PROTOCOL VALIDATION 11

214 Introduction 11.1
214 A Manual Proof Method 11.2
218 Automated Validation Methods 11.3
226 The Supertrace Algorithm 11.4
231 Detecting Non-Progress Cycles 11.5
234 Detecting Acceptance Cycles 11.6
235 Checking Temporal Claims 11.7
235 Complexity Management 11.8
237 Boundedness of PROMELA Models 11.9
238 Summary 11.10
239 Exercises
240 Bibliographic Notes

11.1 INTRODUCTION

In Chapter 9 we studied the problem of checking that the implementation of a protocol conforms to a formal specification. We now discuss the problem of verifying the logical consistency of the formal specification itself, independent of an implementation. For consistency we assume that the specification is formalized as a validation model in PROMELA, although this is not essential to many of the algorithms we discuss. We first describe a manual proof method based on the notion of state invariants only. We then show how the same principle can be used to build an automated validation system. Finally, we extend the algorithms to support also the verification of the other correctness requirements that can be expressed in PROMELA (see Chapter 6).

Most automated validation systems are based on exhaustive reachability analysis. To establish the observance of state invariants, then, it suffices to verify their correctness with a simple boolean test for each state that is reachable from a given initial system state. The main problem that must be addressed in the design of such a system is the ‘‘state space explosion problem.’’ For protocols of a realistic size, the number of reachable system states is usually too large for purely exhaustive analyses. We discuss the nature of this problem and some of the counter-strategies that have been developed.

11.2 A MANUAL PROOF METHOD

Consider a simple transmission system with a sender *S* and a receiver *R*. Process *S* sends messages to process *R* over an unreliable transmission medium that can lose but not insert, reorder, or distort messages. Every message transmitted carries a sequence number. Initially, this number is zero, and it is incremented by one for every new message transmitted. It can grow arbitrarily large. The receiver acknowledges the receipt of messages by echoing the sequence numbers over a similarly unreliable return channel. The receiver stores the largest sequence number it has received in a local variable B. The sender tries to keep track of that number by maintaining a count in a local variable A. The value of A is equal to the largest sequence number that the sender can be certain R has received. Initially, we have

$$A = B = 0$$

In the following we assume that sender and receiver simply exchange sequence numbers and no other data. The protocol is then defined by four atomic operations, two in each process. They can be formalized in PROMELA as follows, where for the time being we will pretend that data of the type `int` have unbounded range. `W` is an arbitrary positive constant.

```
mtype = { mesg, ack }

proctype S()
{       int A;
        do
        :: R!mesg(A + rand()%W)         /* S1 */
        :: S?ack(A)                     /* S2 */
        od
}

proctype R()
{       int B, b;
        do
        :: S!ack(B)                     /* R1 */
        :: atomic {                     /* R2 */
                R?mesg(b);
                B = fct(b,B);
           }
        od
}
```

Transition `R2` consists of two statements that are, at least conceptually, executed in one indivisible step. In the first step a new message is received. In the second step a new value for B is obtained via a function `fct()`. The function records the reception of a message numbered b and returns a value $X \geq B$ for which it can guarantee that all messages with numbers smaller than X were recorded by `fct()` before. It could accomplish this, for instance, by setting

```
        if
        :: (b == B+1) -> B = b
        :: (b != B+1) -> skip
        fi
```

forcing messages to be received in sequence, but it could also be more liberal (see Chapter 4).

Assuming that there are r messages in queue R and s acknowledgments in S, with

$$r \geq 0 \quad \text{and} \quad s \geq 0$$

the following condition holds invariantly for the acknowledgments that are buffered in *S*:

$$A \le S[1] \le S[2] \le \cdots \le S[s] \le B \tag{1}$$

The correctness of this system invariant is proven by induction. First notice that in the initial state the channels are empty and the invariant reduces to $A \le B$, which holds trivially since $A = B = 0$. Next observe that if the invariant holds in an arbitrary system state it must hold in all its successor states, since it cannot be invalidated by the four atomic operations:

- `S1` does not change any of the variables in (1). `S2` transforms (1) into

  $$A = S[1] \le S[2] \le \cdots \le S[s] \le B$$

  which must hold if (1) holds. `R1`, assuming that the acknowledgment is not lost, transforms (1) into

  $$A \le S[1] \le S[2] \le \cdots \le S[s] \le S[s+1] = B$$

  which also must hold if (1) held before `R1` was executed. `R2` can increase, but never decrease, the value of *B*, and thus cannot invalidate the invariant either.

Together, this proves the validity of invariant (1). The next invariant applies to the *r* messages waiting in queue *R*:

$$R[i] < R[j] + W, \quad \text{for} \quad 0 \le i \le r \text{ and } i < j \le r+1 \tag{2}$$

where, for convenience, we define

$$R[0] = B \quad \text{and} \quad R[r+1] = A$$

In the initial state, with $r = 0$, the queue is empty, and the invariant becomes $B < A + W$ which trivially holds for all $W > 0$, since $A = B = 0$. We must check again that the correctness of the invariant is unaffected by the four atomic operations.

- `S1` can add an element $r+1$ to queue R (if the message is not lost):

  $$A \le R[r+1] < A + W \tag{2a}$$

  and then increment *r*. There are only two cases to consider where the invariant could now be violated: $i = r$ and $j = r$. For $i = r$, invariant (2) states

  $$R[r] < R[r+1] + W$$

  By definition, this means

  $$R[r] < A + W$$

  which (2a) clearly cannot violate. For $j = r$, invariant (2) states

  $$R[i] < R[r] + W, \quad \text{for} \quad 0 \le i < r$$

  Since (2a) guarantees that $R[r] \ge A$, after `S1` completes, this reduces to

  $$R[i] < A + W, \quad \text{for} \quad 0 \le i < r$$

which must hold if it held before `S1` was executed. `S2` can only increase the value of A, as a direct result of (1). `R1` does not change any of the variables in (2). `R2` deletes a message from the queue, thus removing one of the conditions from the invariant. Either it has no effect or it sets $R[0]=R[1]$, which also cannot disturb the correctness of (2).

This completes the proof of invariant (2).

THE WINDOW PROTOCOL INVARIANT

Invariants (1) and (2) can be used to prove a more general property of the window protocol.

$$B-W \le R[i] < B+W \quad \text{for} \quad 1 \le i \le r \tag{3}$$

□ To prove this, first note that by invariant (2) we have

$$R[i] \ < \ A+W \quad \text{for} \quad 1 \le i \le r$$

Since by invariant (1) we also have $A \le B$ the right side of (3) is easily proven. Second, by invariant (2) we have

$$B \ < \ A+W \quad \text{or} \quad B-W \ < \ A$$

Since by invariant (1) we also have $A \ \le \ R[i]$ the left side of (3) is also proven.

Invariant (3) implies that the receiver can deduce the true value of a message (i.e., its sequence number) even if only part of the value is transmitted, for instance the value modulo $2W$. It is an elegant demonstration that the selective repeat ARQ protocol, discussed in Chapter 4, needs a range of sequence numbers that is twice the window size W.

DISCUSSION OF MANUAL PROOFS

The proof technique we have discussed was first described by Stein Krogdahl and later refined by Donald Knuth. It is based on the notion of state invariants. Unlike the methods used in most automated validation systems, this method is not based on the inspection of reachable system states, but on the inspection of state transitions. There are usually far fewer state transitions than reachable system states. The example system illustrates this nicely: since the sequence numbers are unbounded, the number of reachable system states is infinite, but the number of state transitions is restricted to four. The effort required to verify that a transition cannot invalidate an arbitrary system invariant, however, can be substantial.

