ALGORITHMS FOR AUTOMATED PROTOCOL VERIFICATION

Gerard J. Holzmann

Gerard J. Holzmann *is a member of the technical staff in the Computing Techniques Research Department in the Computing Science Research Center at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include distributed systems, protocol validation, operating systems, computer graphics, and digital photography. He received an M.Sc. degree in electrical engineering and a Ph.D. degree in technical sciences, both from Delft University of Technology. He joined AT&T in 1980.*

This paper studies the four basic types of algorithm that, over the last 10 years, have been developed for the automated verification of the logical consistency of data communication protocols. The algorithms are compared on memory usage, CPU time requirements, and the quality of the search for errors. It is shown that the best algorithm, according to above criteria, can be improved further in a significant way, by avoiding a known performance bottleneck. The algorithm derived in this manner works in a fixed-size memory arena (it will never run out of memory), it is up to 2 orders of magnitude faster than the previous methods, and it has superior coverage of the state space when analyzing large protocol systems. The algorithm is the first for which the search efficiency (the number of states analyzed per second) does not depend on the size of the state space: there is no time penalty for analyzing very large state spaces. The practicality of the new algorithm has been tested in the verification of portions of AT&T's 5ESS® switch. The models analyzed in these tests generated up to 250 million composite system states, that could be analyzed effectively in an hour's worth of CPU time on a large mainframe computer.

Introduction

The analysis of communication protocols by the exhaustive inspection of reachable composite system states in a finite state machine model is a relatively straightforward and well-understood procedure. It would seem that for an exhaustive analysis we have to inspect at most a number of composite system states that is equal to the Cartesian product of the number of states in each independently

Table I. Parameters

Symbol	Description	Typical value
M	Bytes of memory available for the complete search	10^7 bytes
R	Total number of reachable composite system states	10^8 states
S	Bytes of memory required to store one state from R	10^2 bytes
L	Average number of successor states per state in R	2 states
E	Search efficiency: total CPU time required per state	10^{-2} seconds
D	Average length of one acyclic execution sequence	10^2 states

executing entity of the protocol. Depending on the size of the original system, this may or may not be prohibitively expensive. There are, however, delicate trade-offs to be made that give us a choice of algorithms that may either inspect more or fewer system states than the limit stated above. The trade-offs are between the coverage, or the completeness, of the search and its CPU requirements in terms of run time and memory usage.

In the comparison of search algorithms the parameters in Table I are used. The last column gives an order of magnitude value for each parameter, assuming a medium-size computer (e.g., a Digital Equipment Corporation VAX®-750 computer) and a test protocol of a realistic size. Times are given in seconds, sizes are in bytes, and the length of an execution sequence is measured by the number of states through which it passes.

The values in Table I will be used for comparing the performance of search algorithms under realistic conditions. The value for M needs little justification: it is determined by available hardware. The values for S, R, L and D are conservative estimates based on measurements of a range of different protocols.[1,2] As one specific example, in a verification of the IEEE 802.2 logical link control protocol that we performed,[3] the parameter values were: $S = 150$, $R \approx 10^9$, $L \approx 2.5$, and $D \approx 10^3$. The value for E in Table I is an average based on measurements of typical search algorithms, and matches the performance of at least one generally available implementation of a traditional algorithm.[4] No better values for E have been reported in the literature, though not infrequently inferior values are claimed to be improvements over existing methods.[5,6] In Table II we calculate the cost of the basic operations in a traditional search algorithm. That calculation also leads to the same value for E given in Table I.

The evaluation of search algorithms that follows is machine independent. Obviously, any algorithm will run faster on a bigger machine and slower on a PC. What we are interested in here is, however, the relative performance. Therefore all algorithms are compared for their performance on the same machine with one given set of constraints.

The Problem

Figure 1 gives an overview of the protocol verification problem. The original specification of a protocol is the result of design process and in its final form, for instance, as presented in a standards document such as Reference 3. The problem of verifying that an implementation of the protocol conforms to the original specification is a separate area of research that is outside the scope of this paper. The protocol verification problem considered here is the problem of verifying that the original specification is itself logically consistent. If, for instance, the specification has a design error, an implementation is expected to *pass* a conformance test if it contains the same error. The conformance test should fail only if implementation and specification differ. A verification for the logical consistency of the protocol, however, must reveal the design error.

