Tracing Protocols

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Automated protocol validation tools are by necessity often based on some form of symbolic execution. The complexity of the analysis problem however imposes restrictions on the scope of these tools. The paper studies the nature of these restrictions and explicitly addresses the problem of finding errors in data communication protocols of which the size precludes analysis by traditional means.
The protocol tracing method described here allows one to locate design errors in protocols relatively quickly by probing a partial state space. This *scatter searching* method was implemented in a portable program called *trace*. Specifications for the tracer are written in a higher level language and are compiled into a minimized finite state machine model which is then used to perform either partial or exhaustive symbolic executions. The user of the tracer can control the scope of each search. The tracer can be used as a fast debugging tool but also, depending on the complexity of the protocol being analyzed, as a slower and rather naive correctness prover. The specifications define the control flow of the protocol and may formalize correctness criteria in assertion primitives.

Bell System Technical Journal, Vol. 64, December 1985, pp. 2413−2434.

1. Introduction

Protocol validation by symbolic execution is inherently a time and space consuming task. For lack of better methods, though, many automated protocol validation tools do use symbolic execution algorithms [1,2,7,16,17], and even methods based on validation algebras such as CCS [14] or PVA [9,10] still implicitly formalize symbolic executions[1]. Unfortunately, the assumption that a computer will always be able to take over when the complexity of a complete analysis surpasses our ability to perform algebraic expansions by hand is decidedly wrong [3,5].

A protocol of a realistic size can generate a state space of in the order of 10^9 system states and up. As little as adding one single message type, one protocol variable, or one slot to the message queues can further expand the number of reachable system states by orders of magnitude. For a protocol of this size a symbolic execution algorithm can at best analyze in the order of 10 to 100 system states per second of CPU time, if the state space is built in core [12]. To analyze 10^9 states exhaustively would then take at least 115 days of computation. Furthermore, assuming that each state can be encoded in no more than 10 to 100 bytes, storing a state space of this size would still require a machine with several gigabytes of main memory.

So, if this appears to be infeasible, what is the best that can be done? In the design phase one would like to have tools that can trace the most glaring bugs in a protocol in no more than a few seconds of real time. The completeness of an analysis is not really at issue here; *speed* is. To find more subtle design errors of a completed protocol one may be willing to spent more time, but not much more than perhaps 10 hours or in the order of 10^5 seconds of CPU time. For symbolic execution algorithms, this requirement sets an upper

1) Cf. the expansion theorem in CCS and the shuffle operator in PVA.

limit to the number of states that can be searched at roughly 10^7 states. At 10 to 100 bytes per state, however, we cannot expect to do anything useful with a state space of more than in the order of 10^6 states. The tracer should therefore preferably be able to perform complete analyses in small state spaces holding just a fraction of the total number of states. In the remainder of this paper we will concentrate on these two issues: the effectiveness of partial searches and the possibility of performing complete searches in partial state spaces.

Overview

In the following we will assume that the protocol submitted to a tracer is likely to contain errors and that a designer is interested to see any nonempty subset of these. A protocol tracer may, for instance, scan the state space in an effort to quickly discover typical violations of user specified correctness requirements. It is important to note that the objective of such a partial analysis, or *scatter search* as we shall call it, is to establish the presence rather than the absence of errors.

What we are aiming for is a protocol tracing method that allows us to spent a small fraction of the time required by an exhaustive analysis to find a substantial portion of all design errors. The emphasis is on speed, not on completeness. If a protocol contains an error an exhaustive search would meticulously report every possible circumstance under which the error could make the protocol fail. For our purposes, tracing a single variant of the error in a partial search will suffice.

Section 2 explains how a general symbolic execution algorithm based on depth first search can be organized. It discusses a variant of symbolic execution called scatter searching and compares its performance with exhaustive searching. Section 3 discusses in more detail heuristics that can be used to perform a partial search, and section 4 shows how depth first searches can be organized in incomplete state spaces. Section 5 shows how protocol specific correctness criteria can be verified with a standard symbolic execution algorithm. Section 6 gives a small example of the use of correctness assertions in tracing bugs in a protocol. A larger example is presented in the appendix. Section 7 summarizes the main results.

2. Scatter Searching

Below we will discuss some experiments with a program called *trace* that performs a simple depth−first search in a partial state space generated by a set of interacting finite state machines, where the state space is maintained as a tree of system states. To determine the effectiveness of partial searches the performance of exhaustive searches and scatter searches was compared, using a search depth restriction as a parameter. But, first let us consider the working of the tracer in a little more detail.

With the exhaustive tracing method a state space tree is searched starting from the initial system state, exploring every possible execution path until an endstate, a previous state or an error state is reached, or until the search depth limit is encountered. A return to a previously analyzed state terminates the search under two conditions:

- if the previously analyzed state is in the execution path that leads from the root of the state space tree to the current state, or
- if the previously analyzed state was encountered elsewhere in the state space tree either at the same depth or closer to the root of the tree than the current state.

In the first case the tracer has discovered an execution loop in the protocol. The loop could be checked further on liveness, but to save time the program *trace* simply checks that its repetition does not violate the user specified correctness criteria and continues. In the second case the subtree that would be explored by continuing the search down to the search depth restriction would be contained in the subtree of the previously analyzed state, and cannot lead to new results. The tracer can therefore ignore the subtree and continue exploring new leaves in the tree.

