-T. Chou, and D. Peled, Verifying a Model-Checking Algorithm, *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, March 1996, Passau, Germany, LNCS 1055, Springer Verlag, pp. 241-257.
- [10] Y. Choueka, Theories of automata on ω -tapes: a simplified approach, *Journal of Computer and System Science*, Vol. 8, 1974, pp. 117-141.
- [11] E.M. Clarke, and E.A. Emerson, Synthesis of Synchronization Skeletons for Branching Time Temporal Logic, *Workshop on Logic of Programs*, Yorktown Heights, NY, May 1981, Springer-Verlag, Lecture Notes in Computer Science, Vol. 131.
- [12] E.M. Clarke, O. Grumberg, and D.E. Long, Model checking and abstraction. *ACM-TOPLAS*, Vol. 16, No. 5, pp. 1512-1542, September 1994.
- [13] E.M. Clarke, O. Grumberg, and D. Peled, *Model checking*, MIT Press, Cambridge, Mass., 1999.
- [14] 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.
- [15] P.C. Dillinger, and P. Manolios, Fast and Accurate Bitstate Verification for SPIN, *Proc. 11th SPIN Workshop*, Barcelona, Spain. LNCS, Vol. 2989, Springer Verlag, April 2004.
- [16] E.A. Emerson, S. Jha, D. Peled, Combining partial order reduction and symmetry reduction, *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, Enschede, The Netherlands, 1997, LNCS 1217, Springer-Verlag, pp. 19-34.
- [17] K. Etessami, Stutter-invariant languages, ω -automata, and temporal logic, *Proc. Conf. on Computer Aided Verification*, CAV, 1999, pp. 236-248.
- [18] K. Etessami and G.J. Holzmann, Optimizing Büchi automata, *Proc. CONCUR2000*,

- Springer Verlag, LNCS 1877, Aug. 2000, pp. 153-167.
- [19] R. Gerth, D. Peled, M. Vardi, and P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, *Proc. Symp. on Protocol Specification, Testing, and Verification*, Warsaw, Poland 1995, Chapman and Hall, pp. 3-18.
- [20] 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.
- [21] J. Hajek, Automatically verified data transfer protocols, *Proc. 4th ICCV*, Kyoto, 1978, pp. 749-756.
- [22] G.J. Holzmann, *PAN: a protocol specification analyzer* Technical Report TM81-11271-5, AT&T Bell Laboratories, March 1981.
- [23] G.J. Holzmann, A Theory for protocol validation, *IEEE Trans. on Computers*, Vol. C-31, No.8, pp. 730-738, 1982.
- [24] G.J. Holzmann, An improved protocol reachability analysis technique, *Software, Practice and Experience*, Vol. 18, No. 2, Febr. 1988, pp. 137-161.
- [25] G.J. Holzmann and J. Patti, Validating SDL Specifications: An Experiment, *Proc. Int. Conf on Protocol Specification, Testing, and Verification*, Twente, Neth., June, 1989, pp. 317-326.
- [26] G.J. Holzmann, SPIN — A protocol analyzer, *Unix Research System Tenth Edition, Volume II, Papers, Saunders College Publ.*, pp. 423-429. January 1990.
- [27] G.J. Holzmann, *Design and validation of computer protocols*, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [28] G.J. Holzmann, and D. Peled, An improvement in formal verification, *Proc. Conf. on Formal Description Techniques*, FORTE, October 1994, Bern, Switzerland, pp. 177-194.
- [29] 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*.
- [30] G.J. Holzmann The model checker SPIN, *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, May 1997, pp. 279-295.
- [31] G.J. Holzmann, An Analysis of Bitstate Hashing, *Formal Methods in System Design*, Kluwer, Vol. 13, No. 3, Nov. 1998, pp. 287-305.
- [32] G.J. Holzmann, Designing executable abstractions, *Proc. Formal Methods in Software Practice*, March 1998, Clearwater Beach, FL., ACM Press.
- [33] G.J. Holzmann, and M.H. Smith, A practical method for the verification of event driven systems. *Proc. Int. Conf. on Software Engineering*, ICSE99, Los Angeles, pp. 597-608, May 1999.
- [34] 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 Academic Publ., 1999, pp. 481-497.
- [35] G.J. Holzmann, and A. Puri, A Minimized Automaton Representation of Reachable States, *Software Tools for Technology Transfer*, Springer, Nov. 1999, Vol. 2, No. 3, pp. 270-278.
- [36] G.J. Holzmann, *The SPIN Model Checker - Primer and Reference Manual*,

Addison-Wesley, Boston, Mass., 2004.