In independent work, Mohamed Gouda (see Bibliographic Notes) has argued that all manual proofs can be build on just two basic notions:

- ○ System invariants, and
- ○ Well-founded formulas

A well-founded formula can be used, for instance, to prove termination or to build induction proofs. To construct such a proof we must find a quantity that is inevitably decreased during the lifetime of the program and that forces a desirable outcome of the program when it reaches a minimum. To find the right invariants and well-founded formulas can be hard. In general, the manual proofs must be structured carefully, requiring the user to find and to prove a series of intermediate invariants before the correctness of a more general property can be demonstrated. The advantage of this approach is that it forces the user to thoroughly understand both the design problem and the suggested solution.

This advantage, however, can turn into a disadvantage when the method is applied to larger problems. The manual proofs can be tedious, and they are inevitably susceptible to human error, much like the protocol design that is the subject of the proof. For each invariant that is to be proven the method may require a manual inspection of all atomic state transitions within the system. The manual techniques break down in cases where validation is needed most, i.e., for larger protocols. We accept here, therefore, that there is a need for automatic tools to help us either in constructing proofs, or in finding counter-examples to correctness claims (a euphemism for ‘‘debugging’’). After all, even a proof is not a proof unless its validity can be checked. To quote Lamport [1977]:

> *‘‘A formal proof is one which is sufficiently detailed, and carried out in a sufficiently precise formal system, so that it can be checked by a computer.’’*

Although there is no simple algorithm that could automate the manual proof methods we have discussed, there is, at least for finite state systems, an alternative. The alternative becomes possible if we base our proof method directly on reachable system states, rather than indirectly on the transitions that connect them. Methods of this type can be used to validate both properties of states and properties of sequences of states, as discussed in Chapter 6. The remainder of this chapter is devoted to a discussion of these methods.

11.3 AUTOMATED VALIDATION METHODS

Let us look at the general structure of automated validation systems based on reachability analysis. Initially, we will consider only the validation of state properties, such as assertion violations and improper terminations. We discuss in some detail the limitations of the reachability analysis methods and the strategies that have been developed to exploit them. In later sections we show how the method can be extended to validate properties of sequences of states, such as non-progress conditions and temporal claims, as discussed in Chapter 6.

REACHABILITY ANALYSIS ALGORITHMS

A reachability analysis algorithm tries to generate and inspect all the states of a distributed system that are reachable from a given initial state. Implicitly, it will construct all possible execution sequences, although, depending on the type of algorithm used, not all information about state sequences is necessarily available for analysis. There are three main types of reachability analysis algorithms. In the order in which they are listed here, they can be applied to systems of increasing complexity:

○ Full search (systems up to 10^5 states)

- ○ Controlled partial search (systems up to 10^8 states)
- ○ Random simulation (larger systems)

The full search is the simplest algorithm. It performs the most thorough analysis of the three types of algorithm, but it can only analyze the smallest class of protocols. We quantify the limitations later in this chapter. If the full search method exceeds its limits, it effectively reduces to an uncontrolled partial search method, and the quality of the analysis deteriorates quickly.

The controlled partial search tries to optimize the quality of the reachability analysis specifically for those cases where a full search is infeasible. It attempts this by selecting an optimal fraction of the full state space that can be searched within given constraints of memory and time.

Random simulation techniques are specifically meant for the validation of systems of a complexity that defeats even the controlled partial search. The system state space for these systems can be estimated to be so large that no partial search technique can make a sensible selection. The best possible search in these cases is a random, or biased random, walk of the state space.

There are two different measures for expressing the capabilities of a reachability analysis tool: *coverage* and *quality*. The search coverage is easily quantified as the number of system states tested divided by the number of states in the full state space. A perhaps more appropriate, but less easily quantified measure, is the search's ability to find errors: the number of distinct errors found divided by the total number of errors present. In the comparison of the three basic search methods below we use both measures. In Chapter 13 we develop an automated protocol validation system for PROMELA models that can perform reachability analysis in all three basic modes: random simulations and either fully exhaustive or partial state space searches.

11.3.1 FULL STATE SPACE SEARCH

The standard full, or exhaustive, search algorithm explores all reachable composite system states of a set of interacting finite state machines. How precisely the interaction among the machines is defined is largely irrelevant to the design of the search algorithm. The basic state machine model can be extended with finite message queues, or local and global variables. As discussed in Chapter 8, these additions do not extend the power of the finite state machine model, provided that they are defined over a finite domain.

A state machine, in this model, is defined by a finite number of states and state transitions. Each state transition has two parts: a pre-condition and an effect. The pre-condition is typically a boolean condition on the state of the machine, the queues, and the variables. The transition is enabled, and can be executed, only if the pre-condition holds. The effect of an execution can change the state of the system, for instance the states of the local machine, the queues, and the variables, and perhaps even the state of other machines (e.g., in a multi-party rendezvous system).

The system as a whole is defined by the composite of all individual machine, variable, and queue states, and the combination of all simultaneously enabled local state transitions. From here on, the term *state* is used as a short-hand for *composite system state*. Where this can cause confusion we use the terms *machine state* or *system state*. Given an initial state for each machine in the system, the sets of machine states and system states can each be divided into two disjoint classes: reachable states and unreachable states. Normally it is required that the system not contain any unreachable *machine* states: they would correspond to unexecutable code in an implementation. Normally, also, the set of unreachable *system* states is several orders of magnitude larger than the set of reachable system states. The set of unreachable system states should include all error states.

An exhaustive reachability analysis tries to determine which states are reachable and which are not. Every reachable state and every sequence of reachable states can be checked for a given set of correctness criteria. These criteria can be general conditions that must hold for any protocol, such as the absence of deadlocks or buffer overruns, or they can be protocol-specific requirements such as a temporal claim about the proper working of a message retransmission discipline. In many cases protocol-specific requirements can be formalized as state invariants, the correctness of which can be verified with a simple boolean test in every reachable system state.

In the algorithm below, the reachability analysis starts with a small routine named `start()` that initializes two sets: a working set of system states to be analyzed, called `W`, and a set of states that have been analyzed, called `A`.

```
start()
{       W = { initial_state };  /* work set:   to be analyzed */
        A = { };                /* previously analyzed states */
        analyze();
}
```

Set `A` is also referred to as the *system state space*. When the algorithm terminates, it should include all the reachable system states The basic structure of the reachability analysis algorithm is as follows.

```
analyze()       /* exhaustive or full search */
{       if (W is empty) return;

        q = last element from W;
        add q to A;
        if (q == error_state)
                report_error();
        else
        {       for each successor state s of q
                        if (s is not in A or W)
                        {       add s to W;
                                analyze();
                        }
        }
        delete q from W;
}
```

The order in which states are retrieved from working set `W` seems irrelevant at first, but it turns out to be an important control point. If states are stored in set `W` in *first-in last-out* (i.e., stack) order, the algorithm performs a depth-first search of the state space tree. If states are stored and removed in *first-in first-out* order, this changes into a breadth-first search (element `q` must be deleted upon retrieval from set `W` in this type of algorithm). A breadth-first search has the advantage that it finds the shortest error sequences first. A depth-first search, however, has the advantage that it requires a smaller work set `W`. An intuitive explanation for this is that the size of `W` in a depth-first search is a function of the depth of the search tree, but a function of its width in a breadth-first search. The depth of the search tree depends on the maximum length of a unique execution sequence. The width of the tree, however, is determined by the maximum number of distinct execution sequences, which is usually a much larger number.

As an example, consider a protocol where every state has two successors. The state space is then equivalent to a binary expanding tree. After n transitions, the breadth of the search tree is 2^n states. The depth of the tree, however, is only n states.

There is one other important advantage to the depth-first search discipline. When an error is discovered we would like the algorithm to be able to produce an execution sequence that leads to the error via a valid sequence of state transitions, starting from the initial system state. With a breadth-first search method, the path from the initial system state must be reconstructed from information stored in the state space set `A`. With a depth-first search, however, such a path need not be reconstructed: a sequence is implicitly defined by the stack order of set `W`.