Though the state matching method is more general than the one described in for instance [1], the design of the experimental tracer so far is fairly standard. The exhaustive trace method, however, can be considered to be a special case of the *scatter search*. In a scatter search not every possible execution sequence is necessarily explored. The tracer makes an estimate of the likelihood that exploring a new sequence can lead to the discovery of a new error, and will search only those sequences that optimize its chances of finding the largest set of unique errors in the smallest amount of time. The tracer's estimate will be based on a

heuristic that should be general enough to work on any type of protocol. One straightforward way to do this, for instance, is to restrict the amount of nondeterminism that will be taken into account by the tracer. These and other techniques will be discussed in more detail in section 3.

Test Results

To test the performance of a partial search we want to compare its coverage or 'scope' with that of an exhaustive search. The test protocol chosen for these comparisons was large enough to show the necessity of partial searches and also to give some room for experimenting with different flavors of partial searches. The size of the testcase however precluded, by the nature of the problem, the compilation of a definitive list of 'all' errors for reference. As a measure of the scope of the scatter search we will therefore take the number of errors traced and compare it with the number of errors traced by an exhaustive search method.

Figure 1 shows results of tracing an experimental data switch control protocol, generating a state space in the order of 10^9 system states [12]. The protocol was analyzed several times, both for the exhaustive search and the scatter search, with a search depth restriction that was incremented in steps of 10 levels for each new analysis run.

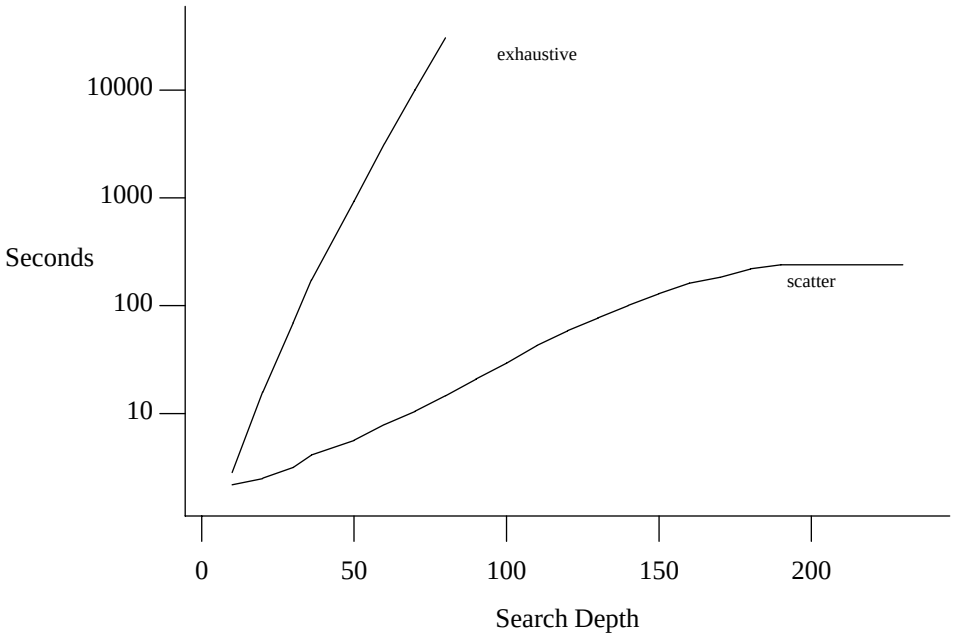

Figure 1 – Runtime

An exhaustive search for this protocol became unfeasible beyond a depth of 80 levels, i.e. numbers of states down from the root of the state space tree. The tree scanned by the scatter search method had a maximum depth of 189 steps. Setting the search depth restriction beyond 189 therefore no longer affects the scope of the analysis. To illustrate this, the curve for the scatter search was continued in Figure 1 up to a depth of 230. The longest scatter search required less than 4 minutes of CPU time to complete. The runtime of the exhaustive search tends to be exponential in the search depth. Using Figure 1 it can be estimated that searching the state space tree down to the same depth (189 steps) with the exhaustive search would take some 3000 years of CPU time.

Fortunately, the test protocol analyzed contained a generous number of design errors. No attempt was made to classify them. In Figure 2a the number of deadlocks reported by the tracer is shown as a function of the search depth, and in Figure 2b the number of deadlocks versus the time it took to find them is plotted on logarithmic scales.

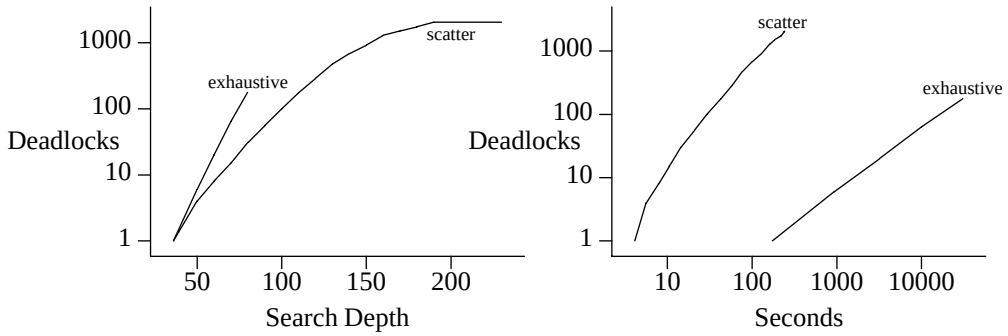

Figure 2 – Deadlocks

No deadlocks are found at search depths 10, 20, and 30. The first error is reported with the scatter search for a search depth of 40 steps, requiring 4 seconds of CPU time. For the same search depth restriction the exhaustive search reports the first 3 errors in 6 minutes. By repeating the analyses for intermediate levels between 30 and 40 we found that the first error is reported both in the scatter and the exhaustive search mode at level 35, requiring 4 seconds for the former and 3 minutes for the latter search. The two intermediate tests were included in the results shown in Figure 2.