To verify logical consistency, an extended finite state machine (xFSM) model of the protocol is constructed. The model will generally specify a set of asynchronous, communicating finite state machines. The

Table II. Cost of State Space Maintenance (Seconds)

Operation	Number of bytes	Cost per byte	Typical value
Calculating the hash	S	10^{-6}	10^{-4}
Comparing states	$A \times S/H$	10^{-5}	10^{-2}
Inserting new states	S	10^{-5}	10^{-3}

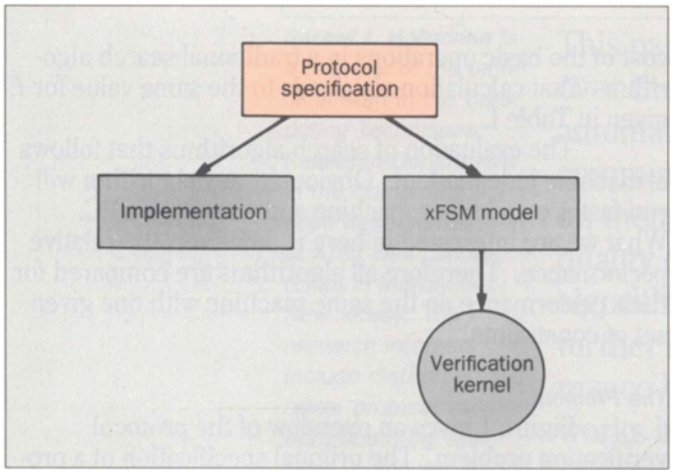

Figure 1. Specification, implementation, and verification.

model can contain interaction primitives, such as send and receive operations on finite message queues, or it can leave it to the model builder to construct primitives from simpler constructs. For the remainder of the paper it is irrelevant which particular constructs are used. It is important only that the model defines behavior in terms of states and state transitions.

The individual behavior of one of a set of asynchronous finite state machines in the model is defined by a finite set of local machine states and local state transitions. The local machine states will be called "process states." The system as a whole is defined, minimally, by the composite of all individual process states and the combination of all simultaneously enabled local state transitions. We will use the term "state" as shorthand for "composite system state" from here on. Where this can cause confusion we will use the terms "process state" or "system state." Given an initial state for each machine in the system, the sets of process states and system states can be divided into two disjoint classes: reachable states and unreachable states. Normally it is required that the model not contain any unreachable *process* 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.

The formal model can be subjected to an exhaustive reachability analysis to determine which states are reachable and which are not. Every reachable state, and possibly every sequence of reachable states, can be checked for a given set of correctness criteria (i.e., system invariants). These criteria can be general safeness conditions that must hold for any protocol, such as the absence of deadlocks or buffer overruns, or they can be protocol-specific liveness requirements such as the proper working of a message retransmission discipline.

The task of building a finite state machine model can be separated from the task of performing the actual reachability analysis. In the effort to construct a tractable model we can exploit symmetries in the specification, apply reduction rules, and projection methods.[7] But once the FSM model has been constructed, the reachability analysis task is always the same, and largely independent of the formalism and methodology that was used to derive the model. The reachability analysis task is performed by the *verification kernel* from Figure 1. It can be studied as a separate problem. This, then, is the topic of this paper: the problem of designing the fastest possible verification kernel for reachability analyses.

Search Algorithms

We define four basic types of search algorithms. They are listed in decreasing order of memory usage. For each algorithm a sample implementation is given in pseudo-code. The first three algorithms require an initialization routine:

```
init()
{
        W = { initial_state };
        A = { };
}
```

The routine initializes two sets: a working set of system states to be analyzed, called W, and a larger set of analyzed states, called A. Set A is also referred to as the "system state space." Ideally, it will include all reachable system states R when the algorithm terminates.

Type 1: Exhaustive Search. An unmodified exhaustive search strategy, also called a *perturbation analysis*, was one of the first methods tried for programming the reachability analysis[8,9] and is still in use.[10–14] It attempts to build the complete system state space containing all reachable states R. If connectivity information also is stored it can be used for the detection and analysis of execution cycles.[14]

```
type_1()     /* exhaustive search */
{
   while (W nonempty)
   { q = element from W;
     if (q is error_state)
         report_error();
     else
     {   for each successor state s of q
           if (s is not in A or W)
               add s to W;
     }
     delete q from W;
     add q to A;
     }
}
```

Practical bounds to specifically the amount of available memory restrict the applicability of this type of search algorithm. These bounds can be quantified as follows. To make the comparison fair, it is assumed that no connectivity information is stored (only an algorithm of Type 1 can truly use this information in analyses). To include it in the formula below, replace every occurrence of S with $S + B \times L$, with B the number of bytes required to store a link. Typically $B = 4$, the size of an address pointer on our reference machine.

