On Limits and Possibilities of Automated Protocol Analysis

Gerard J. Holzmann

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

It is not likely that many traveling salesmen can be discouraged from their job by a lecture on its complexity [9]. Not surprisingly, writers of automated protocol analyzers are much the same. The problem of determining whether an arbitrary message passing system contains deadlocks is PSPACE-complete at best (for bounded queue lengths) [7-9]. Yet for any given formal analysis model it is easy to derive state space exploration routines that can find such errors with certainty – given a sufficient amount of time and space. In practice, therefore, one of the main problems is to optimize the speed and memory usage of an automated validator. To phrase it differently: it is not hard to validate protocols, it is hard to do it (sufficiently) fast.

In reachability analyses, the limits of what can be analyzed in practice can be moved substantially if the traditional finite state machine model is abandoned. To illustrate this, we introduce a simple symbolic execution method based on vector addition. It is extended into a full protocol validator, carefully avoiding known performance bottlenecks. Compared with previous methods the performance of this validator is roughly two orders of magnitude in speed faster and allows validation of protocol systems up to 10^6 states in only minutes of CPU time on a medium size computer.

Invited paper 7th IFIP WG6.1 Int. Conf. on Protocols Specification, Testing, and Verification, Zurich, Switzerland, 1987. North-Holland Publ., pp. 137-161.

1. Introduction

Many different theories have been suggested for the automated analysis of distributed systems. The more elaborate methods can verify the correct working of a distributed system for quite subtle correctness criteria. A question that is rarely asked or answered, however, is which problems can in reality, within practical limits, be verified with these tools. There is a subtle balance between analytical power and performance that for the time being (pending substantially faster hardware) tends to make the more elaborate systems useless in practice. The purpose of this paper is to show that the limits of what we thought could be analyzed in practice can be moved substantially if we break away from elaborate theories and build an analyzer that deliberately avoids the known run-time performance bottlenecks.

A simple analysis method, that has proven to be ultimately suitable for performing automated validations is symbolic execution [3,4,5,16,17]. It is probably safe to say that symbolic execution in one form or another is at the core of most existing and planned automated protocol validation systems today. It is the basis of all CSP [10] and CCS-like [15] evaluation systems, the Pandora algebraic validation system [11,12], the S/R validator [1], temporal logic based systems such as the Carnegie-Mellon model checker [6], etc. The pitfalls of symbolic execution are equally well known: the size of the state space and the time it may take to inspect all reachable states in it can rise dramatically with the problem size [13]. There are three answers to this problem:

- reduction (exploiting equivalences),
- truncation (being more selective in what is analyzed), and
- efficacy (doing more work in less time).

With the first technique we try to partition the set of reachable system states into equivalence classes and direct the analyzer to inspect just one state from every class. It is as yet an open problem how this can best be accomplished.

With the second technique we content ourselves with inspecting only a fraction of the full state space

but attempt to select that portion that is most likely to reveal errors [13,14].
The importance of the third technique is easily underestimated, though admittedly it seems to be the least inspired method. Yet the performance of an analyzer can usually be improved by orders of magnitude by carefully studying, for instance, memory usage patterns and writing special purpose allocators. Frustrating as it may be, the effect of such tuning techniques can easily outweigh the effect of the other two techniques combined.

The performance of a validator can be expressed in the number of system states it can generate and analyze per unit of time. To be able to measure optimal performance precisely, the next section will introduce a model for protocol validation that can help to avoid performance bottlenecks in state generation and state checking.

2. A Sample Validator

Consider the following analytical model *M*, which is defined as a collection of states and transition rules.

$$M = (S,T)$$

The state set *S* has one *current state*. Each state is completely specified by a vector of values.

$$s = \{ v_0, v_1, v_2, ..., v_n \}.$$

The protocol designer may attach specific meanings to certain values, such as the control flow state of a process, the value of a variable or the length and contents of a message queue, but to the model the significance of the individual values is irrelevant. A *transition rule* consists of a predicate and an action.

$$t = \{ p, a \}$$

The predicate is a Boolean function of the state vector, and the action defines a transformation of the state vector. Assume we have a protocol with two processes, one of 6 states and one with 4 states. Each process has a message queue of one slot, and each queue can be empty or contain one of two different messages. The state vector in our model will have four components with a total length of

$$\lceil \log(6) \rceil + \log(4) + \lceil \log(3) \rceil + \lceil \log(3) \rceil = 9 \text{ bits.}$$

Theoretically the model system can be in $2^9 = 512$ different states, of which $6*4*3*3 = 216$ are actually relevant to the protocol being modeled. The transition rules further constrain the set of possible executions and the set of reachable states. If we define the transition rules of an alternating bit protocol [2] on our example system (the above process and queue sizes were chosen to match that problem) the set of reachable states in an exhaustive analysis is only 65.