DISCUSSION OF THE FULL SEARCH METHOD

The main problem with the full search strategy is its restricted applicability. It is important to note that the coverage of the full state space search technique is not necessarily 100%: it depends on the size of the state space and the amount of memory that is available for the search. If the size of the state space is R and the maximum number of states that can be stored in memory during the search is A both the coverage and the search quality can only reach 100% when $R \leq A$. When $R > A$ the coverage reduces to A/R, but the search *quality* is likely to be worse.

For large protocols the exhaustive search algorithm deteriorates into a low-quality partial search.

Consider a protocol for two processes, each having 100 states, one message queue, and each accessing five local variables. The two message queues are restricted to five slots each, and the effective range of the local variables is assumed to be limited to ten values. The number of distinct messages exchanged is ten. In this relatively small example system, there are $10^{5 \cdot 2}$ possible states of the protocol variables. Each process can be in one of 10^2 different states, so two processes can maximally be in 10^4 different composite system states. Finally, each queue can hold between zero and five messages, where each message can be one out of ten possible messages. The total number of system states in the worst case then is

$$10^{10} \cdot 10^{4} \cdot \left[\sum_{i=0}^{5} 10^{i} \right]^{2}$$

or in the order of 10^{24} different states. If we assume, quite unrealistically, that each state can be encoded in 1 byte of memory and can be analyzed in 10^{-6} sec of CPU time, we would still need a machine with at least 10^{15} times as much memory as currently available, and would need roughly 10^{11} *years* of CPU time to perform an exhaustive analysis.

Fortunately, the number of effectively reachable states is usually much smaller than the total number of system states calculated above. After all, it is the purpose of a protocol to restrict the the behavior of the protocol processes, and thus the number of effectively reachable states, in order to realize the desired functionality. Still, even relatively small protocol systems can easily generate anywhere from 10^5 to 10^9 reachable system states. The number of states grows dramatically if, for instance, the size of a message queue is increased, or if the assumptions about the behavior of the ‘‘environment’’ in which the protocol is executed (e.g., the channel characteristics) are relaxed.

The exhaustive search method unavoidably breaks down when the state space grows beyond approximately 10^5 states. A quick ‘‘back of the envelope’’ calculation can illustrate this.

> If one system state can be stored in S bytes of memory, and we have a machine with M bytes available, we can generate and analyze no more than M/S states. M is a machine-dependent constant that is typically in the range from 10^6 to 10^7. Values for S are typically in the range 10 to 10^2 bytes, with larger values corresponding to the larger numbers of reachable states. This leads to an estimate for the maximum state space size of about 10^5 states. This value can also be found experimentally by running the full search algorithm until it has exhausted available memory.

This means that in many cases the full search method is feasible only if we can reduce the complexity of our validation models to the maximum that a given machine can analyze. The complexity of protocol models can be reduced substantially by structuring and layering techniques, but in some cases, even after such reductions, the problems to be analyzed remain inherently complex and cannot be further reduced without losing essential features.

As one example, consider the window protocol described in Chapter 7. It is a simple protocol, with no obvious further simplification. In its basic form this protocol is well within the range of the full search method. As illustrated in Chapter 14, however, the complexity of the search goes up dramatically if the assumptions about channel behavior are relaxed and can make a full search impossible.

11.3.2 CONTROLLED PARTIAL SEARCH

If the state space is larger than the available memory can accommodate, the exhaustive search strategy discussed above effectively reduces to a partial search, without guaranteeing that the most important parts of the protocol are inspected. This observation has led to the development of a new class of algorithms that specifically try to exploit the benefits of a partial search. They are based on the premise that in most cases of practical interest the maximum number of states that can be analyzed, A, is only a fraction of the total number of reachable states R. A controlled partial search, then, has the following objectives:

- ○ To analyze precisely A states, with $A = M/S$
- ○ To select these A states from the complete set of reachable states R in such a way that all major protocol functions are tested
- ○ To select the A states in such a way that the search *quality*, i.e., the probability of finding any given error, is better than the *coverage* A/R

An algorithm for the partial search looks exactly like the earlier algorithm for an exhaustive search, with only one difference: not all successor states are analyzed.

```
analyze()       /* partial search */
{       if (W is empty) return;

        q = last element from W;
        add q to A;
        if (q == error_state)
                report_error();
        else
        {       for some successor state s of q
                        if (s is not in A or W)
                        {       add s to W;
                                analyze();      /* recursive */
                        }
        }
        delete q from W;
}
```

It is interesting to note that even a random selection of successor states is superior to an uncontrolled partial search, since it guarantees that the complete state space is sampled, rather than the unknown fragment that happens to be generated first in a full search. The selection can also be based on a heuristic that favors executions that are likely to reveal design errors fast. Many different ways of organizing a controlled partial search have been studied. They include methods based on:

- ○ Depth-bounds
- ○ Scatter searches
- ○ Guided searches
- ○ Probabilistic searches
- ○ Partial orders
- ○ Random selections

We discuss the first five methods briefly below. The last method, based on random selections, is developed in the remainder of this chapter. References to more detailed descriptions of all techniques are included in the Bibliographic Notes.

DEPTH-BOUNDS

A fairly standard and simple partial search technique is the placement of a bound on the length of the execution sequences that are analyzed. It limits the search to a useful subset of behaviors, ruling out, for instance, degenerate cases of multiple overlapping executions. In the full search algorithm, for instance, it allows us to restrict the maximum size of working set `W`.

SCATTER SEARCH

In a scatter search, executions are selected that lead closer to potential deadlock states. One of the requisites of a deadlock state, for instance, is that there are no messages pending (all channels are empty). The algorithm therefore favors receive operations over send operations. The goal of the method is to increase the probability of finding errors fast.

GUIDED SEARCH

In a guided search, the state selection criterion is a cost function that is dynamically evaluated for each successor state. The cost function can be fixed, as in a scatter search, or it can be changed dynamically during the search. Not much is known about the types of cost functions, or ‘‘guiding expressions,’’ that could prove to be useful.

PROBABILISTIC SEARCH

In a probabilistic search, successor states are explored in decreasing order of their probability of occurrence. All transitions in the system are labeled, minimally with a tag that gives them a ‘‘high’’ or a ‘‘low’’ probability of occurrence, and these tags are used as the selection criteria.

PARTIAL ORDERS

The main factor that is responsible for the state space explosion problem is the large number of possible interleavings of concurrent events. As shown in Chapter 5 (page 96), not all interleavings are necessarily relevant in the search for error states.

There are several ways of exploiting partial orders. A first method is based on the definition of a heuristic for either

- *Fair progress* state exploration or
- *Maximum progress* state exploration

Both heuristics work by assigning a search priority to the protocol processes. The number of transitions that are inspected during the search is limited, with preference given to the transitions that belong to high-priority processes. Transitions in lower-priority processes are only considered if all higher-priority processes are blocked. In a *fair* progress exploration technique, the relative priority of a process is *decreased* when one of its transitions is executed during the search; in the *maximum* progress exploration technique the relative priority is *increased.*

A second, more recent, method to exploit partial orders is based on formal definitions of equivalence relations on system behavior. The goal is then to prune away that part of a search that can be *proven* to be irrelevant. (References to these and the other techniques are collected in the Bibliographic Notes.)

RANDOM SELECTIONS

In a controlled partial search based on random selections of successor states, no effort is made to predict where likely errors in the state space are to be found. We will argue below that this is not only the simplest technique to implement, but is also likely to produce the highest quality search. It is the only technique that can satisfy all three requirements for a controlled partial search that were listed at the start of this section.

DISCUSSION OF CONTROLLED PARTIAL SEARCH METHODS

The first four techniques for controlling the partial search that we discussed above have one main problem in common. All four methods try to predict where the errors in a protocol can be found. This is an inherently risky approach. As a corollary of Murphy's law, the errors are likely to hide where a designer or a validator has decided *not* to look. Next to the random selection of successor states, the techniques based on partial orders can, in principle, avoid that problem. The dependencies between processes, however, can be subtle. Consider, for instance, a system of three processes A, B, and C, where A and B interact with C, but not with each other. It would be tempting to conclude that since A and B are disjoint, all possible interleavings of their behaviors are necessarily equivalent. But, alas, this assumption is invalid. Note that the behavior of process A can depend on B's behavior indirectly through their mutual interaction with C. Every distinct interleaving of the actions of A and B can be significant in determining the outcome.