To complete a Type 1 algorithm we need $S \times R$ bytes of memory, or

$$M \geq S \times R$$

The maximum number of states A that effectively can be analyzed then is

$$A \leq \frac{M}{S}$$

and this search will take

$$T \geq \frac{E \times M}{S} \text{ seconds}$$

For the values from Table I we have

$$A \approx 10^5 \text{ states} \quad \text{and} \quad T \approx 10^3 \text{ seconds}$$

It is therefore reasonable to analyze small- to medium-size protocols exhaustively with a Type 1 algorithm. Larger models are more challenging. Specifically, if $A << R$ an algorithm of Type 1 cannot run to completion. The machine will simply run out of memory. For a protocol of realistic size, then, the algorithm can perform only a partial search, without, however, guaranteeing that the most important parts of the protocol are inspected. This observation leads to the next class of algorithms.

Type 2: Partial Search. A partial search strategy[4,15,16] tries to optimize the coverage of the search for given constraints in memory usage and run time. It is based on the premise that in most cases of practical interest A, the maximum number of states that can be analyzed, is only a fraction of R, the total number of reachable states. The partial search 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 probability of finding any given error is *better* than the coverage A/R.

A Type 2 algorithm is almost the same as a Type 1 algorithm, with one crucial difference: not all successor

states for each state are analyzed.

```
type_2()     /* partial search */
{
   while (W nonempty)
   { q = element from W;
     if (q is error_state)
        report_error();
     else  {
        for some successor states s of q
         if (s is not in A or W)
             add s to W;
     }
     delete q from W;
     add q to A;
   }
}
```


The selection criterion for choosing states to be analyzed controls the eventual size of set A and hence the coverage of the search A/R. It also controls the chance of finding errors. Possible selection criteria for organizing the partial search are discussed below under "Organizing a Type 2 Search."

The time and space requirements of a Type 2 search are the same as the effective requirements of a Type 1 search, but this time the fraction of the state space that is searched is under user control.

Type 3: Stack Search. A stack search strategy[17,18] attempts to minimize memory usage while maintaining full coverage of the state space, at the expense of run time. This is necessarily a depth-first search method. The state space consists of a single execution sequence that is maintained on a stack. In the sample code below, the stack is maintained in set W. Set A is not used.

```
type_3()     /* stack search */
{    q = the last element in W
     if (q is error_state)
       report_error();
     else
     {   for each successor state s of q
           if (s is not in W)
           {  add s to W;
              type_3();  /* recursion */
              delete s from W
           }
      }
}
```

The maximum amount of memory required depends on the length of the longest acyclic execution sequence D_{max}. We can write the following expression:

$$M \geq S \times D_{max} \quad \text{and} \quad T \geq E \times R^*$$

The stack search guarantees an exhaustive search of all system states. Not all execution cycles can be detected, since connectivity information is restricted to a single execution sequence. The algorithm may analyze more states than the minimum required for an exhaustive search, since it cannot tell whether or not a newly generated state in the current execution sequence was encountered before in a previously analyzed sequence. The number of effectively analyzed states is in the range

$$R \leq R^* \leq L^D$$

With R, D, and L as defined in Table I. With the typical values from Table I we have

$$D_{max} \leq \frac{M}{S} \approx 10^5$$

a condition that is easily satisfied. However, we also have

$$10^8 \leq R^* \leq 10^{29}$$

and therefore

Table III. Comparison of Search Methods

Algorithm	Memory	Typical Value	Time	Typical Value	Coverage	Typical Value
Type 1	$S \times R$	10^{10} bytes	$E \times R$	10^6 seconds	R	100%
Type 2	$S \times A$	10^7 bytes	$E \times A$	10^3 seconds	A/R	0.1%
Type 3	$S \times D$	10^5 bytes	$E \times L^D$	$>10^6$ seconds	$\rightarrow R$	$\rightarrow$100%
Type 4	S	10^2 bytes	$\rightarrow \infty$	?	$\rightarrow R$	?

$$10^6 \leq T \leq 10^{27} \text{ seconds}$$

which means a run time of 10 days or more on our reference machine ($10 * 24 * 60 * 60 = 864{,}000$ seconds).

Type 4: Memory-less Search. A memory-less search, also called guided simulation or random walk,[16,19] minimizes memory usage at the expense of run time.

```
type_4()     /* memory-less search */
{     q = initial_state;
      while (q is not an error_state)
      { select one successor state s of q
        set q = s;
      }
      report_error();
}
```

The selection of the successor state can be random, or biased towards the selection of states likely to lead to error.[19] Execution cycles cannot be automatically detected or analyzed, but, in theory, coverage of all reachable system states is guaranteed.

$$M \geq S \quad \text{and} \quad T \geq E \times R^+$$

where

$$R^+ >> R$$

The memory requirements are easily satisfied. Using Table I, we have

$$M \geq 10^2 \text{ bytes}$$

However, since this search maintains no history of states, the search cannot establish when an exhaustive search has been completed. The algorithm must be run infinitely long before 100% coverage could be concluded.