Given the definition of the alternating bit protocol, let us derive a description in terms of state vectors and transition rules. In *Argos* [13,14] a simple version of the protocol may look like:

```
proc sender
{       queue sender[1];

        do
        ::      receiver!msg1;
                do
                :: sender?ack1 –> break
                :: sender?ack0 –> skip
                :: sender?timeout –> receiver!msg1
                od;
                receiver!msg0;
                do
                :: sender?ack0 –> break
                :: sender?ack1 –> skip
                :: sender?timeout –> receiver!msg0
                od
        od
}
```

```
proc receiver
{       queue receiver[1];

        do
        :: receiver?msg1 –> sender!ack1
        :: receiver?msg0 –> sender!ack0
        od
}
```

The language defines for each statement whether it is *executable* or not. The send statement "sender!ack1", for instance, means: "attach the message *ack1* to the queue named *sender*". It is executable only if the queue addressed is not full, and it blocks if it is. Similarly, the receive statement "receiver?msg0" means: "delete the message *msg0* from the head of queue *receiver*". It is executable if the queue contains the right message in its first slot. It blocks in all other cases.

The only control flow construct used here is the repetitive construct *do :: od.* It consists of a list of *options*, each preceded by a *::* flag. The do statement is executable if at least one of the options is, and an option is executable if its first statement is executable. If more than one option can be executed, one may be picked at random. After the completion of an option a *do* structure is repeated. The only way to terminate it is to execute an explicit *break* statement or *goto* jump. But enough about the language. It is easy to recognize the six relevant states in the sender process and four in the receiver (the fourth state is an unreachable end-state after the *do* statement). The state vector will look as follows:

$$s = \{ v_{sp}, v_{rp}, v_{sq}, v_{rq} \}$$

where we can interpret v_{sp} as the state of the sender process, v_{rp} as the state of the receiver process, v_{sq} the state of the sender queue and v_{rq} the state of the receiver queue. Initially we have

$$s_0 = \{ 0,0,0,0 \}$$

There are three transition rules for the initial system state: the receiver process defines two (there are two options in the *do* structure) and the sender process defines one. The one predicate for the sender process can be written as follows, where dashes indicate don't care conditions:

$$p_0^1 = \{ (v_{sp} \equiv 0), -, -, (v_{rq} \equiv 0) \}$$

It states that the sender process must be in the initial state 0 and the receiver queue must be non-full, in this case empty since the queue lengths are 1. An empty queue is encoded in the model with a zero in the first slot, and a non-full queue with a zero in the last slot (in this case the first slot is also the last slot). If we also conveniently assume that the receiver queue encodes *msg1* with an integer number 1 and *msg0* with an integer 2, the corresponding action can be defined as a vector to be added to s_i:

$$a_0^1 = \{ 1,0,1,0 \}.$$

so that

$$s_1 = s_0 + a_0 = \{ 1,0,1,0 \}.$$

The predicates for the two transition rules defined by the receiver process in the initial state are:

$$p_0^2 = \{ -, (v_{rp} \equiv 0), -, (v_{rq} \equiv 1) \}$$

with an action vector, to be added to the state vector:

$$a_0^2 = \{ 0,1,0,-1 \}.$$

and

$$p_0^3 = \{ -, (v_{rp} \equiv 0), -, (v_{rq} \equiv 2) \}$$

with action

$$a_0^3 = \{ 0,2,0,-1 \}.$$

using the same encoding in integers of the two messages that can be appended to the receiver queue.