Very probably, no protocol designer would be interested in tracing this protocol beyond the first 100 error sequences generated. For the given testcase this would mean that with an exhaustive search the first 70 steps in state space can be searched requiring roughly 3 hours of CPU time. Alternatively, the first 100 steps can be traced with a scatter search in only 30 seconds of CPU time.

Note that the time required to find the first error, the minimum search depth required to trace it and the relation between search depth and the number of errors reported are favorable for the scatter search method.

State Space

The protocol used for these tests requires roughly 40 bytes in the state space per system state. A total of 332,527 system states is generated in the longest exhaustive search analysis performed. As a result, for every new state generated, in the exhaustive search a data base of up to 15 Megabyte must be probed for a state match. Even with the best hashing methods this is bound to slow down the analysis noticeably. In the scatter search the largest number of states seen is 172,402 at a depth of 189 in the tree, corresponding to a data base of 8 Megabyte. The scatter search therefore should slow down less rapidly. This effect is illustrated in Figure 3. The time efficiency is expressed in the average number of states analyzed per second for each analysis run.

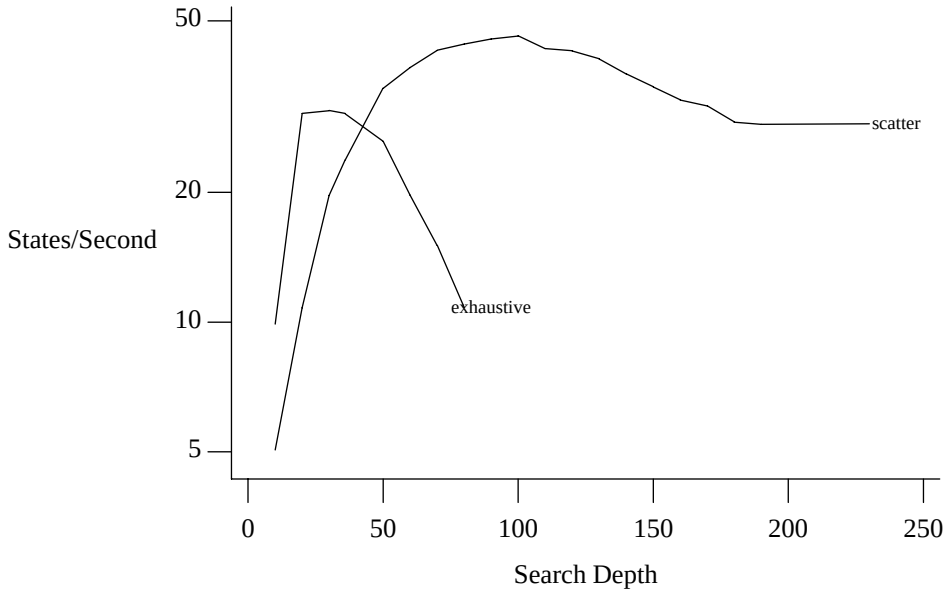

Figure 3 – Time Efficiency

The steep left hand side of the curves can be attributed to the overhead involved in the setup of a state space, which is felt more if the number of states explored is small. With the current tracer, the optimum speed for both search methods is reached when the state space contains roughly 1000 states.

3. Search Heuristics

It is relatively straightforward to give preference to the shortest complete execution sequences and to defer analysis for longer sequences. We have already used this method in the preparation of the figures above by bounding the depth of the tree explored during a search. In this section we consider some other partial search heuristics.

3.1. Fewer Interleavings

A method for reducing the run time of an analysis effectively is to restrict the amount of nondeterminism in the protocol model. In an exhaustive search each node in the state space tree is root to one subtree for each executable option in each finite state machine in the protocol. Not all interleavings of these actions is necessarily relevant. Consider two executable actions: one action a local to machine $M1$, e.g. an assignment to a local variable, and the other an external action b in machine $M2$, e.g. a send or a receive. There are two possible orders in which these two actions could be executed, corresponding to the two sequences

$$a;\ b \text{ and } b;\ a.$$

Each of the two sequences leads into a new state that forms the root of an entire subtree in the state space. The question is of whether or not the two subtrees are equivalent with respect to the errors to be traced. Note that the execution of internal action a will not change the environment for the remote machine $M2$, so neither the executability nor the result of b can be any different when a is executed first or last. Similarly, the execution of a is independent of the environment affected by $M2$ and also its executability and result is independent of whether b preceded it or not. In this case then, it suffices to search one of the two possible interleavings and to ignore the other. Unfortunately, there are not many cases where a complete subtree can be ignored without restricting the scope of an analysis. In some cases, though, we can predict in what way the scope will be affected. It would be unwise to restrict the nondeterminism that is local to a finite state machine, as shown in the following *Argos* [11] fragment:

```
if
:: A?one –> P()
:: B?two –> Q()
fi
```

Argos is a CSP−like [8] guarded command language [6] defined on buffered message channels. A detailed discussion of the language itself can be found in [11]. In the above example *A* and *B* are channel names (bounded buffers declared elsewhere), *one* and *two* are message names, and *P* and *Q* are procedure names. If message *one* is the first message in *A* and message *two* is the first message in *B* both input statements are executable and the process executing the above fragment can make a nondeterministic choice between the two alternatives, and then proceed with the execution of either $P()$ or $Q()$. The protocol tracer cannot foresee which of the two alternatives may produce an error without executing them. Note that ignoring one of the two alternatives in an analysis implies ignoring a potentially important code fragment, i.e. either $P()$ or $Q()$, without having reason to assume that this code would be error free. In this case then both alternatives will have to be explored. The situation is different for the nondeterminism that results from concurrency, as illustrated by the following *Argos* fragment:

```
proc P1 { A?one –> P() }
proc P2 { B!two –> Q() }
```

