Computer Networks and Simulation III
S. Schoemaker (Editor)
© Elsevier Science Publishers B.V. (North-Holland), 1986

PROTOCOL TRACING

Gerard J. Holzmann
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Abstract

Tools for performing automated analyses of data communication protocols are frequently based on symbolic execution algorithms that search the behavior of finite state machine models. The symbolic execution algorithms, however, are inherently restricted in the size of the models that they can handle. They work well for models that generate up to 100,000 system states. Beyond that the time and space complexity problems prevent effective analyses. A simple model, that would lend itself to faster symbolic execution, is often too restrictive for adequate modeling of real life problems. Extending the model to include, for instance, variables and value passing expands the state space to be searched by orders of magnitude, far beyond the scope of an unmodified symbolic execution algorithm.We will discuss an experimental protocol tracer that specifically aims to battle the complexity of larger protocols.

The material presented here is loosely based on [Holz '85b].

1. INTRODUCTION

Protocol validation by symbolic execution is known to be a complex task [Bra '80, Cun '81]. Protocols of realistic size can generate state spaces of 10**9 system states and up. A symbolic execution algorithm, however, at best analyzes in the order of 10 to 100 system states per second of CPU time [Holz '85a]. To analyze 10**9 states exhaustively then takes at least 115 days of computation. Assuming that each state can be encoded in 10 bytes would mean that we would also need a 10 gigabyte machine to store the complete state space. Analysis purely by exhaustive symbolic execution therefore seems hardly feasible in these cases.

To be useful as an interactive design tool, a protocol analyzer should be able to report bugs in seconds rather than in days. In particular, a case can be made that in the design phase the completeness of an analysis is less important than its **speed**. A designer can probably do more with a tool that

produces a representative selection of errors in three seconds than with one that generates an exhaustive list of design errors in three months. For more complete analyses one may be willing to spend more time, but not much more than in the order of 10**5 seconds of CPU time. For symbolic execution algorithms, these requirements set an upper limit to the number of states that can be searched at roughly 10**7 states. Similarly, having say 10 Megabyte of storage space available sets an even lower limit to the number of states that can be kept in the state space during a search to 10**6 states.

In the following, we will assume that the protocol submitted to a tracer in the design phase is likely to contain errors and that a designer is interested in seeing any nonempty subset of these fast. With a protocol tracer we can, for instance, try to scan the state space in an effort to quickly find typical violations of correctness requirements. 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. This protocol tracing method allows us to spent a small fraction of the time required by an exhaustive analysis to find a large fraction of all 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 suffices.

A small number of automated protocol validation systems have already been completed, or are being developed [Blu '82, Holz '84, Holz '85c, Kur '85, Ram '85, Yem '83]. Little has been published about their scope or specific runtime and space requirements, which makes it difficult to compare them adequately. In particular, though most if not all automated tools are based on some form of symbolic execution, little is known about the techniques that were used in each case to reduce the notorious effect of a combinatorial explosion. The first important work on automated protocol validation was done by Zafiropulo and his group in Zurich [Bra '78, Wes '82, Zaf '78, Zaf '80]. Similar work, though less well documented was reported by Hajek [Haj '78]. The conference proceedings of the IFIP workshops on protocol verification [IFIP '80-'85] give a good overview of the progress made since then.

2. SCATTER SEARCHING

Trace is a Unix™ based program that performs a depth-first search in the state space generated by a set of interacting finite state machines. The state space is maintained as a tree of system states, where each system state is a unique combination of control-flow states for the state machines, local and global variables values and buffer states (i.e. the contents of queues).

Unix is a trademark of AT&T Bell Laboratories.

We will compare the performance of three different search methods with this tracer: exhaustive searching, scatter searching, and a variant of exhaustive searching that we shall call **partial searching** In the first few comparisons a search depth restriction in the state space tree is used as a parameter.

With the exhaustive search method the state space tree is explored starting from the root along every possible execution path until an endstate or an error state is reached, until the search depth limit is encounted or until, under certain conditions, a previously analyzed state is encountered. In the exhaustive search a return to a previously analyzed state can terminate the search only when the following two conditions are met:

- 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 a system execution loop (difficult to find with, for instance, the more popular breadth-first search method). 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 "partial search" is an attempt to restrict the runtime of the exhaustive analysis, without restricting the scope too much. In this case the search is always terminated when a previously analyzed state is encountered, whether it was previously found closer to the root of the state space tree or not.