To determine mechanically, therefore, which particular interleavings can safely be ignored in state space searches can be non-trivial. An accurate assessment may well be more expensive that a full blast exhaustive search, and thus be self-defeating as an optimization technique.

A final problem with the first five methods is that, although they can reduce the size of the state space, none of these methods provides a tool for matching the size of the state space to the size of available memory. For all these methods the size of the fraction of the state space that is effectively searched can only be determined experimentally, and is protocol dependent. This means that we may have to perform many validations, with different selection criteria, before we can find the optimal one that analyzes precisely M/S states. In Section 11.4 we will develop the idea of the random selection of successor states and show that it can be used to effectively solve also this problem.

Before doing so, we discuss a final state space exploration method, also based on the random selection of successor states, but this time without any attempt to build a state space

11.3.3 RANDOM SIMULATION

The controlled partial search methods have wide applicability. There are applications, however, where also a controlled partial search becomes infeasible. One can attempt, for instance, to apply a protocol validation algorithm directly to highly-detailed, compiled code that runs in a machine. In those cases, the parameter *S*, measuring the size of a single composite system state in bytes, can range anywhere from 10^3 to 10^5 bytes of memory.

Even on a larger machine, the largest number of states that can be maintained by a partial search then drops to a few hundred system states at best, in a state space that is many orders of magnitude larger. In cases like these, the only sensible approach is to discard sets `A` and `W` from the search algorithm and to explore the state space with a random simulation or 'random walk.' The algorithm is as follows.

```
analyze()       /* random simulation */
{       q = initial state;
        while (1)       /* forever */
        {       if (q is error_state)
                {       report_error();
                        q = initial state;
                } else
                        q = a successor state of q;
        }
}
```

This technique works largely independent of the size and complexity of the system being modeled; even ‘‘infinite size’’ systems can be explored in this manner. The coverage of the method, of course, cannot be measured, though in principle an exhaustive coverage of finite state spaces is guaranteed, given a sufficient amount of time. In practice this is not a very useful guideline, since a ‘‘sufficient’’ amount of time can easily mean a century of computation time or worse. In experiments, however, Colin West was able to show that even for an immeasurably small search *coverage* the *quality*, or error-finding capability, of the search can be adequate.

The remainder of this chapter is devoted to the development and motivation of a controlled partial search technique that was named *supertrace*. An implementation in C of an exhaustive search algorithm for PROMELA, with a supertrace option for large problems, is discussed in Chapters 12 to 13.

11.4 THE SUPERTRACE ALGORITHM

Given *M* bytes of memory, how can we organize a state space search to use precisely *M* bytes, no more and no less, and perform the largest search possible within that arena? To answer that question we look in a little more detail at the memory storage methods that are traditionally used. The standard way to maintain the state space set *A* in either a full or a partial search algorithm is to use a storage technique called *hashing*. Hashing allows us to determine quickly whether or not a new state *s* is already a member of set *A* and can be discarded or is not yet in *A* and needs to be inserted. The method is to use the contents of *s* to calculate a hash value $h(s)$, which is used as an index into a lookup table of states. The table is organized as shown in Figure 11.1.

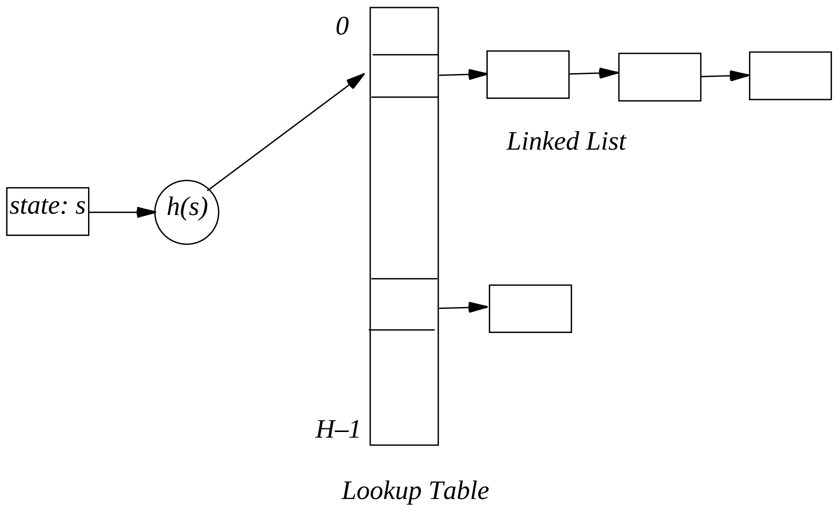

Figure 11.1 — Hash Table Lookup

Assume that we have H slots in the hash table. Hash function $h(s)$ is defined such that it returns an arbitrary value in the range $0..(H-1)$. For the same state $s \in A$, $h(s)$ always returns the same value. But there is also a possibility that two different states produce the same hash value. In the case we are studying, the hash table will have to accommodate a large number of states, which means $A > H$. The hash function will then produce the same hash value for an average of A/H different states. All states that hash to the same value are stored in a linked list that is accessible via the lookup table under the calculated index (the hash value). On the average then, when the table is full, each new state must be compared to A/H other states before it is either inserted into the linked list, or discarded as redundant. When A grows beyond the first H states, the number of comparisons required grows steadily, and the search efficiency degrades: there is a time penalty for analyzing systems of more than H states.

A typical value for H is 10^4 slots. The table itself takes up $H \times B$ bytes of memory, plus B bytes for each state that is inserted, where B is the size of an address pointer. On most machines $B = 4$, which means that a table with say 256,000 slots requires more than 1 Mbyte of overhead that can no longer be used to store states. To accommodate the largest possible state space, therefore, a small value for H is required. As shown above, however, a small value for H means a low search efficiency.

If we could somehow manage to use a very large value for H, the number of hash conflicts could be minimized and thus the speed of the search algorithm could be optimized. Let us assume we can use the full search algorithm with a value for H in the order of 10^8 slots. In a state space of up to 10^5 states we can expect to have fewer than $10^5 / 10^8$ hash conflicts, or less than one conflict in a thousand states. This means that there will rarely be more than a single state in the linked list that is connected to each slot in the hash table. But this means that we do not have to store complete state descriptions in the hash table: in all but a few cases the hash table index (the hash value) uniquely identifies a state. The only bookkeeping required is to remember if a slot in the hash table is filled or not. A single bit of storage per state will suffice. If we have M bytes of memory available, we have $8M$ bits for the state space (assuming 8 bits per byte). A 10 Mbyte machine can thus give us a state space large enough to hold 80 million states. The hash function $h(s)$ is used to calculate the position of a bit in the available memory arena M. A bit value of 1 will now indicate that the state corresponding to this hash value has been previously analyzed. The state itself is not stored.

Since no states are stored, there are no states to compare a new state against: the bit position uniquely identifies the state. The method can be expected to work well if the state space is sparse and indeed H is very large. A large value of H makes hash conflicts rare for all cases where $A < H$. Most importantly, however, when $A > H$ the hashing automatically defines a randomized partial search method that matches the coverage of the search to the available memory. The method therefore approximates an exhaustive search for smaller protocols and slowly changes into a controlled partial search method for larger protocols. For smaller protocols, however, we do not need a partial search method: we can use a traditional exhaustive search technique.

> *Supertrace is a controlled partial-search technique that is* ***only*** *meant for the validation of protocol systems that cannot be analyzed exhaustively.*

As an exhaustive search technique the supertrace algorithm would compare unfavorably with almost any other standard depth-first search method, simply because it cannot guarantee 100% coverage due to the possibility of unresolved hash conflicts (cf. Tables 13.1 and 13.2 in Chapter 13). We will show, however, that as a partial search technique, the new algorithm is superior to other methods.