It defines two processes $P1$ and $P2$. Assuming that both initial actions are executable, it must be decided in what order they will be executed by the tracer. This time it may, but it will not always make a difference in what order these two i/o statements are executed. In an exhaustive search both orders are always analyzed. Ignoring one of the two possible orders, however, can half the amount of work to be done for this node in return for the chance that it will cause the tracer to miss error sequences. No code fragments are ignored here, only a potentially erroneous timing of executions. Fortunately, not all orderings have the same probability of leading into error states. For instance, if we are primarily interested in finding deadlocks states, that is states in which all message channels are empty and not all processes have reached their endstates, we may choose to explore the sequence starting with a receive action and ignore the other. In practice this heuristic performs remarkably well, as illustrated by the results discussed earlier.

3.2. Tracing Priorities

If at some node in the state space tree there are N concurrent processes, all executable, the tracer can decide to ignore any $M \leq N$ of the processes to reduce the search. In the tests reported in Figures 1 to 3 we set $M = 1$ for the scatter search and $M = N$ for the exhaustive search. In the case where $M = 1$ the search heuristic is implemented as a priority scheme that determines which process should be executed next. Highest priority is given to internal actions. At the next level we place receive actions, since these tend to bring the system closer to a deadlock state with empty channels. A lower priority is given to send actions, and a lower priority still to channel timeouts. Timeouts are given lowest priority in the partial searches since they tend to create many spurious error reports. In partial search mode the correct working of the timeout mechanism is assumed, that is, a timeout is only considered to be enabled when there is no other option to continue the protocol. Though this definitely reduces the scope of an analysis, it does allow us to trace for another class of errors first and defer the costly tracing of timing errors.

3.3. Queue Sizes

The capacity of a communication channel for holding messages can also have an important effect on the size of a state space. In the specification language *Argos* the channels are modeled by finite queues. A channel then can be in only a finite number of states

$$\sum_{i=0}^{N} |S|^i$$

where N is the number of slots in the channel (i.e. the queue size), and S is the size of the channel *sort*, i.e. the set of all messages that can be recognized by the channel. Reducing the number of slots N by one can reduce the size of the state space, and speed up the analysis, by a factor of up to

$$\sum_{i=0}^{N} |S|^i \; - \; \sum_{i=0}^{N-1} |S|^i \; = \; |S|^N$$

In the scatter searches of Figures 1 to 3 the queue sizes were restricted to two slots. To study the effect of a variation of the queue size the tests were repeated for a small range of sizes.

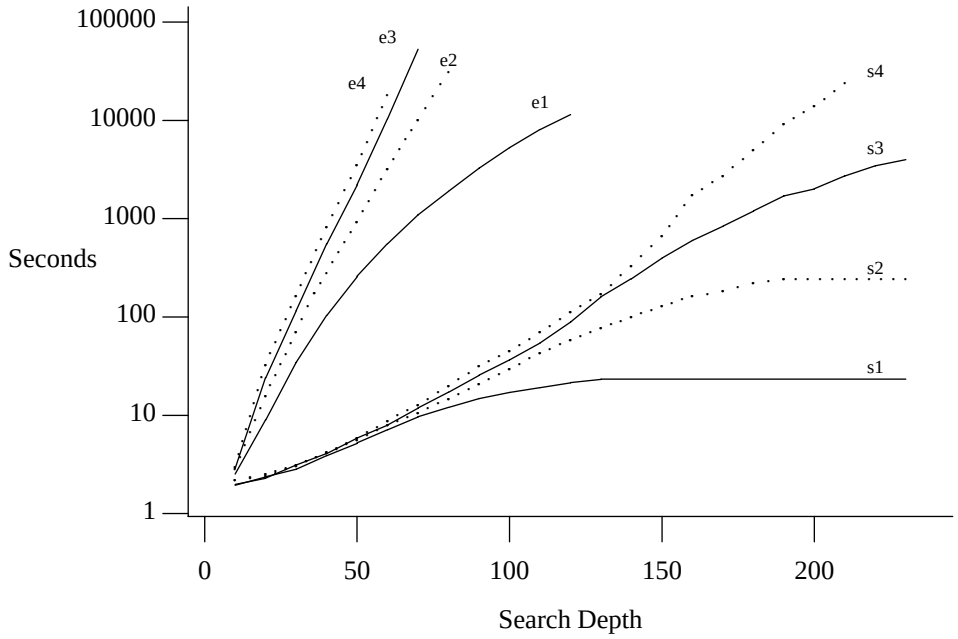

Figure 4 – Effect of Queue Sizes on Runtime
s – scatter search; e – exhaustive search; 1,2,3,4 – queue sizes

Figure 4 shows the effect of a variation in the number of slots between one and four for both the exhaustive and the scatter search.

4. Restricting The State Space

In the introduction we mentioned that the tracer should be able to perform searches in even incomplete state spaces since the size of a complete state space generally precludes its storage or even its usage during the search. In this section we will see how this can be accomplished.

First it should be noted that in a depth first search, at each execution step only those states that lead from initial state to the current state are indispensable in the state space. The presence of these states is necessary for the detection of system execution loops. Not every system state, though, can be found at the start of such an execution loop, and therefore it is not necessary to remember each state along a single execution path. The only states that must be remembered are those in which at least one of the interacting finite state machines is at the start of a local execution loop. Figure 5a shows a small but consistent reduction in the numbers of states if we restrict the state space to such ‘loop states.’