The "scatter search" method tries to be a little more intelligent about which sequences are analyzed and which are skipped. The tracer `guesses' the likelyhood that sequences can lead into new error states. One straightforward way to do this, for instance, is to restrict the amount of nondeterminism that will be taken into account by the tracer. It is very hard to measure the scope of an analysis, especially if, by the nature of the problem, no definite list of all flaws in a protocol can be compiled for reference. As a measure of the scope of the scatter search we will take the number of errors traced and compare it with the number of errors traced by an exhaustive search method.

Rather than constructing one or more special purpose test cases for the measurements, the performance of the analyzer was tested on a single protocol of realistic size and of practical relevance. The test protocol is a model of an experimental data switch control protocol, independently developed, studied and subsequently abandoned by a programmer who shall remain anonymous. Selecting a larger practical test case has the important advantage

that the tests are realistic. For one thing, the tests had to be long enough to meaningful comparisons could be made between the different types of analyses. There are however also disadvantages. The protocol was large enough that its state space could not be exhaustively searched within given hardware (memory size) or human (lifetime) constraints. Memory available to store the state space was restricted to 7 Mbyte of RAM which for the given protocol holds roughly 175,000 states (virtual memory allowed the tracer to exceed that limit where necessary). The runtime of the validations was restricted to an arbitrary 10 hours of CPU. Since the size of the state space generated by the test protocol precluded the compilation of an exhaustive list of errors against which the quality of the analysis techniques could be measured, the results were only used to weigh their relative merit, not to set more absolute standards

Figure 1 shows how the runtime of a validation varies with the search depth for these three different search modes. The queue size for the test protocol was fixed at two slots per queue in all measurements that follow, except those that specifically measured the effect of the queue size on validations. Fortunately, the protocol model triggers a generous number of error reports. No attempt was made too classify them.

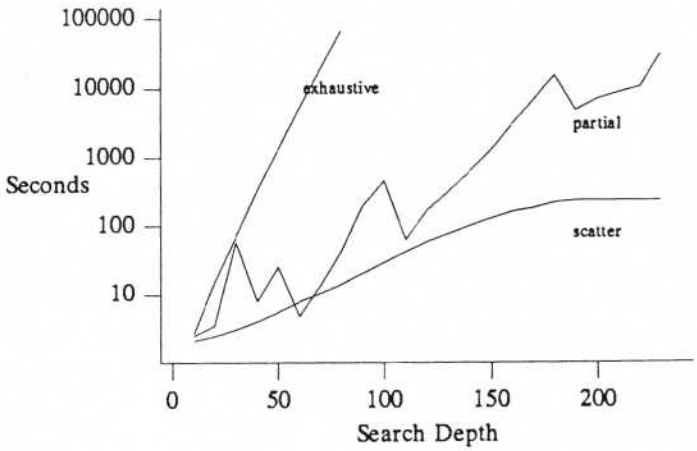

Figure 1 Runtime

An exhaustive search for this protocol becomes unfeasible beyond a depth of 80 execution steps in the tree. The tree scanned by the scatter search method has a maximum depth of 189 steps The longest scatter search requires less than 4 minutes of CPU time to complet. Searching the state space tree down to the same depth with an exhaustive search (190 steps) would take an estimated 3000 years of CPU time. The runtime of a partial search turns out to be rather unpredictable.

In Figure 2 the number of deadlocks reported versus the time it took to find them is plotted.

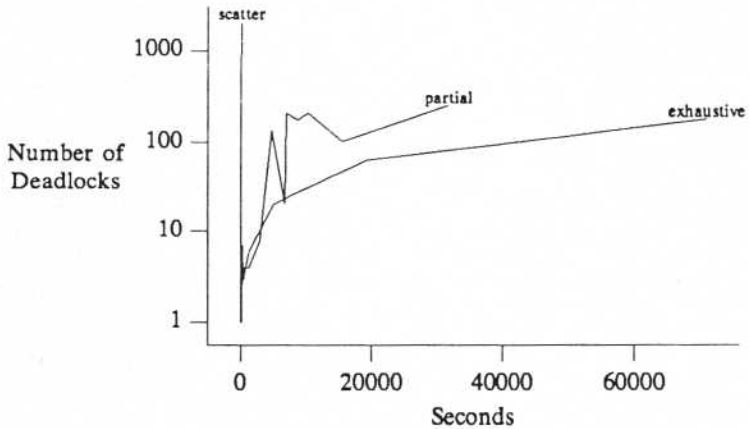

Figure 2 Deadlocks vs Time

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 6 hours of CPU time. Alternatively, the first 100 steps can be traced with a scatter search in roughly 30 seconds of CPU time.

The time required to find the first error and the minimum search depth required to trace it are both favorable for the scatter search method. The exhaustive search is too slow and the runtime of the partial search method too erratic to be useful.