Comparison

Table III compares the characteristics of the four search methods in three categories: memory, time, and coverage (cf. Table I). If S and R are small, a Type 1 algorithm is obviously the best choice. It is by far the most popular search algorithm in use today. If, on the other hand, S approaches the size of M, the only feasible algorithm is a Type 4 search. Colin West[20] reports that even very large software systems can be analyzed successfully with this search method.

For all other applications, algorithms of Types 1 and 4 must be disqualified: the first because of the memory requirements and the second because of the time requirements. When complete coverage is crucial and the time requirements bearable, a Type 3 algorithm may be feasible. In many cases, however, the time requirements will be prohibitive and the best choice is a Type 2 algorithm. It now becomes crucial to find the proper search heuristics to balance memory usage against coverage, and to make the chance of finding errors substantially larger than the coverage suggests. We will discuss such search disciplines below.

Organizing a Partial Search

One of the simplest methods to organize a partial search is a random, or biased random, selection of successor states, much like the random walk of the Type 4

algorithm.[19,29] The bias can be a heuristic that favors executions that are likely to reveal design errors fast. Four such search disciplines are discussed below.

Biased Random Walk. West[19,20] describes a Type 4 algorithm in which successor states are selected that cause multiple messages to be simultaneously in transit: transitions that correspond to send operations are given priority over transitions that correspond to receive operations. This condition is likely to reveal subtle coordination errors in protocol designs. It is interesting to note that this heuristic is the opposite of the scatter search, discussed next, and most likely will reveal a different class of design errors. The biased random walk was applied successfully to the analysis of very large software systems.[20]

West notes that even in cases where the absolute coverage of the search is extremely small, the quality of the search can still be good, where "quality" can be expressed as the fraction of the total number of design errors that is found with this method.

Scatter Search. In a scatter search,[1] 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 method was used to implement a Type 2 analyzer called trace. The probability of finding deadlocks in a protocol design with this heuristic can indeed be better than the coverage of the partial search would suggest. For instance, approximately 80 percent of the errors can be found in a search with a coverage of 25 percent of the state space.[1]

Guided Search. In a guided search,[16] the bias is a cost function that is dynamically evaluated for each successor state. The cost function can be static, as in a scatter search, a biased random walk, or a probabilistic search, or it can change dynamically during the search. This was first suggested as a method for guiding a Type 4 search, but it applies equally well to Type 2 searches. As yet, however, too little is known about the types of cost functions, or "guiding expressions," that could prove to be useful.

Probabilistic Search

In a probabilistic Type 2 search,[15] 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 selection criteria.

There are some potential difficulties with this search method. First, a realistic selection of the probability tags is very hard, and in some cases impossible.[21] A second problem is that most design errors are not likely to be found in the probable execution sequences. On the contrary, it can be argued that design errors are most likely to hide in the part of a protocol that is least likely to be validated: the executions with a low probability of occurrence. The executions with high probability of occurrence are usually well-understood, and easily debugged with trial implementations. Finally, the probability tags are meant to resolve selections within processes, but it is much harder to apply them for the resolution of selections between processes. Note that in a concurrent system every asynchronous process (i.e., xFSM) is equally likely to make transitions. Once a process is selected to make the next transition the probability tags apply, but it is unclear how they could be used at each step in the reachability analysis to select the process that makes the transition. This concurrency between processes, however, is the main factor responsible for the state space explosion phenomenon.

An Assessment. The main problem with all partial search methods is the difficulty of matching the size of set A to the available space M/S. For any protocol, the total number of reachable states R is hard to predict. It is therefore very difficult to define A as a fraction of R and still remain within the available arena M/S. If the final size of set A is underestimated, the partial search will again be truncated in a way that defeats the search heuristics.

To solve this problem we can try to turn the problem around: given M bytes of memory, how can we organize the search to use precisely M bytes, no more