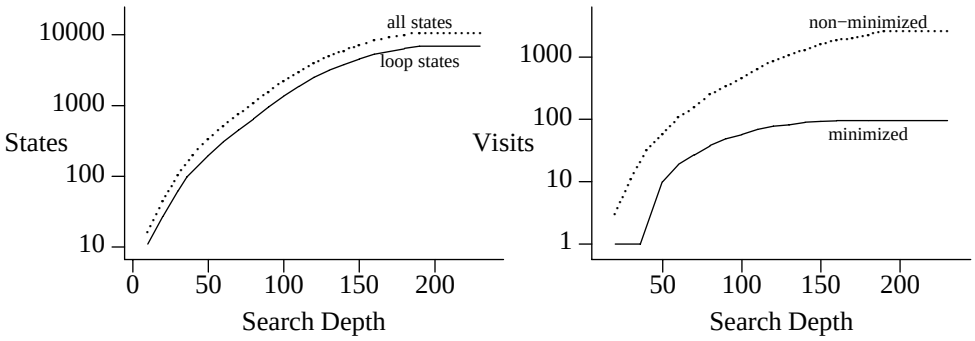

Figure 5 – Reducing Numbers of States Stored (scatter searches)

4.1. Minimization

Since the analysis is performed on finite state machines we can try to minimize the machines in an effort to reduce time or space complexity without affecting the scope of an analysis. The machines can not be reduced under the standard notion of language equivalence, since that will change the behavior or the protocol. A stronger notion of state equivalence [11], similar to that defined in CCS [14] can be used.

Figure 5b compares the analyses of minimized state machines and non−minimized state machines. The protocol tested defines 4 processes, 34 message types, 6 message channels, and 3 local variables. The state machines generated for the processes contain 69, 47, 7, and 5 states respectively. The strongly equivalent minimized machines contain 35, 31, 7, and 5 states. As it turns out, the number of states generated in the state space is roughly the same in both cases. The connectivity of the state space tree, however, is different, causing the same states to be visited more frequently for the non−minimized machines, resulting in a small increase in runtimes.

The effort to minimize the amount of work to be done in the search algorithm is concentrated on minimizing the theoretical maximum number of states in the product space of the individual finite state machines. We can do this by reducing the number of states per state machine (eg by masking a variable or a message queue) or, less straightforwardly, by reducing the number of state machines as such. The last thing we would like to do is of course to extend the number of state machines that we begin an analysis with.

Somewhat paradoxically, this approach seems to conflict with the more conventional structured approach to program design that tells us to identify functions and to separate these in a relatively large number of logical entities. For protocol design this approach was most recently suggested in [13], describing a method where each logical entity is formalized in a small finite state machine that interacts with the others. Dividing a single automaton of 16 states into two state machines of 8 states each, however, quadruples the number of states in the product space. Similarly, dividing it into 4 even simpler state machines of 4 states each expands the product state space to 16 times its original size. In general, increasing the number of state machines leads to an exponential growth of the product state space and is counterproductive in analyses.

4.2. Cache Size

We noted above that, unlike the more commonly used *breadth* first search (e.g. [3,13,17]), the state space in a *depth* first search need only contain the states in a single execution path from the root to the current state. Storing other states can avoid double work, but does not affect the scope of the analysis. This property of the depth first search method gives us greater flexibility in controlling the state space size during analysis runs. If more states can be stored, though, the search will be more time efficient. Figure 6 shows the effect of the size of the state space on the time and space requirements of a search, for a state space cache of 150,000 states that is reduced in steps of 1,000 to a cache of 50,000 states. ‘Double work’ is measured here as the total number of states created or recreated while searching. Note that the number of states stored in the cache could roughly be halfed without noticeable effect on the runtime or the total number of states created.

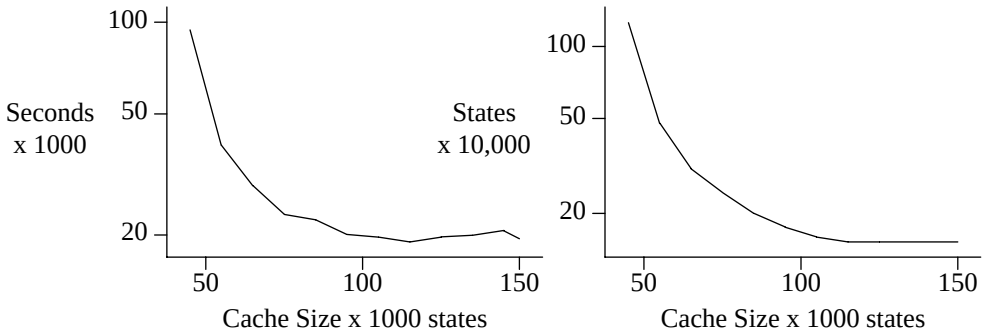

Figure 6 – Size of State Space Cache

With a partial state space cache it has to be decided which state will be deleted from a full cache when a new state must be created. A simple blind round robin selection of states was found to outperform a series of other, more subtle, schemes [11,12]. It is the strategy used in the test of Figure 6.