HASH CONFLICTS

The overhead of the lookup table with a supertrace algorithm reduces from

$$HB + (S + B)A$$

bytes to

$$H/8$$

bytes. However, since the states are no longer stored we can no longer compensate for hash conflicts. Remarkably, this defect has a positive effect on the overload behavior of the algorithm during partial searches. Here is how it works.

If a new state s is generated and it is found that the flag is set at index $h(s)$, we must conclude that state s was analyzed before and should be ignored. When a hash conflict occurs, the above conclusion is wrong, and the search will ignore a state that should have been analyzed: the search is truncated. As $A/H \rightarrow 0$, the number of hash conflicts that will be encountered approaches zero, and the method approaches (but can never guarantee to be) a fully exhaustive search. Indeed, therefore, it is best to choose H as large as possible.

The maximum value for H that we can choose for given memory size M is $H = 8M$. Let us see how this algorithm compares to a traditional partial search.

The memory requirements are the same. The limit to the coverage of the traditional search, however, is $A = M/S$. Storing the same M/S states in the hash table of the modified algorithm, with $H = 8M$, gives a ratio

$$A/H = M/(8MS) = 1/(8S)$$

For a typical value of $S \simeq 100$, the probability of a hash conflict then approaches 10^{-3}. But the new algorithm is not restricted to a maximum of M/S states. It can analyze a maximum of H distinct states. The hash conflicts, which increases as the state space fills up, now work to scatter the states that are selected for analysis across the set of reachable states in an approximately random manner.

There are two cases to consider. For $R < M/S$, the coverage of the traditional algorithm will be the same as or slightly better than the new algorithm, since it avoids the effect of the hash conflicts. However, when $R < M/S$ we do not need a partial search algorithm at all since we can still perform an exhaustive (traditional) search in memory. The supertrace algorithm should not be used in these cases.

For problems with $R > M/S$, the coverage of the new algorithm, i.e., the total number of effectively analyzed states compared to the total number of reachable states, is substantially higher than the coverage of the traditional algorithm. For $R >> M$ it approaches $8M/R$, compared to $M/(S\ R)$ for the traditional algorithm (see also Figure 11.2).

If state description S becomes larger the traditional algorithm can analyze fewer and fewer states, but the performance of the new algorithm stays the same. If, for M fixed at 10^7 bytes of memory, S grows from 100 to 1000 bytes per state, the coverage of a traditional partial search algorithm *drops* from 10^5 to 10^4 analyzable states. The coverage of the new algorithm, however, remains constant at a maximum of $H = 8 \cdot 10^7$ analyzable states.

The effect is illustrated, for a fixed size S, in Figure 11.2. Increasing S is equivalent to moving the dotted and the dashed line to the left: the behavior of the traditional algorithm changes, but the behavior of the supertrace algorithm remains constant.

For state spaces that are larger than an exhaustive search algorithm can accommodate, the traditional method breaks down very rapidly, its coverage dropping by a factor of ten for every tenfold increase in the number of reachable states. The coverage of the new algorithm is substantially better.

When $A \rightarrow R$, A is the same order of magnitude as H, which means that a large fraction

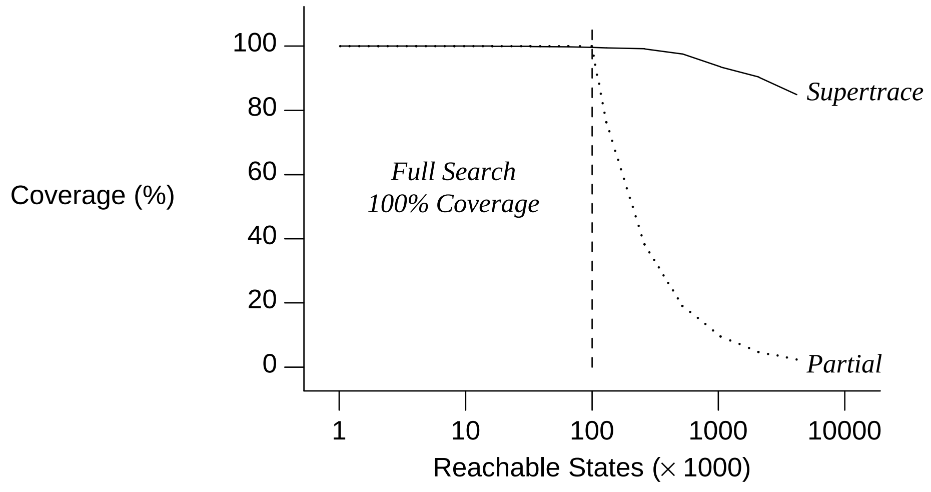

Figure 11.2 — Comparison of Two Algorithms

of the state space can still be analyzed, the hash conflicts acting as a random pruning that scatters the search over the oversized state space. For still larger protocols with $A > H$ the coverage of the search approaches H/A, or $8M/R$.

MULTIPLE HASHINGS

The hash functions helps us to make a fast random selection of states from a large state space, and thus implements an efficient controlled partial search. Assume a hypothetical 10 Mbytes of memory available for the search and a state space of 800 million states of 100 bytes each. The coverage of all traditional search methods, except supertrace and random simulation, is limited to the analysis of $10^7/100$ in $8 \cdot 10^8$ states or 0.0125 %. A single run of supertrace would give a maximum coverage of $8 \cdot 10^7/8 \cdot 10^8$ or 10%. The question is: Can we ever achieve a still better coverage with the same system constraints? Surprisingly, the answer for the supertrace algorithm is: Yes.

The hash function can be used as a parameter in repeated searches. Suppose the first search with hash function $H1$ selected 80 million states are random from the 800 million reachable states. A second search with a different hash function $H2$ will also select 80 million states, but it will make a different selection. We may expect that there will be a 10% overlap between the two state sets, but the combined coverage of the two searches has now gone up to $80 + 72$ million states out of the 800 million candidates, or 19%. Continuing this process, we can in theory get arbitrarily close to a coverage of 100% of the state space, provided that a sufficient number of independent hash functions can be found.

The validator developed in Chapter 13 uses this principle to increase the coverage of searches. It uses two hash functions in each single run.

11.5 DETECTING NON-PROGRESS CYCLES

So far, we have only discussed the validation of state properties, using a straightforward reachability analysis algorithm. The complexity of the algorithm, even for this simple case, is in PSPACE. We have therefore made a deliberate effort to find the fastest, most frugal implementation so that the range of problems we can apply it to is as large as possible. But we are not done. There are other properties that we may be interested in proving, specifically for PROMELA validation models. If, as we have argued above, the efficiency of a straight reachability analysis is a concern, the efficiency of the more subtle types of validation is crucial.

A straightforward check for non-progress conditions could be based on the construction and inspection of all strongly connected components in the reachability graph that is implicitly defined by the state space of the system being analyzed. This approach, though commonly used, fails when the state space is too large to be stored completely. Here we explore a different option that has a modest expense and, most importantly, that can be used in combination with a supertrace algorithm to do partial validations of very large systems.

Our first problem is to detect cycles in the reachability graph that do not pass through any states marked as progress-states. The algorithm we develop is only for identifying non-progress cycles. We will not try to combine it with a simultaneous search for assertion violations and improper terminations. A first attempt to find the non-progress cycles is to perform a standard depth-first reachability analysis where all sequences are truncated when a progress-state is reached. That is, progress-states are treated as if they have no successors. All cycles that can be constructed in a search of this type, must be non-progress cycles. The size of the state space that is created in this search is at most equal to the size of a straight depth-first search. It is likely to be smaller due to the truncations at progress states.

> To see how this may be implemented, refer to the algorithm for the full state space search given in Section 11.3.1. A cycle is detected if the depth-first search reaches a state that is already in work-set `W`, assuming that states are extracted from set `W` in last-in first-out order.