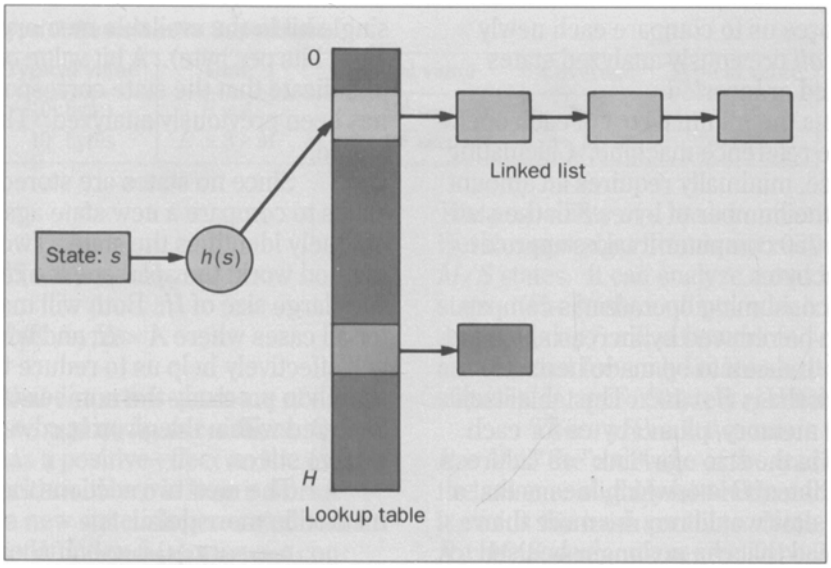

Figure 2. Hash table lookup.

and no less, and perform the largest search possible within that arena. Supertrace[2] is the first method to address this problem. It is interesting to note, though, that the storage method used in supertrace was primarily meant to reduce the run time requirements of the search algorithm. An effective use of available memory is a welcome side effect.

The assessment that led to this new search strategy is based on the following observations. The standard way to maintain set A in a Type 1 or Type 2 algorithm is by using a storage technique called "hashing" (e.g., References 22 and 23). The hashing technique allows us to quickly determine 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 to a lookup table of states. The table is organized as shown in Figure 2.

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 $0 \ldots H-1$. For the same state $s \in A$, $h(s)$ will always return the same 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 necessarily produce the same hash value for on the average A/H different states. All states that hash onto 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.

In a Type 1 or Type 2 algorithm, therefore, the following three operations must be performed for a new state:

1. Calculating a hash value
2. Comparing the new state against a subset of the previously analyzed states already in the hash list
3. Inserting the new state into a linked list.

Note carefully that these operations are not a peculiarity of a specific implementation but fundamental operations that are unavoidable in this type of search. The use of the hash table is an optimization technique that, in return for a small extra cost (1), avoids the worst

case in (2) that would force us to compare each newly generated state against *all* previously analyzed states before it can be discarded or inserted.

Table II tabulates the minimal cost of each operation on our medium size reference machine. Calculating a hash value, for instance, minimally requires an amount of time that is linear in the number of bytes S in the state description. For a VAX-750 computer it takes approximately 10^{-6} seconds per byte.

The most time-consuming operation is comparing states. Its effect can be reduced by increasing H, but clearly there is another trade-off to be made here.

A typical value for H is $H \approx 10^4$. The table itself takes up $H \times B$ bytes of memory, plus B bytes for each state that is inserted. B is the size of a "link" or "address pointer." On most machines, B is 4, which means that a table with, say, 256,000 slots would require more than one megabyte of overhead that can no longer be used to store states. Since this is about 10 percent of the total amount of memory available for the search (Table I), it is prudent to make H smaller.

For typical values of H (10^4), S (10^2), and M (10^7), and with A growing from 1 to M/S states, we find a minimal cost (the sum of all three operations) increasing with A from approximately 10^{-3} to 10^{-2} seconds per state (cf. Table I). This is again an optimistic estimate. Note that we disregarded all other operations that are required to perform the analysis of the states. The search efficiency E degrades as the search progresses: there is a growing time penalty for analyzing large systems.

Supertrace

If we were able to avoid the last two operations from Table II completely, we would be able to speed up the search algorithm by a factor of $10 \times A/H$, or by about 2 orders of magnitude. We can do this as follows. We use a very large value for H to reduce the number of hash conflicts. Here we will use $H = 8 \times M$, which for typical M makes $H = O(10^8)$, instead of $O(10^4)$. This hash value is now used to calculate the position of a single bit in the available memory arena M (there are eight bits per byte). A bit value of one will now be used to 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 also no states to compare a new state against: the bit position uniquely identifies the state. Two things make this method work: the sparseness of the state space and the very large size of H. Both will make hash conflicts rare for all cases where $A < H$, and when $A > H$ the hashing will effectively help us to reduce the coverage of the search to precisely the number of states that can analyzed within the given hardware constraints, that is, $O(10^8)$ states.

The next two sections explain the storage method in more detail.

Detailed Explanation. If H can be chosen to be much larger than A we have

$$A \ll H \quad \text{and} \quad A/H \to 0$$

which means that the number of different states that hash onto the same value becomes negligible. Hash value $h(s)$, then, almost uniquely identifies state s. This has two important implications.

First, it means that we do not need the linked lists in the lookup table, since the expectation is that at most one state will be stored under each index. If, in a small number of cases, two distinct states produce the same hash value, one of the two will have to be discarded and may, mistakenly, have to be analyzed more than once. If the number of cases in which this occurs is indeed negligible, we can omit the linked lists and with it avoid the overhead of the associated address pointers.