The predicate for a timeout is also readily defined: the timeout on queue *sender* is enabled whenever that queue is empty (and process sender is in the right control flow state). A simple heuristic could also be used to restrict the analysis to more likely cases of timeout malfunctioning: a timeout is enabled whenever the queue addressed is empty and no normal send or receive action is executable

elsewhere in the system [13].

The idea will be clear. All that is missing to make this analyzer work is a preprocessor that can translate formal specification into a state vector and transition rules, and a runtime environment that can evaluate the predicates and perform the vector additions – both fairly standard programming jobs. State generation is a fast two-step process: the evaluation of a predicate, possibly followed by a single assignment to the state vector. The method is general enough to cover also problems that are traditionally considered to be hard in writing an automated analyzer, such as value transfer, variables, generalized flow control.

3. Extensions

The above analyzer can already perform symbolic executions of protocols such as alternating bit, Arpanet 3way handshake, CCITT recommendation X.21, etc.[*]. But symbolic execution as such is not enough, we are interested in errors. Deadlocks are defined in the model as states in which all predicates are false simultaneously. Small protocols yield their errors easily to this analyzer. For larger test cases, though, there still is a problem. One problem is to determine when to terminate the analysis: when do we know that we have exhausted all possible executions? It is easy to maintain a list of all state vectors that have been encountered during the search and check each newly created state against the states in the list to guarantee that we are not doing double work. For small protocols, with say up to a few thousand reachable states, this is trivial. For medium sized and larger protocols it is not. One of my standard test protocols, for instance, requires a state vector of 395 bits. The actual number of reachable systems states is not anywhere near 2^{395}, but still in the order of 2^{23} or eight million states. There are two major problems in handling protocols of this size [†].

- state matching (to avoid double work), and
- state storing.

Eight million states of nearly 400 bits each takes some 400 Mb of memory (not counting any of the overhead required for implementing a convenient state space structure). You will need a fair sized disk to store this information. The first problem is even worse though. Of course it is not necessary to check every newly created state against all previously analyzed states. The states can be accessed through hash functions that can reduce the number of states to check for matches to only a small number. But consider this. Although the protocol strictly defines how successor states are created, the new state vector could match literally any single one of the states in the state space (since the action part defined by the protocol is unrestricted). As far as state matching is concerned, new states are created at random and may require a random portion of the (400 Mb) state space to be checked for matches (selected by the hash functions). If the state space is maintained on disk, no disk-block caching discipline will be able to optimize disk access: at each symbolic execution step another random block is needed, and it will be hard to avoid one real disk read/write operation per execution step in the larger protocols. This alone will slow down the analysis to about 100 steps/second. In practice it turns out that the calculation of hash values, and the actual state matching will take away another order of magnitude, which makes the analyzer run at roughly 10 steps/second. With eight million states to analyze this means a runtime of ten days or more.

4. Fast State Space Matching

Fortunately, the analyzer based on state vectors described in section 2 points the way to an interesting solution to the runtime complexity problem.

The automated validators that have been described in the literature all seem to run at the same basic speed that we came up with at the end of the last section: 10 to 20 steps per second. In part this is due to the amount of work required to compare states, in part to the amount of work required to interpret the protocol specification to generate successor states. We already solved the second half of this problem by defining a simple and fast vector addition system that lets us do the hard interpretation work at compile time instead of at runtime. The state space matching problem remains. Without state space matching, the analyzer I have described in section 1 can run at an unprecedented speed of 5000 steps/second on a medium size machine (VAX/750). If only we could make the state space checking cheap we would have a super fast protocol validator. There turns out to be an acceptable solution.

---

[*] The analyzer, with the additions described in this and the next section, takes less than 0.5 seconds on a DEC VAX/750 to exhaustively validate these protocols.

[†] Note carefully that this is not an extravagant example. Any protocol of practical relevance will have a complexity that is similar or worse.