5. Assertion Checking by Symbolic Execution

By default a protocol tracer can check a protocol for the observance of general correctness requirements such as absence of deadlock and completeness. The validation language *Argos* allows for the specification of assertions to check on the observance of other correctness requirements. Assertions are defined as a restricted class of processes. They specify global system behavior in terms of external actions. For example, the specification

```
assert
{       do
        :: large!mesg; small!mesg
        od
}
```

is a requirement on the order in which messages of the type *mesg* are sent to the two channels *large* and *small*. The assertion is that in each execution sequence a message on channel *large* must precede a message on channel *small*, and that these two actions will be executed repeatedly (the are enclosed in a *do* loop) in precisely this order.

The main restriction to assertion specifications is that they can only refer to external actions, i.e. sends and receives, and not to variables. Assigments and boolean conditions are only allowed in process definitions, not in assertions. The control flow constructs are the same as those for process specifications: concatenations, selections, iterations, jumps, procedure calls, and macros. In other words: the assertions specify *global* constraints on the execution of the system as a whole in terms of message exchanges only. The scope of the assertion, that is the set of external actions that is traced to verify or to violate an assertion, is implicitly defined by the set of external actions it refers to. If an external action occurs at least once in an assertion body all its occurrences in an execution of the protocol are required to comply with it. Compliance with the assertion then means that the execution of these actions should match the context specified in the assertion.

Since we define assertions as restricted processes, the assertion primitives can be compiled into a restricted class of state machines and minimized with the same algorithm that is used for the compilation of the protocol processes. The protocol tracer uses the assertion state machines to *monitor* the external actions on which they are defined. Alternatively, though our tracer does not exploit this possibility, it may be possible to develop a heuristic that allows the tracer to select those executions in a partial search that have the best chance of violating the correctness requirements expressed in the assertions.

If an action is within the scope of an assertion, the state of the corresponding state machine will be updated as a side effect of the execution of that action, as if the assertion machine itself generated it. Since the assertion primitives cannot access variables or channels the ‘state’ of an assertion machine is uniquely defined by its control−flow state. The ‘execution’ of an assertion machine then costs very little in the

tracing algorithm. When the protocol system reaches an endstate, compliance with the assertion can be established by verifying that the assertion machine can reach an endstate too. If this is not true the assertion is violated and the current execution sequence can be listed as a counter example. Similarly, if the assertion machine can not be executed for an action that is within its scope, the assertion has been violated and a counter example can be produced. With little overhead or added complexity, the finite state machine model can thus be exploited to combine the depth−first search with assertion checking capabilities.

6. An Example

A small example can illustrate how the experimental protocol tracer described in this paper is typically used. More elaborate examples can be found in the appendix and in [11]. A protocol is defined in the language *Argos*. The example below shows three processes *a*, *b*, and *c*, three message queues of one slot each named *A*, *B*, and *C*, and one assertion labeled *assert* (a keyword in the language).

```
assert { C!a; C!b }

proc a
{       queue A[1];

        C!a; A?c
}

proc b
{       queue B[1];

        C!b; B?c
}

proc c
{       queue C[1];

        if
        :: C?a –> A!c; C?b –> B!c
        :: C?b –> A!c; C?a –> B!c
        fi
}
```

The assertion states that the two messages *a* and *b* will be appended precisely once to queue *C* when the three processes are executed, and that they can be sent in that order only. Process *a* starts by sending message *a* to queue *C* and then waits for a response *a* to arrive in queue *A*. Similarly, process *b* first sends *b* to queue *C* and then waits for a message *c*. The third process waits for a message to arrive in queue *C*, which is assumed to be either an *a* or a *b*, anything else would be an error. Process *c* then responds by sending a *c* message to queue *C* and waits for a second message to arrive: a *b* is the first received message was an *a*, or an *a* if the first message was a *b*. It will complete by sending a *c* into queue *B*.

The protocol is compiled into four finite state machines of three states each for *a* and *b*, three states for the *assert* primitive, and seven states for process *c*. The protocol tracer then takes over and completes an exhaustive search in 1.35 seconds, reporting the obvious assertion violation for the execution sequence that starts with C!b. The violation is reported by the tracer in the following format.

queue:	A	B	C
event:			
1			b
2	c		

```
3                a
4          c
```

Each column corresponds to a queue and each line to a time step. The first event is the sending of message *b* to queue *C*, which already violates the assertion. Then message *c* is sent to *A* by process *c*, message *a* is sent to *C* by process *a*, and finally a *c* message is sent to queue *B* by process *c*.

Changing the assertion to a more reasonable statement such as "assert { A!c; B!c }" will avoid the problem. The exhaustive search for this assertion completes in 1.32 seconds. Omitting the assertion completely will trigger a default search for deadlocks and incompleteness (eg. unspecified receptions), which completes in 1.18 seconds. Note that it is relatively straightforward to formalize liveness criteria in assert statements. In this case, as for many protocols generating up to 10^5 system states, exhaustive analyses are quite feasible. The real problems of partial searching only occur for the larger protocols comparable in size to the experimental protocol used for the tests reported earlier in this paper.

7. Conclusions

The main assumption we make in this paper is that in a design phase a protocol is typically known to contain errors and there is a need for a protocol tracing tool that can quickly find a representative subset of these errors. The user of such a protocol tracer is not so much interested in completeness but is very much interested in speed. With these assumptions important reductions in the time and space requirements of a tracer become feasible.

The protocol tracer described here consumes only a small fraction of the time and space required by an exhaustive analysis algorithm to find a relatively large fraction of the errors present. The run time of a state space search is reduced by several orders of magnitude by restricting the number of interleavings, by using search depth and queue size restrictions and compile time minimizations (Figures 1, 2, 4, and 5b). A more general method of reducing runtime would be the definition of equivalence classes, or *state space foldings*, as described in e.g. [10]. The experimental protocol debugger *trace* does allow for the definition of such foldings, but too little experience with this technique has yet been gained to report any results.