Second, it means that we do not have to store the state, since the lookup table index identifies it. It suffices to store a flag that indicates whether or not the state has been "entered" into the lookup table. To store the flag we need just one bit. The overhead of the lookup table

Table IV. Comparison of Type 2 Search with Supertrace

Algorithm	Memory	Typical value	Time	Typical value	Coverage	Typical value
Type 2	$S \times A$	10^7 bytes	$E \times A$	10^3 seconds	A/R	0.1%
Supertrace	M	10^7 bytes	$E' \times 8 \times M$	8×10^3 seconds	$8 \times M/R$	$\rightarrow 80\%$

reduces from

$$H \times B + (S + B) \times A \quad \text{bytes}$$

to

$$H/8 \quad \text{bytes}$$

Note, however, that since the states are not stored we can no longer recognize hash conflicts. Quite remarkably this defect has a positive effect on the overload behavior of the algorithm. Here is how it works.

Hash Conflicts. 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. In the case that this is wrong, i.e., when a hash conflict occurs, the search will ignore a state that should have been analyzed: the search is truncated. A search of this type is therefore necessarily a Type 2 search. With $A/H \rightarrow 0$, however, the number of hash conflicts approaches zero, and the search approaches a Type 1 search. Indeed, therefore, it is best to choose H as large as possible.

The maximum value for H we can choose, under the given system constraints from Table I, is $H = 8 \times M$. Let us see how the modified algorithm compares to an unmodified Type 2 search.

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

$$\frac{A}{H} = \frac{M}{S \times 8 \times M} = \frac{1}{8 \times S}$$

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 increase as the state space fills up,[2] will now work to scatter the states that are selected for analysis across the set of reachable states in an approximately random manner.

There are now two cases to consider. For $R < M/S$ the coverage of the traditional algorithm will be the same or slightly better than the new algorithm, since it avoids the hash conflicts. Note, however, that when $R < M/S$ we do not need a Type 2 algorithm: we can perform an exhaustive Type 1 search in core.

For large problems with $R > M/S$, the typical application area of the Type 2 search, the coverage of the new algorithm—that is, 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 $8 \times M/R$, compared to $M/(S \times R)$ for the traditional algorithm (see also Table IV).

Note also that 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. For $S = 500$, for instance, the coverage of the traditional Type 2 algorithm *drops* to a maximum of $A = 100{,}000$ analyzable states out of R reachable states. The coverage of the new algorithm, however, slowly *grows* toward a maximum of $H = 8 \times M$ analyzable states out of the same R reachable states.

The effect is illustrated, for a fixed size S, in Figure 3. The data in Figures 3 and 4 are taken from Reference 2. Increasing S is equivalent to moving the dotted and dashed lines to the left.

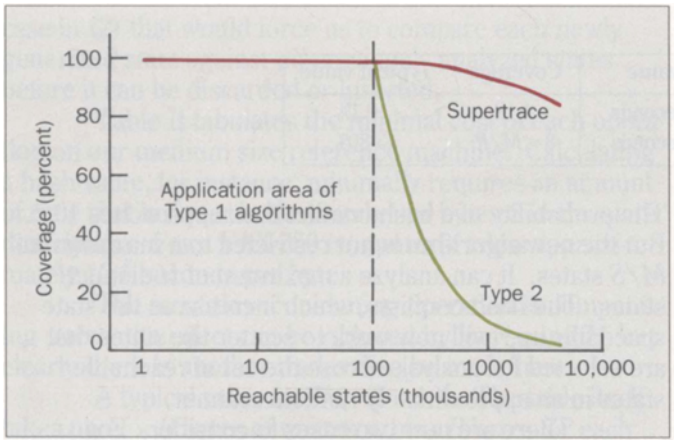

Figure 3. Comparison of two algorithms—coverage.

For protocols that can be analyzed exhaustively with a Type 1 algorithm, there is little difference between the two algorithms, with a slight edge for the traditional method. For larger protocols, though, the traditional method breaks down very rapidly, its coverage dropping by a factor of 10 for every tenfold increase in the number of reachable states. The coverage of the new algorithm is substantially better.

Next, consider the CPU time requirements. Note that the last two operations from Table II have disappeared. The only remaining cost is the calculation of the hash function, which means that the value of E from Table I increases from $O(10^{-2})$ to $O(10^{-4})$. The new algorithm is faster by up to 2 orders of magnitude. The difference is illustrated in Figure 4.