We have noticed that, compared to the partial search method, the exhaustive search method can end up analyzing states twice if the second time a state is encountered it is found higher up in the state space tree. To check just how much double work is caused by these overlaps, Figure 3 compares the number of unique states to the total number of times that a previously analyzed state had to be analyzed again (dotted line). At a search depth of 80 steps the number of states searched is almost four times larger than would be required in a minimal search. For the test protocol this means that a search up to approximately 90 steps would be feasible if the redundancy of the overlaps could be avoided completely. The redundancy therefore has a noticeable effect, though not nearly as large as the effect of a change in the search discipline.

*) It is rather doubtful what a protocol designer could do with a list of over 2,000 error reports from a tracer, even granting that it can be produced in less than 4 minutes.

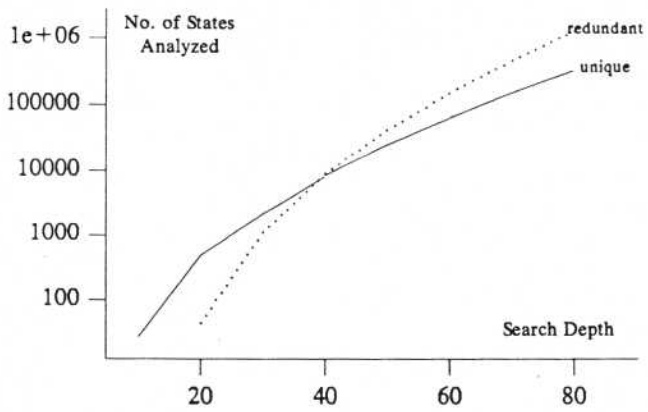

Figure 3
Redundancy in
Full Search

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 may have to 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 4. The time efficiency is expressed in the average number of states analyzed per second for each analysis run.

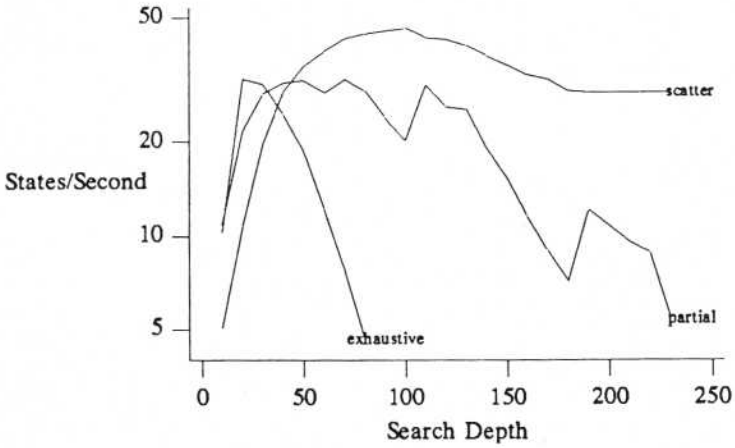

Figure 4 Time Efficiency

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, an optimal speed is reached when the state space contains approximately 1000 states.

3. RESTRICTING RUN TIME

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.

3.1 Tracing Priorities

Another method for reducing the run time of an analysis 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 are necessarily relevant, and some may be ignored without affecting the scope of the search [Holz '85b]. If the requirement that the analysis be complete is abandoned the method can be generalized by assigning priorities to the different types of actions than can occur.

If we have 'N' concurrent processes, at each node in the state space tree the tracer can simply decide to ignore any 'M <= N' processes in a partial 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' a simple priority scheme determined which process would be executed next. Highest priority was given to internal actions. At the next level we placed receive actions, since these tend to bring the system closer to a deadlock state with empty channels. A lower priority was given to send actions, and a lower priority still to channel timeouts. Timeouts were 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 really 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.

3.2 Queue Sizes

The capacity of a communication channel for holding messages can have an important effect on the size of a state space. Note that a channel 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, 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 4 the buffer sizes were restricted to two slots. Figure 5 shows the effect of a variation in the number of slots between 1 and 4 for both the exhaustive and the scatter search.

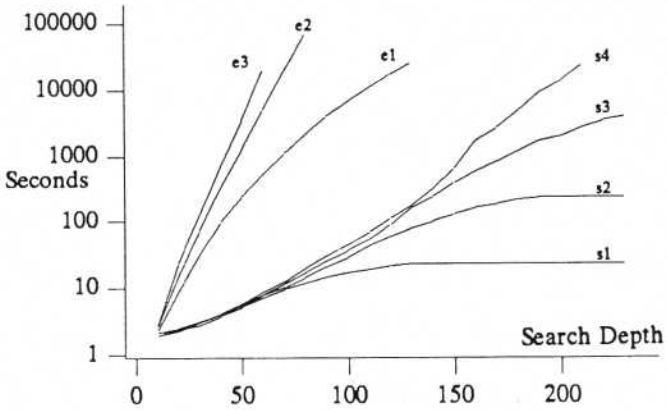

Figure 5 Effect of Queue Sizes on Runtime

s - scatter search; e - exhaustive search; 1,2,3,4 - queue sizes

4. RESTRICTING THE SPACE SPACE

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. Storing other states can avoid double work, but does not affect the scope of the analysis as such.