The number of states stored in a state space can be reduced by carefully selecting the states that may be revisited (Figure 5a). More importantly, though, the depth first search technique used allows one to perform searches with an incomplete state space cache. For the protocol tested the cache could be reduced to less than 50% of the state space size (Figure 6).

Acknowledgement

The experiments with assert primitives and the statespace cache were inspired by discussions with Bob Kurshan and Sudhir Aggerwal. I am also grateful to Doug McIlroy, Lee McMahon, Rob Pike, Ed Sitar and Ken Thompson for discussions, support and inspiration during the development of the protocol tracer.

8. References

1. T.P. Blumer, and R.L. Tenney, "A formal specification technique and implementation method for protocols," Computer Networks, **6**, (1982), No. 3, pp. 201−219.

2. D. Brand and W.H. Joyner Jr., "Verification of protocols using symbolic execution," Computer Networks, **2**, (1978), pp. 351−360.

3. D. Brand and P. Zafiropulo, "Synthesis of protocols for an unlimited number of processes," Proc. Computer Network Protocols Conf., IEEE 1980, pp. 29−40.

4. C.S. Crall and D.P. Sidhu, "Executable logic specifications for protocol service interfaces," unpublished report, Iowa State University, May 8, 1985, 52 pgs.

5. P.R.F. Cunha and T.S.E. Maibaum, "A synchronization calculus for message oriented programming," Proc. Int. Conf. on Distributed Systems, IEEE 1981, pp. 433−445.

6. E.W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," Comm. ACM, **18**, No. 8, (Aug. 1975), pp. 453−457.

7. J. Hajek, "Automatically verified data transfer protocols," Proc. 4th ICCC, Kyoto, Sept. 1978, pp. 749−756.

8. C.A.R. Hoare, "Communicating sequential processes," Comm. ACM, **21**, No. 8, (August 1978), pp. 666−677.
9. G.J. Holzmann, "A Theory for protocol validation," IEEE Trans. on Computers, **C−31**, No.8, (August 1982), pp. 730−738.
10. G.J. Holzmann, "The Pandora system – an interactive system for the design of data communication protocols," Computer Networks, **8**, No. 2, (1984), pp 71−81.
11. G.J. Holzmann, "Automated protocol validation in 'Argos,' assertion proving and scatter searching," 1984, submitted for publication, available from the author, 23 pgs.
12. G.J. Holzmann, "Trace – performance measurements," AT&T Bell Laboratories, unpublished internal report, Jan. 1, 1985, 29 pgs.
13. R.P. Kurshan, "Proposed specification of BX.25 link layer protocol," AT&T Technical Journal, **64**, No. 2, (Feb. 1985), pp. 559−596.
14. R. Milner, "A Calculus for Communicating Systems," Lecture Notes in Computer Science, **92**, (1980).
15. National Bureau of Standards, Specification of a transport protocol for computer communications, **4**, "Service Specifications," June 1984.
16. C. West, "Applications and limitations of automated protocol validation," Proc. 2nd IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification, USC/ISI, Idyllwild, CA. May 1982, pp 361−373.
17. P. Zafiropulo, C.H. West, H. Rudin, D.D. Cowan and D. Brand, "Toward analyzing and synthesizing protocols," IEEE Trans. Commun., **COM−28**, No. 4, (1980), pp. 651−661.