Next, we look at the "overload behavior." After all, in the original problem statement from Table I there are not A states to analyze but R. We have already noted that there is a time penalty for overload in the traditional Type 2 algorithm: the search slows down as the state space grows beyond the first H states. In the new algorithm the search efficiency will *not* degrade as the search progresses. There is a fixed cost associated with each step, which depends only linearly on the size of a state description S: the cost of calculating the hash value (Table II).

When $A \rightarrow R$, for the numbers from Table I, A is in the same order of magnitude as H, which means that a large fraction 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 $8 \times M/R$.

Table IV compares the new strategy with a traditional Type 2 algorithm.

This method was first described in Reference 24 and further explored in Reference 2. The method has been applied successfully to the analysis of a larger protocol systems that, because of their size, cannot be analyzed exhaustively with a traditional Type 1 or Type 2 method. Applications include a verification of the IEEE logical link control protocol[3] and portions of the AT&T 5ESS switch telephone switching code.[25]

Application to 5ESS Switch. The application of supertrace to 5ESS switch code has been the most substantial test performed so far.[25] The test was interesting for two main reasons: the size and complexity of the verification problem, and the formalism in which the problem is presented. The interaction code of the 5ESS switch (about 10 percent of all switching code) is written in the CCITT language SDL. The claim made in this paper is that the verification kernel, once developed and optimized, can be applied to a wide range of formalisms, including SDL. To prove the point, though not without difficulty, a preprocessor was written that translates SDL into an extended FSM model and applies the supertrace algorithm to the result.

At the time of writing, the resulting verification system for SDL has been in daily use by groups of 5ESS switch designers for more than a year. It is, for instance, used to verify the correctness of a substantial recoding of CCITT7 Telephone User Part features in the 5ESS switch. After an initial training of a few hours, a designer can learn how the supertrace software is applied to an SDL design for an exhaustive search. Using the SDL

preprocessor has given the advantage that the original SDL code needs little or no modification for the purposes of analysis: it is almost completely the same source that will run in the switch itself once the design process is completed.

Conclusions

The reachability analysis method that is described in this paper, offers a significant improvement over all traditional methods for protocol verification. The storage method used in supertrace is analyzer and machine independent. The state space search technique is also independent of the specific machine model used: it applies equally well to FSM models (e.g., Reference 12), Estelle (e.g., Reference 13), the S/R model,[10] and Petri Net models (e.g., Reference 11), to name just a few. To demonstrate this, a preprocessor for the CCITT specification language SDL has been developed.

Supertrace is the first search method in which the memory requirements of the search can be matched exactly to the system constraints. The method has been applied in both breadth-first and depth-first search algorithms.

The practical performance of automated protocol verification systems is an often overlooked issue, but is one of the main problems that prevents their more general use. With the search algorithm discussed in this paper, protocol descriptions generating in the order of 10^5 system states can be analyzed exhaustively in minutes of CPU time on a medium-size machine, or in seconds on a large machine. With this performance a large fraction of the protocols in which one would be interested in practice can be analyzed. Larger protocols can be analyzed with a partial search. In this case the new method, compared to a traditional algorithm, gives a faster response and a superior coverage of the state space.

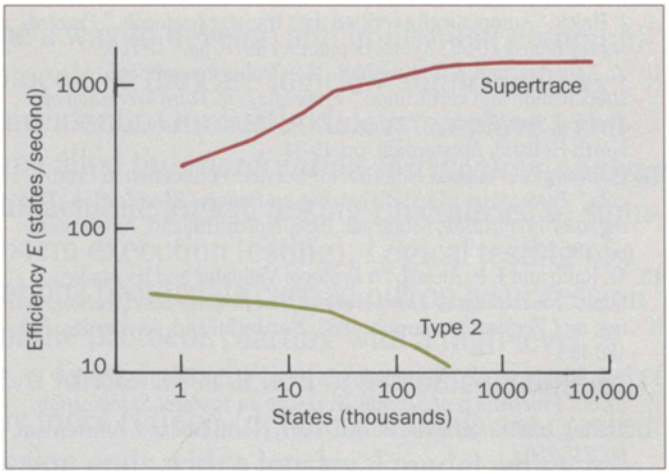

Figure 4. Comparison of two algorithms—search efficiency.

Acknowledgments

Rob Pike, Jim Reeds, and Ken Thompson helped me develop good hashing algorithms to optimize the performance of the supertrace software. I am also grateful to Joanna Patti and John Chaves who pioneered the application of supertrace to 5ESS switch code.