The flaw of this method is that it does not allow us to detect cycles that do not pass through the initial system state. There may well be a cyclic execution sequence (as defined in Chapter 6) that first passes through a finite number of states, some of which may be marked as progress states, before entering a cycle of strictly non-progress states. This observation, however, immediately leads to a new algorithm that does work.

A non-progress cycle might start in any reachable system state. So we must inspect two distinct state spaces: one created by the original depth-first search, and one that is created when transitions from progress states are disabled. The task of our search algorithm is to inspect every possible prefix of a cyclic sequence in the original state space and see if it can be continued into a cycle in the second state space. The implementation is simple. We can add a two-state demon to our validation model that defines in which mode the search will operate, as follows

```
proctype demon() { bit magic = 0; magic = 1 }
```

The initial state of demon process is just before the assignment, with variable `magic` equal to zero. The second, and final, state of the demon is immediately after the assignment, with `magic` equal to one. The demon process can switch from the initial state to the final state nondeterministically, and once it has switched it cannot go back. The value of variable `magic` defines in which mode the search is performed. When `magic` is zero, a normal depth-first search is performed, without any error checking. When `magic` is one, all transitions that originate in progress states are disabled. All subsequent execution sequences should be terminating. If there is any cycle of states that are reachable while `magic` is one, it must be a non-progress cycle.

> The value of `magic` can only change once in any given execution sequence, and it can only change from zero to one. Let us assume that, after `magic` has changed value, a cycle of states is detected that is not a non-progress cycle. By definition that cycle contains at least one transition originating at a progress state. This transition can only occur when `magic` is equal to zero. This means that the value of variable `magic` changes from zero to one and back at least once each time through the cycle. This contradicts the earlier observation that `magic` only changes value once.

The algorithm we have constructed further has the property that if any non-progress cycle exists, at least one will be detected. To prove that, let us assume that there exists a reachable strongly connected component that contains only non-progress states. (A strongly connected component is any set of states in which every member can reach every other member of the set via one or more transitions.)

> The algorithm generates two copies of every reachable state; there is one copy in which `magic` is equal to zero, and one copy with `magic` equal to one. There is a transition from the first copy to the second that corresponds to the one transition that the demon process can make. Consider the case where no state from the strongly connected component has been generated with `magic` equal to one. Consider the first such state that is generated. Since the strongly connected component is assumed to be reachable, and the transition of the demon process is always executable, this must happen at some point in the search. Call this state the *seed* state. The depth-first search tree that is rooted at the seed has all states from the strongly connected component as successors, including itself (by the definition of a strongly connected component). Since the seed state is also reachable from itself (by the same definition) via non-progress states only (by our original assumption), it must be revisited. The moment the seed is revisited, a cycle is detected. By our earlier proof, that cycle must be a non-progress cycle.

There can, of course, be many different paths through a strongly connected component, each one of which may represent a different type of non-progress cycle. The algorithm above does not guarantee that all variants are detectable in a single execution of the search. It does guarantee that at least one variant is detected. If no non-progress cycles are detected, therefore, we can be certain that none exist.

Figure 11.3 illustrates how a difficult case of a non-progress cycle is detected. The circles represent system states and the arrows represent transitions. The states are

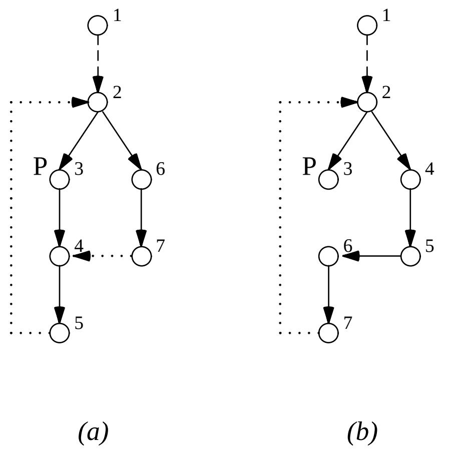

Figure 11.3 — Detection of a Non-Progress Cycle

numbered in the order in which they are visited during a search. The state marked P is a progress-state. State 1 is the initial system state. The dashed line from state 1 to state 2 is an arbitrary execution sequence. The dotted lines indicate state matches. Remember that after the creation of every state our depth-first search algorithm checks to see if the state was created before. If a match is found the search is truncated. If the match occurs in work set W, i.e., on the stack, a cycle is detected.

In Figure 11.3a a fragment of the state space is shown as it would be created in a normal depth-first search. In Figure 11.3b the same states are shown after the transition of the demon process to the state in which transitions starting at progress states are disabled. The numbers indicate the order in which states are visited.

Before the transition of the demon process, in Figure 11.3a, just one cycle is detected by the normal depth-first search method. It is detected when the fifth state visited is found to match the second state, which is on the stack. The loop is benign, since it contains the progress-state. The search continues, after removing states 4, 3, and 2 from the stack, with the new state 6. The seventh state visited matches the fourth one, and the search is completed. The last match does not produce a cycle, because state 4 is no longer on the stack.

The non-progress loop through the states marked 2, 6, 7, 4, and 5 therefore remains undetected in the standard depth-first search. After the transition of the demon process, all transitions from the state marked P are disabled. This means that the states are now visited in the order indicated in Figure 11.3b. The first, harmless, cycle can now no longer be constructed, but the second cycle can, and is correctly detected.

With the addition of a simple two-state demon process, the algorithm is trivial to implement. An implementation in C is given in Appendix E. Its expense is a doubling of the time and space requirements. Perhaps the most important advantage of the algorithm, however, is that it can be used in combination with any controlled partial search method. Specifically it can be used with a bit state space technique, as used in the supertrace algorithm.

11.6 DETECTING ACCEPTANCE CYCLES

The detection of acceptance cycles (see Chapter 6, page 118) is substantially harder than the detection of non-progress cycles, discussed in the last section. This time, all execution cycles that pass through at least one acceptance state must be detected. We are interested in finding an algorithm that continues to work with supertrace, so that its application to very large problems is not excluded.

The following algorithm is due to Mihalis Yannakakis (see the Bibliographic Notes). The expense of the algorithm is at worst a doubling of the time and space requirements of the basic search. We conduct a depth-first search with two state spaces instead of one (i.e., two copies of set `A`). Call the second state space set `C`. When no acceptance state is encountered, set `C` remains unused, and the search is precisely the same as before. For every acceptance state that is removed from work set `W` and added to set `A` (i.e., after all its successor states have been visited) the algorithm switches sets `A` and `C` and begins a new search. Call the acceptance state the *seed* of that search. If at any time during this search the seed state can be revisited, an acceptance cycle is found, and an error can be declared (i.e., the temporal claim is satisfied, which means that an undesirable behavior is possible). When no such error is found, the second search terminates when all successors of the seed have been added to set `C`. At this point, sets `A` and `C` are swapped again, and the depth-first search continues as before.

No state will be visited more than twice in this search, once in set `A` and once in set `C`. It is not hard to convince ourselves that any cycle found by this algorithm is necessarily an acceptance cycle. It is harder to show, however, that in the absence of hash collisions any acceptance cycle that exists is also found.

> Assume that there are acceptance states that belong to one or more strongly connected components in the reachability graph. If all states in the reachability graph are numbered in the order in which they are added to set `A`, consider the acceptance state with the lowest number. Call that state the *seed*. Because the seed belongs to a strongly connected component it is reachable from itself. The acceptance cycle is detected if and only if none of the intermediate states along that path have been added to set `C` before the seed. If there is any such state, however, it necessarily has a lower search number than the seed. All states along the path we are interested in belong, by definition, to the same strongly connected component as the seed. If any one of those states has a lower search number, and was added to set `C` before, its complete set of successors must have been analyzed as well, *before* we reach the seed. This means that all these successor states have a lower search number than the seed. The set of successors, however, includes the seed, because they all belong the same strongly connected component. This means that this is not the first visit to the seed, which contradicts our assumption.