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 state. Note that the number of states stored in the cache can roughly be halfed without noticeable effect on the runtime or the total number of states created.

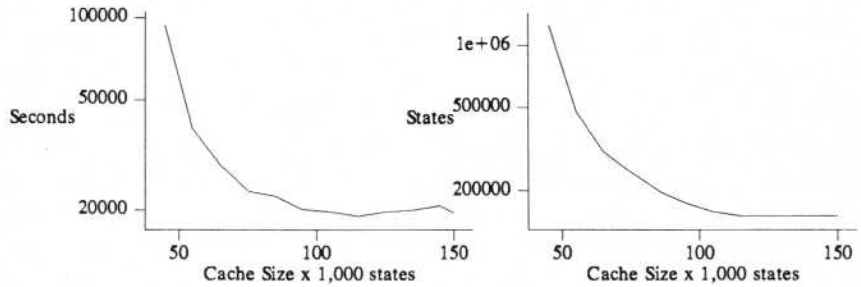

Figure 6 Size of State Space Cache

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 [Holz '85ab]. It is the strategy used in the test of Figure 6.

5. CONCLUSIONS

The protocol tracing method described in this paper, **scatter searching** , is based on the assumption that in a design phase a protocol is typically known to contain errors. The protocol tracer is meant to find a representative subset of these errors in as little time as possible. 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. We have also shown that the depth first search technique used in the experimental tracer allows for searches in incompletely stored state spaces, thus alleviating the memory requirements of protocol analyses.

6. REFERENCES

[Blu'82] Blumer, T.P., and Tenney, R.L., "A formal specification technique and implementation method for protocols," Computer Networks, Vol 6, (1982), No. 3, pp. 201-219.

[Bra'78] Brand, D., and Joyner, W.H. Jr., Verification of protocols using symbolic execution. Computer Networks, Vol. 2 (1978), pp. 351-360.

[Bra'80] Brand, D., & Zafiropulo, P., "Synthesis of protocols for an unlimited number of processes," Proc. Computer Network Protocols Conf., IEEE 1980, pp. 29-40.

[Cun'81] Cunha, P.R.F., & Maibaum, T.S.E., "A synchronization calculus for message oriented programming," Proc. Int. Conf. on Distributed Systems, IEEE 1981, pp. 433-445.

[Haj'78] Hajek, J., "Automatically verified data transfer protocols," Proc. 4th ICCC, Kyoto, Sept. 1978, pp. 749-756.

[Holz 84a] G.J. Holzmann, "The Pandora system - an interactive system for the design of data communication protocols," Computer Networks

[Holz'85a] Holzmann, G.J., "Trace - performance measurements," AT&T Bell Laboratories, internal report, Jan. 1, 1985, 29 pgs.

[Holz'85b] Holzmann, G.J., "Tracing Protocols," AT&T Technical Journal, Vol 64, No. 12, (December 1985).

[Holz'85c] G.J. Holzmann, "Automated protocol validation in `Argos,' assertion proving and scatter searching," 1985, submitted for publication, available from the author, 23 pgs.

[IFIP'80-'85] Proc. IFIP-INWG/WG 6.1, Int. Workshops on Protocol Specification Testing, and Verification, North Holland Pub., Amsterdam, 1980-1985.

[Kur '85] Kurshan, R.P., "Proposed specification of BX.25 link layer protocol, AT&T Technical Journal, Vol 64, No. 2, (Feb. 1985), pp. 559-596.

[Rama '85] Ramamoorthy, C.V., "?", IEEE Trans. on Software Engineering, Vol. SE-11, No. 9, (Sept. 1985).

[Wes '82] West, C., "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.

[Yem '83] Yemini, Y., and Nounou N., "Cupid: a protocol development environment," Proc. 3rd IFIP/WG 6.1 Int. Workshop on Protocol Specification, Testing and Verification, Zurich, Sw., May-June 1983, pp 347-357.

[Zaf '80] Zafiropulo, P., West, C.H., Rudin, H., Cowan, D.D., and Brand, D., Toward analyzing and synthesizing protocols, IEEE Trans. Commun. COM-28, No. 4, (1980), pp. 651-661.