References

1. G. J. Holzmann, "Automated protocol verification in Argos, assertion proving and scatter searching," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 6, June 1987, pp. 683-696.
2. G. J. Holzmann, "An Improved Protocol Reachability Analysis Technique," *Software, Practice and Experience*, pp. 137-161, Feb. 1988.
3. IEEE Standard 802-2-1985, ISO DIS 8802/2, "IEEE Standards for Local Area Networks: Logical Link Control," IEEE Standards Board, New York, 1985.
4. G. J. Holzmann, "Tracing Protocols," *AT&T Technical Journal*," Vol. 64, No. 10, December 1985, pp. 2413-2434.
5. S. T. Vuong, D. D. Hui, and D. D. Cowan, "Valira—a Tool for Protocol Verification via Reachability Analysis," *Proceedings of Sixth Workshop on Protocol Specification, Testing, and Verification*, Quebec, 1986, North-Holland, Amsterdam, pp. 35-42.
6. Y. Kakuda, Y. Wakahara, and M. Norigoe, "An Acyclic Expansion Algorithm for Fast Protocol Verification," *IEEE Transactions on Software Engineering*, Vol. 14, No. 8, August 1988, pp. 1059-1070.
7. S. S. Lam and A. U. Shankar, "Protocol Verification via Projections," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, July 1984, pp. 325-342.
8. C. H. West, "General Technique for Communications Protocol Validation," *IBM Journal of Research and Development*, Vol. 22, No. 3, June 1978, pp. 393-404.

9. J. Hajek, "Automatically verified data transfer protocols," *Proceedings of Fourth ICCC*, Kyoto, September 1978, pp. 749-756.
10. A. Aggarwal and K. P. Kurshan, "A calculus for protocol specification and verification," *Proceedings of Third Workshop on Protocol Specification, Testing, and Verification*, Zurich, 1983, North-Holland, Amsterdam, pp. 19-34.
11. A. Bourguet, "A Petri Net Tool for Service Verification in Protocols," *Proceedings of Sixth Workshop on Protocol Specification, Testing, and Verification*, Montreal, 1986, North-Holland, Amsterdam, pp. 281-292.
12. O. Rafiq and J. P. Ansart, "A Protocol Validator and its Applications," *Proceedings of Third Workshop on Protocol Specification, Testing, and Verification*, Zurich, 1983, North-Holland, Amsterdam, pp. 189-198.
13. J. L. Richier et al., "Verification in Xesar of the Sliding Window Protocol," *Proceedings of Seventh Workshop on Protocol Specification, Testing, and Verification*, Zurich, 1987, North-Holland, Amsterdam, pp. 235-250.
14. K. K. Sabnani and A. M. Lapone, "PAV—Protocol Analyzer and Verifier," *Proceedings of Sixth Workshop on Protocol Specification, Testing, and Verification*, Quebec, 1986, North-Holland, Amsterdam, pp. 29-34.
15. N. F. Maxemchuck and K. K. Sabnani, "Probabilistic Verification of Communications Protocols," *Proceedings of Seventh Workshop on Protocol Specification, Testing, and Verification*, Zurich, 1987, North-Holland, Amsterdam, pp. 307-320.
16. J. M Pageot and C. Jard, "Experience in Guiding Simulation," *Proceedings of Eighth Workshop on Protocol Specification, Testing, and Verification*, Atlantic City, June 1988, North-Holland, Amsterdam.
17. G. J. Holzmann, "A theory for protocol verification," *IEEE Transactions on Computers*, Vol. C-31, No. 8, August 1982, pp. 730-738.
18. G. J. Holzmann, "PANDORA—an interactive system for the design of data communication protocols," *Computer Networks*, Vol. 8, No 2., 1984, pp. 71-81.
19. C. H. West, "Protocol Verification by Random State Exploration," *Proceedings of Sixth Workshop on Protocol Specification, Testing, and Verification*, Quebec, 1986, North-Holland, Amsterdam, pp. 233-242.
20. C. H. West, "Protocol Validation in Complex Systems," *Proceedings of SIGCOMM '89*, Austin, Texas, September 1989, pp. 303-312.
21. H. Rudin, "Protocol Engineering: A Critical Assessment," *Proceedings of Eighth Workshop on Protocol Specification, Testing, and Verification*, Atlantic City, June 1988, North-Holland, Amsterdam.
22. R. Morris, "Scatter Storage Techniques," *Communications of the ACM*, Vol. 11, No. 1, January 1968, pp. 38-44.
23. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974, pp. 111-113.
24. G. J. Holzmann, "On Limits and Possibilities of Automated Protocol Analysis," *Proceedings of Seventh Workshop on Protocol Specification, Testing, and Verification*, Zurich, 1987, North-Holland, Amsterdam, pp. 339-346.
25. G. J. Holzmann and J. Patti, "Validating SDL Specifications: an Experiment," *Proceedings of Ninth International Workshop on Protocol Specification, Testing and Verification*, Twente University, The Netherlands, June 1989, North-Holland, Amsterdam.

(Manuscript received July 6, 1989)