- [37] G.J. Holzmann and R. Joshi, Model-driven software verification, *Proc. 11th SPIN Workshop*, Barcelona, Spain, April 2004, Springer Verlag, LNCS 2989, pp. 77-92.
- [38] R.P. Kurshan, Homomorphic reduction of coordination analysis. *Mathematics and Applications*, IMA Series, Springer-Verlag, Vol. 73, pp. 105-147, 1995.
- [39] Z. Manna, and A. Pnueli, *Tools and rules for the practicing verifier*, Stanford University, Report STAN-CS-90-1321, July 1990, 34 pgs.
- [40] Z. Manna, and A. Pnueli, *The temporal logic of reactive and concurrent systems: Specification*, Springer-Verlag, 1991.
- [41] K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic, Boston, 1993.
- [42] G.E. Moore, Cramming more components onto integrated circuits, *Electronics*, 19 April, 1965.
- [43] D. Peled, On projective and separable properties, *Colloquium on trees in algebra and programming*, Edinburgh, Scotland, 1994. Springer Verlag, LNCS 787, pp. 291-307.
- [44] A. Pnueli, The temporal logic of programs, *Proc. 18th IEEE Symposium on Foundations of Computer Science*, 1977, Providence, R.I., pp. 46-57.
- [45] J.P. Queille and J. Sifakis, Specification and Verification of Concurrent Systems in Cesar, *Proc. of Fifth Int. Symp. on Programming*, 1981, pp. 337-350.
- [46] F. Schneider, S.M. Easterbrook, J.R. Callahan, and G.J. Holzmann, Validating requirements for fault tolerant systems using model checking, *Proc. Int. Conf. on Requirements Engineering*, ICRE, April 1998, Colorado Springs, Co., IEEE, pp. 4-14.
- [47] A.S. Tanenbaum, *Computer Networks*, 1st Ed. 1981, Prentice Hall, Englewood Cliffs, New Jersey, (2nd edition 1988).
- [48] R.E. Tarjan, Depth first search and linear graph algorithms,” *SIAM J. Computing*, 1:2, pp. 146-160, 1972.
- [49] LATA Switching Systems Generic Requirements (LSSGR), *FR-NWT-000064*, 1992 Edition. Feature requirements, including: *SPCS Capabilities and Features*, SR-504, Issue 1, March 1996. Telcordia/Bellcore.
- [50] W. Thomas, Automata on infinite objects, *Handbook of Theoretical Computer Science*, Elsevier Publ., 1990, Ed. J. Van Leeuwen, Volume B, pp. 133-187.
- [51] A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Mathematical Soc.*, Ser. 2-42, 1936, pp. 230-265, see p. 247.
- [52] 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.
- [53] C.H. West, General technique for communications protocol validation, *IBM J. Res. Develop.*, 1978, Vol. 22, No. 3, pp. 393-404.
- [54] C.H. West, and P. Zafiropulo, Automated validation of a communications protocol: the CCITT X.21 recommendation, *IBM J. Res. Develop.*, 1978, Vol. 22, No. 1, pp. 60-71.
- [55] P. Wolper, M.Y. Vardi and A.P. Sistla, Reasoning about Infinite Computation Paths, *Proc. 24th IEEE Symposium on Foundations of Computer Science*, Tucson, 1983, pp. 185-194.

[56] P. Wolper, and D. Leroy Reliable hashing without collision detection, *Proc. Conf. on Computer Aided Verification*, Crete, June 1993, LNCS, Vol. 697, Springer-Verlag, pp. 59-70.