The state of the protocol is given by a vector of arbitrary values. We can interpret this vector any way we want. The values have significance to the protocol, but are mere bits to the analyzer. Interpreting the state vector as a mere bit vector means that we can interpret it as a *number*. For state vectors up to 25 bits this gives us an arbitrary number between 0 and 32 million. Most machines easily have 4Mbyte (32Mbit) of main memory available, so we can allocate a array of 4 million bytes and let the *position* in that array (and within a byte) uniquely define a state. Note that we never store the 25 bits required to describe the state: it is implicitly defined by the address. If we analyze a state we can set a flag in the array. If we match for a state we check the flag. There is *no* searching and *no* real matching: if the index number defined by two states is equal the states are equal. It is as simple as that. Implementing this I obtained analysis speeds of 2000 steps per second *with* "state matching". Analyzing 8 million states now takes little more than an hour on a VAX/750 (or less than 3 minutes on a Cray).

If the bit vector is larger than 25 bits (as in my larger test case), we need a fast way to cast the bit vector into the 25 bits we have, using a good hash function. The state space is usually very sparse so this method works remarkably well over a wide range of protocols. Even when hash conflicts are not corrected the method will continue to work. Note that a hash conflict can cause the analyzer to assume erroneously that a state was previously analyzed and need not be analyzed again. It will truncate a search (the second method to battle complexity mentioned in the introduction) but will never cause spurious error reports. For the test case, in runs up to one million reachable states, hash conflicts occurred in 0.2% of the cases, making the coverage of the partial run 99.8% [□]. How much the search is truncated is quite appropriately dependent on the difference between the size of the state array and the length of the real state bit vector.

Not all has been said about this analysis method, but the above will suffice to illustrate the main point: the limits of what can be analyzed with an automated validator can be moved, and as the above indicates, can be moved quite substantially, though they cannot be removed altogether.

5. Conclusion

The performance of automated protocol validation systems is an often overlooked issue, but most likely to be one of the main problems that prevents a more general use. With the sample validator discussed in this paper protocol descriptions generating in the order of 10^5 to 10^7 system states are analyzed conveniently in minutes of CPU time on a smaller machine (VAX/750 or equivalent), or in seconds on a larger machine (Cray or equivalent). Most likely this covers a large fraction of the protocols one would be interested in in practice. Still larger protocols are beyond the capabilities of the automated validators, and it looks like they will be for some time to come.

[□] Repeating a run with an alternate hash function can recover also the 0.2% missed.

References

[1] Aggarwal, S., Kurshan, R.P., et al., IFIP, WG 6.1, 3rd Int. Workshop on Protocol Specification, Testing and Verification, North-Holland Publ., Amsterdam, 1983.

[2] Bartlett, K.A., Scantlebury, R.A., and Wilkinson, P.T. "A note on reliable full-duplex transmission over half-duplex lines," "Communications of the ACM" , Vol. 12, No. 5, (1969), 260-265.

[3] Bochmann, G., & Sunshine, C.A., "Formal methods in communication protocol design," IEEE Trans. on Communications, Vol. COM-28, No. 4, April 1980, pp. 624-631.

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

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

[6] Browne, M.C., Clarke, E.M., Dill, D.L., & Mishra, B., "Automatic verification of sequential circuits using temporal logic," IEEE Trans. on Computers, Vol. C-35, No. 12, December 1986, pp. 1035-1043.

[7] 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.

[8] DeTreville, J., "On finding deadlocks in protocols," March 22, 1982, unpublished technical memorandum, Bell Laboratories.

[9] Garey, M.R. and Johnson, D.S., "Computers and intractability," Freeman Press, San Fransisco, (1979), 340 pgs.

[10] Hoare, C.A.R., "Communicating sequential processes," Comm. ACM, Vol. 21, No. 8, August 1978, pp. 666-677.

[11] Holzmann, G.J., "A Theory for protocol validation," IEEE Trans. on Computers, Vol. C-31, No.8, (1982), pp. 730-738.

[12] Holzmann, G.J., "The Pandora system – an interactive system for the design of data communication protocols," Computer Networks, Vol. 8, No. 2, (1984), pp 71-81.

[13] Holzmann, G.J., "Tracing Protocols," AT&T Technical Journal, Vol. 64, No. 10, (1985), pp. 2413-2434.

[14] Holzmann, G.J., "Automated Protocol Validation in Argos, assertion proving and scatter searching," IEEE Trans. on Software Engineering, Vol. 13, No. 6, June 1987.

[15] R. Milner, "A Calculus for Communicating Systems," Lecture Notes in Computer Science, **92**, (1980).

[16] 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.

[17] 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.