11.7 CHECKING TEMPORAL CLAIMS

To check temporal claims, as they were defined in Chapter 6, every state transition in the reachability graph for the original system, without the temporal claim, must be matched with a state transition in the finite state machine that represents the temporal claim.

Fortunately, this requirement is relatively easy to meet, and compatible with supertrace. After the generation of a successor state during the standard search we include one extra test, a forced transition of the temporal claim process to a new state. If such a transition cannot be made, the search can be truncated as if a state match was found. It means that the undesirable behavior that is expressed in the claim cannot be realized after the last transition in the system is made. The details of an implementation in C are given in Appendix E. In the best possible case, if no transition from the initial system state can be matched by a transition in the claim, the time and space requirements of the new algorithm reduce to almost zero. In the worst possible case, however, the size of the state space is multiplied by the number of reachable states of the claim.

The worst-case expense of the validation of temporal claims increases linearly with the size of the claim, measured as the number of states of the extended finite state machine that defines the claim. With the discussion of the last two sections, we can compare the complexity of the validation of different types of PROMELA correctness requirements. The minimum expense is incurred for the validation of properties of states, such as assertions and improper terminations. It can be twice as hard to check for non-progress properties, and $2N$ times as hard to check a temporal claim of N states.

If we turn this argument around, we can say that, with the same search quality, for the validation of state properties the system can be $2N$ times larger than for the validation of temporal claims. It is therefore important that a validation system, such as PROMELA, allows us to validate each type of property separately,[1] so that the simpler requirements do not incur the expense of the more complicated ones. The system size determines in all cases precisely which types of validation of a given quality can be performed. If the best search quality that can be realized for a given system is insufficient, we can do two things:

- ○ Express the correctness requirement differently so that it can be checked more efficiently
- ○ Express the system behavior differently in an effort to reduce its final size

We discuss the second method in more detail below.

1. Temporal claims could be used to express state properties, and even non-progress conditions, and could therefore be used as a single default mechanism for specifying correctness requirements.

11.8 COMPLEXITY MANAGEMENT

The validation of protocol systems that generate up to a few hundred thousand states is well within reach of of the automated validation systems we have described. The validation of larger systems, however, can be a substantial challenge in the management of complexity. It could well be claimed that complexity management itself is the most important issue in the design of a validation strategy. In this section we review some of the main issues.

The discussion of partial search techniques (page 224) was the primary motivation for the complexity management technique that we have chosen as the basis of the supertrace algorithm. Two other important issues remain to be discussed:

- ○ Reduction methods
- ○ Incremental composition

Both methods are applied before a state space search is started, instead of taking effect during a search as in the partial searches. They therefore apply to all search methods, from fully exhaustive searches to random walks. We discuss them in more detail below.

REDUCTION METHODS

The design of a validation model trivially determines the complexity of the validation that is to be performed. If protocol layering and structuring techniques are applied, it is often possible to separate, without loss of generality, the validation of multiple orthogonal protocol functions. An example of that is given in Chapter 14, where the validation of the flow control protocol from Chapter 7 is separated from the validation of the session control protocol.

In Chapter 8, Section 8.9, we discussed a technique to further reduce the complexity of a validation model by systematic generalizations that do not affect the scope of a validation. Similar ideas have been based on the notion of ‘‘protocol projections,’’ as first described by Lam and Shankar.

In some cases, however, it may still be hard or impossible to find the ideal behavior preserving reduction. In those cases we have one more complexity management option. There are many modeling parameters that control the range of possible behaviors defined by a model. The determining factors for the complexity of PROMELA models, for instance, are the number of processes, message queues, and variables, and the size of the message queues. Decreasing the number of slots in message queues can reduce the maximum amount of asynchrony in a concurrent system and dramatically decrease the number of reachable composite system states, without necessarily decreasing its scope. A validation model often can be analyzed exhaustively by restricting some of these parameters. The model, of course, becomes a partial one when the parameter settings are decreased. This means that we often have a choice between performing an exhaustive search for a partial model, or a partial search for a full model. Which approach is the most appropriate naturally depends on the problem being studied.

INCREMENTAL COMPOSITION

In the reachability analysis algorithms we have discussed up to this point, we have assumed that all asynchronous processes that contribute to the global behavior of the protocol are combined in a single step in the generation of the global system state space. In some cases, an *incremental composition* method can be used to reduce the size of the state space that is being constructed. (See Algorithm 8.3, and see also Chapter 8, Section 8.7.) With this method we first generate the set of all reachable composite system states of two or more of the protocol processes. This partial state space is then reduced by standard state machine minimization and then composed with the remaining processes, again in an incremental fashion.

In a typical application of this method, at each step two separate state machines are replaced by one state machine, which is reduced in size before it is combined with the other machines. To work, this method obviously requires that the validation model consist of *more* than two state machines (asynchronous processes). It further relies crucially on the user's ability to find precisely those combinations of state machines that can produce the greatest reductions. The reduction is meant to remove behavior that is internal to the machines that are combined. It reduces the combined machine to the *external* behavior of the machines that were collapsed.

This means that the method works best if it is applied to machines that are tightly coupled (that is, they exchange a lot of messages) and that are relatively independent of the rest of the system. If the user, by mistake, combines two machines that are disjoint, the state space explosion problem is worsened: effectively the two machines would be replaced by an irreducible composite state machine that defines the complete Cartesian product of all states in the two individual state machines. A large fraction of those states can be recognized as unreachable only when the remaining composition steps are taken.

Several researchers have implemented the incremental composition method and applied it to validation models that only use *rendezvous* communications. The advantage here is that the rendezvous points can disappear in the reduction steps. It is not clear if the method can still be effective when it is applied to systems such as PROMELA that allow asynchronous, buffered message exchanges. In these cases the internal buffers may complicate the minimization process.

11.9 BOUNDEDNESS OF PROMELA MODELS

It is not immediately obvious that any given PROMELA model can be reduced to a finite state system and validated with the algorithms we have discussed in this chapter. A PROMELA validation model allows an arbitrary number of process instantiations and an arbitrary number of message queues to be created. The following program, for instance, is valid in PROMELA.

```
proctype A()
{       chan Ain = [1024] of { int, int };

        do
        :: run A()
        od
}
init { run A() }
```

To simulate the execution of this model would require an infinite amount of memory and an infinite length of time. Any real execution of the model, however, can only take place on a finite machine. Most models are therefore finite by design, and it can even be argued that the possibility of infinite growth is a design error.

PROMELA restricts the maximum number of processes and message queues that can be created. The precise limit is not defined. At some point during the execution of the example program above the `run` statement will become unexecutable and block the last process that was created. Every PROMELA model is therefore by definition a finite state system and can be analyzed with a standard reachability analysis algorithm. Each process has a fixed number of states, each message queue has a fixed number of slots, and the range of all variables used in the system is fixed. When the model is executed, it can only reach a finite number of possible states. At some point in the execution of process `A()` above, for instance, the `run` statement becomes unexecutable and prohibits further growth.

In Chapters 12 and 13 we discuss the implementation of a program that converts PROMELA specifications into the required finite state models. The program implements all three basic search modes we have discussed: random simulation, bit state space search, and the full state space search.

11.10 SUMMARY

Given a new, carefully designed protocol, how can we gain confidence that it will not fail in some unexpected way? For instance, we may want to prove that the protocol is robust under adverse channel behavior, or we may want to show that certain undesirable events, such as system deadlocks, cannot occur. The methods we have described in this chapter are based on the verification of correctness requirements that can be expressed as system invariants: properties that remain invariantly true for all possible executions of the system.

The manual proof method we gave is based on an exhaustive inspection of state transitions, and the automated variant is based on an exhaustive inspection of system states. The manual validation procedure can be expected to work for systems of up to ten or twenty state transitions, but is largely independent of the number of reachable states. The credibility of these manual proofs, however, is at best inversely proportional to their length.