APPENDIX

The following specification describes a transport protocol defined by the National Bureau of Standards [15]. The specification is based on the model given in [4]. Four processes are defined: a local user process *AU* connected to a server process *A*, and a remote user *BU* connected to server process *B*. The control flow constructs and the I/O statements in *Argos* are based on CSP, using buffered message channels instead of rendez−vous. Process *A*, for instance, receives messages via two channels: one is named *ua* and is used by the user process to request services, the other is named *ca* and is used here to receive control messages from the remote server. Messages from server to user are sent through channel *UA*. The communication between the two servers is modeled with control messages $m1$ to $m7$, as defined in [4]. The analysis discussed here uses no *assert* primitives and is thus a general one for completeness and absence of deadlocks. The arrow and the semicolon are syntactically equivalent statement separators. A double colon flags the start of an option in a repetitive construct (*do* ··· *od*) or in an alternative construct (if ··· *fi*). In this case, the state transition diagram defining the protocol is most conveniently, though not most elegantly, modeled by assigning a label to every state and including a goto−jump for every transition. Processes *A* and *B* are symmetrical. Null transitions from the original protocol were deleted from the model.

```
proc A
{       queue ca[8], ua[8];

closed:
        do
        :: ca?m1 → UA!conn_ind → goto rcvd
        :: ua?conn_req → cb!m1 → goto crsent
        :: ua?abort  → cb!m4
        :: ca?m4 → UA!d
        od;
crsent:
        if
        :: ca?m2 → UA!conn_conf → goto estab
        :: ca?m7 → UA!disconn → goto closed
        :: ua?abort  → cb!m4 → goto closed
        :: ca?m4 → UA!d  → goto closed
        fi;
rcvd:
        if
        :: ua?conn_resp → cb!m2 → goto estab
        :: ca?m7 → UA!disconn → goto closed
        :: ua?abort  → cb!m4 → goto closed
        :: ca?m4 → UA!d  → goto closed
        fi;
estab:
        do
        :: ua?close_req → cb!m3 → goto Aclose
        :: ca?m3 → UA!close_ind → goto Pclose
        :: ua?data_req → cb!m5
        :: ca?m5 → UA!data_ind
        :: ua?expid_req → cb!m6
        :: ca?m6 → UA!expid_ind
        :: ca?m7 → UA!disconn → goto closed
        :: ua?abort  → cb!m4 → goto closed
        :: ca?m4 → UA!d  → goto closed
        od;
```

```
Aclose:
        do
        :: ca?m3 → UA!close_ind → goto closed
        :: ca?m5 → UA!data_ind
        :: ca?m6 → UA!expid_ind
        :: ca?m7 → UA!disconn → goto closed
        :: ua?abort  → cb!m4 → goto closed
        :: ca?m4 → UA!d  → goto closed
        od;
Pclose:
        do
        :: ua?data_req → cb!m5
        :: ua?expid_req → cb!m6
        :: ua?abort  → cb!m4 → goto closed
        :: ca?m4 → UA!d  → goto closed
        :: ua?close_req → cb!m3 → goto closed
        :: ca?m7 → UA!disconn → goto closed
        od
}

proc B
{       queue cb[8], ub[8];

closed:
        do
        :: cb?m1 → UB!conn_ind → goto rcvd
        :: ub?conn_req → ca!m1 → goto crsent
        :: ub?abort  → ca!m4
        :: cb?m4 → UB!d
        od;
crsent:
        if
        :: cb?m2 → UB!conn_conf → goto estab
        :: cb?m7 → UB!disconn → goto closed
        :: ub?abort  → ca!m4 → goto closed
        :: cb?m4 → UB!d  → goto closed
        fi;
rcvd:
        if
        :: ub?conn_resp → ca!m2 → goto estab
        :: cb?m7 → UB!disconn → goto closed
        :: ub?abort  → ca!m4 → goto closed
        :: cb?m4 → UB!d  → goto closed
        fi;
estab:
        do
        :: ub?close_req → ca!m3 → goto Aclose
        :: cb?m3 → UB!close_ind → goto Pclose
        :: ub?data_req → ca!m5
        :: cb?m5 → UB!data_ind
        :: ub?expid_req → ca!m6
        :: cb?m6 → UB!expid_ind
        :: cb?m7 → UB!disconn → goto closed
        :: ub?abort  → ca!m4 → goto closed
```

```
        :: cb?m4 → UB!d  → goto closed
        od;
Aclose:
        do
        :: cb?m3 → UB!close_ind → goto closed
        :: cb?m5 → UB!data_ind
        :: cb?m6 → UB!expid_ind
        :: cb?m7 → UB!disconn → goto closed
        :: ub?abort  → ca!m4 → goto closed
        :: cb?m4 → UB!d  → goto closed
        od;
Pclose:
        do
        :: ub?data_req → ca!m5
        :: ub?expid_req → ca!m6
        :: ub?abort  → ca!m4 → goto closed
        :: cb?m4 → UB!d  → goto closed
        :: ub?close_req → ca!m3 → goto closed
        :: cb?m7 → UB!disconn → goto closed
        od
}

proc AU
{       queue UA[8];
        pvar m = 0;

        do
        :: UA?conn_conf → ua!close_req; UA?close_ind
        :: UA?close_ind → ua!close_req
        :: UA?conn_ind → ua!conn_resp
        :: (m == 0) → m = 1; ua!conn_req
        :: (m == 1) → m = 2; ua!abort
        :: UA?default
        od
}

proc BU
{       queue UB[8];

        do
        :: UB?conn_conf → ub!close_req; UB?close_ind
        :: UB?close_ind → ub!close_req
        :: UB?conn_ind → ub!conn_resp
        :: UB?default
        od
}
```

The queue sizes were arbitrarily set to 8 slots per channel. The protocol tested is defined by the behavior of the two server machines, as visible to the users. The user behavior is no part of the formal protocol. An arbitrary set of user processes was defined specifically for the test. The local user *AU* will open the connection by sending a *conn_req* message to *ua* and some arbitrary time later it will close it with an *abort* message. The remote user *BU* is considered to be passive, responding only to close messages and accepting, but ignoring all others. *Default* is a keyword for receptions that match any input from the queue specified.
The protocol as specified above is compiled, in 23 seconds of CPU time on a VAX−750, into four finite

state machines of 27, 27, 10 and 6 states respectively. The compiler flags a series of incompleteness errors, noting for instance that control message *m*7 can be received but is never sent. Ignoring those warnings, an exhaustive analysis with *trace* takes just under 3 seconds of CPU time and reports 4 error sequences that reduce to two types of errors. The first one is an unspecified reception of the message *conn_resp* in state *closed*, for instance, after the following message exchange:

```
queue:      ca     ua     UA    cb     ub     UB
event:
 1          conn_req,
 2                 m1,
 3                                  conn_ind,
 4          abort,
 5                 m4,
 6                                  d,
 7                        [conn_resp],
```

Each column corresponds to a queue and each line to a time step. The first event recorded is the sending of a message *conn_req* into queue *ua*, followed by an *m*1 into queue *cb*, etc. The last message sent is enclosed in square brackets to indicate that it was sent but could not be received. Comparing the event sequence with the program shows that server process *B* is in state *closed* at the time.
The second problem is an unspecified reception of *m*2 also in state *closed*:

```
queue:      ca     ua     UA    cb     ub     UB
event:
 1          conn_req,
 2                 m1,
 3                                  conn_ind,
 4          abort,
 5                       conn_resp,
 6                 m4,
 7   [m2],
 8                                  d,
```

It is now straightforward to study the behavior of the protocol for different user behaviors, which can reveal, for instance the possibility of the unspecified reception of a message *conn_resp* in state *Pclose*, or the more obvious deadlock after a simultaneous *conn_req* message from both users.