The automated procedure does not have this drawback, but its applicability depends crucially on the number of reachable system states. For relatively small systems, up to approximately 10^5 reachable system states, we can apply a fully exhaustive state space search. The purpose of the exhaustive search is to show the *absence* of errors. If it can be completed without reporting any errors, it is certain that the protocol cannot violate any of the correctness criteria.

For larger systems, up to approximately 10^8 reachable system states, the best validation that can be performed is a controlled, partial search. The purpose of a partial search is to show the *presence* of errors, not the absence. The partial search is designed in such a way that if it is applied to a protocol that contains an error, it optimizes our chances of exposing it within the constraints of the machine on which the validation algorithm is run. We have discussed three different ways of achieving this objective:

- □ Using search heuristics to restrict the partial search to system states that are likely to contain the errors.
- □ Using a hashing technique that dramatically increases the number of system states that can be manipulated.
- □ Using reduction methods to simplify validation models before they are subjected to a search.

The first method has the disadvantage that it tries to predict where the errors are likely to be, an inherently dangerous strategy. The second strategy does not have this problem, and turns out to be the only one that allows us to match the scope of the analysis to the constraints of the system on which the validation algorithm is executed, whatever they may be. The application to PROMELA is elaborated in the next two chapters.

For exceptionally large validation problems, finally, the only workable validation method is a random simulation that tries to explore as many system states as possible, trying to home in on those states that can violate the system invariants.

EXERCISES

11-1. Use the manual proof technique to show that the alternating bit protocol preserves the correctness of the window protocol invariant for a window size of one. □

11-2. Modify the partial search algorithm to include a maximum or fair state space exploration heuristic. □

11-3. The following solution to Dijkstra’s mutual exclusion problem (see Chapter 2, and Dijkstra [1965]) appeared in the *Communications of the ACM*, Hyman [1966]. It is reproduced here as it was published (in pseudo Algol).

```
Boolean array b(0;1) integer k, i, j,
comment process i, with i either 0 or 1;
C0:  b(i) := false;
C1:  if k != i then begin
C2:  if not (b(j) then go to C2;
     else k := i; go to C1 end;
     else critical section;
     b(i) := true;
     remainder of program;
     go to C0;
```

```
11      end
```

Show that the solution is incorrect by modeling the solution in PROMELA, and performing an automated validation with one of the reachability analysis algorithms discussed in this chapter. □

11-4. The reachability analysis algorithms we have considered verify the observance of system *state* invariants. Consider possible extensions to the basic full-search algorithm to check for properties of system state *sequences*: paths through the global state space. What extensions are necessary, for instance, to be able to prove or disprove for the alternating bit protocol that there is no infinite sequence of transitions in which the 1-bit sequence number remains unchanged? □

11-5. There are algorithms that can find all strongly connected components in a directed cyclic graph, e.g., Aho, Hopcroft & Ullman [1974, p. 192]. Consider how such an algorithm could be used to extend the capabilities of the reachability analyzers, what the cost in added time and space complexity would be, and how these extensions would be affected by partial searching. □

BIBLIOGRAPHIC NOTES

The manual proof technique based on system invariants is due to Krogdahl [1978] and Knuth [1981]. The proof of the window invariant discussed here was also first given in Knuth [1981]. The method was also used more recently in Brown, Gouda, and Miller [1989]. Gouda’s manual validation method based on state invariants and well-founded formulas is inspired by the seminal paper Floyd [1967].

Several other attempts have been made to develop automated protocol validation tools that are not based on reachability analysis. Early experience with some automated versions of these tools was reported in Schwabe [1981] and Sunshine and Smallberg [1982]. A promising new manual proof theory is based on the Oxford specification language Z. See for instance Duke, Hayes, King and Rose [1988], Duke, Hayes and Rose [1988], and Hayes, Mowbray and Rose [1989].

Work on automated protocol validation methods was pioneered by Brand and Joyner [1978], Hajek [1978], West and Zafiropulo [1978], West [1978], Zafiropulo [1978], and Razouk and Estrin [1980]. The work of Colin West and Pitro Zafiropulo [1978] provided a first demonstration that with automated tools even protocols that have withstood the scrutiny of years of development in an international standardization organization can, within a few seconds of computer time, be shown to be flawed. In this case, the protocol was the CCITT Recommendation X.21, and the validation tool was a straightforward implementation of the validation theory developed in Zafiropulo [1978]. Important subsequent work was reported in Zafiropulo et al. [1980], Rubin and West [1982]. Excellent surveys of the work on protocol validation can be found in IFIP conference proceedings such as IFIP [1983], or the April 1980 special issue on ‘‘Computer Network Architectures and Protocols’’ of the *IEEE Transactions on Communications*, which contains the standard reference Bochmann and Sunshine [1980].

There are many results on the computational complexity of the validation task of a communicating finite state machine model, see for instance Cunha and Maibaum [1981], Brand and Zafiropulo [1983], Apt and Kozen [1986], Reif and Smolka [1988]. In general, the problem of finding deadlocks in a system of communicating finite state machines is PSPACE complete at best, and becomes formally undecidable when the message channels are unbounded.

This result, of course, does not mean that any further analysis of finite state machine models is pointless. It does mean that the complexity of a protocol validation algorithm is a main concern. These algorithms can carry no more overhead than strictly necessary to solve the problem. Though it can be tempting to extend a search algorithm to capture more subtle features, it is generally ill-advised to do so if the method is to survive application to problems of a realistic size.

The necessity of partial search techniques was first described in West [1986b] and in Holzmann [1985, 1987a]. An overview of a range of search heuristics that have since been invented for partial searches can be found in Lin, Chu and Liu [1987]. The random state space exploration method as first studied by Colin West [1986b, 1989]. Probabilistic partial search techniques were described by Maxemchuck and Sabnani [1987]. A scatter search technique with guiding expressions was introduced in Pageot and Jard [1988]. A heuristic for partial orders was first suggested in Holzmann [1985]. Several more formal approaches have been investigated in the last few years, e.g., Probst [1990], Valmari [1990] and Godefroid [1990]. The fair progress state exploration heuristic was first suggested in Rubin and West [1982], and further explored in Gouda and Han [1985]. Maximum progress state exploration was described in Gouda and Yu [1984]. The concept of ‘‘protocol projections’’ was introduced in Lam and Shankar [1984].

The bit state space technique was first described in Holzmann [1987b] and elaborated in Holzmann [1988]. The hashing technique is based on a much older technique called ‘‘scatter storage,’’ described in Morris [1968], and applied in McIlroy [1982]. The bit state space search technique can easily be applied to all FSM based models, e.g., Rafiq and Ansart [1983], Estelle, e.g., Richier et al. [1987], the S/R model, Aggarwal, Kurshan and Sharma [1983], and Petri Net models, e.g., Bourguet [1986], to name just a few.

The extension of the exhaustive search algorithm with assertion proving capabilities was described in Holzmann [1987a]. An comparison of search algorithms based on reachability analysis appeared in Holzmann [1990].

The algorithm for the detection of non-progress cycles has not been published before. The algorithm for the detection of acceptance cycles, for instance in the context of a temporal claim, is due to Mihalis Yannakakis of AT&T Bell Laboratories. It was first described in Courcoubetis, Vardi, Wolper, and Yannakakis [1990]. A standard algorithm for detecting strongly connected components in a graph can be found in Aho, Hopcroft and Ullman [1974].

The application of pure finite state models to the protocol validation problem can be found in, for instance, Brand and Zafiropulo [1983], Bochmann [1983], or Knudsen [1983].

Many interesting approaches to the protocol validation problem could not be discussed here. In particular this goes for the work on the S/R model and omega regular languages, Aggarwal, Kurshan and Sharma [1983], Har'El and Kurshan [1990], and model checking systems for circuit verification, e.g., Clarke [1982], Browne, Clarke, Dill, and Mishra [1986].