An improved protocol reachability analysis technique — Source link

Gerard J. Holzmann

Institutions: Bell Labs

Published on: 01 Feb 1988 - Software - Practice and Experience (Wiley)

Topics: Finite-state machine, State space, State vector, Symbolic execution and Reachability

An Improved Protocol Reachability Analysis Technique

GERARD J. HOLZMANN

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

An automated analysis of all reachable states in a distributed system can be used to trace obscure logical errors that would be very hard to find manually. This type of validation is traditionally performed by the symbolic execution of a finite state machine (FSM) model of the system studied.

The application of this method to systems of a practical size, though, is complicated by time and space requirements. If a system is larger, more space is needed to store the state descriptions and more time is needed to compare and analyze these states. This paper shows that if the FSM model is abandoned and replaced by a ***state vector model*** significant gains in performance are feasible, for the first time making it possible to perform effective validations of large systems.

Software, Practice and Experience, Vol. 18, No. 2, pp. 137-161, 1988.

INTRODUCTION

It is notoriously difficult to define, or even to understand, the behavior of a system of interacting asynchronous processes. Improbable sequences of events that lead to errors are easily overlooked by a protocol designer. If such errors occur in practice, they will be virtually irreproducible and very hard to fix. A distinct advantage of an automated validator is that it has no preconceived notions about what is meant or what is likely in a protocol definition: it examines possibilities not probabilities. It can patiently perform the analysis for millions of cases, faster and more accurately than could ever be done by hand.

Many different ways have been suggested for performing automated protocol validations.[1,2] The most successful techniques are based on some form of reachability analysis by the symbolic execution of a formalized description of the protocol. A first method of this type was the duologue matrix analysis.[3,4,5] Though restricted to modeling interactions of just two non-cyclic processes, its usefulness was demonstrated quite convincingly with an analysis of CCITT draft recommendation X.21.[6] The technique was expanded for multiple processes and for cyclic behavior into the 'perturbation' analysis method.[7,8,9] Inspired by these results many variations of perturbation analysis have been developed and applied to formal models. The technique for performing automated validation is now well established. Yet all validation systems have one main problem in common.[9,10] Even the diligence of an automated validator is generally insufficient to allow for an exhaustive validation of complex protocol systems. The number of states is simply too large and the analysis too slow.

The existing methods can comfortably analyze in the order of 10^4 states at a speed of roughly 10 to 50 states per second of CPU time on a medium size machine.[6,9,10,11] Many protocols of a realistic size, however, are beyond the scope of these tools. Even seemingly minor changes in a protocol, such as the extension of a buffer size, the addition of sequence numbers to messages, or the introduction of transmission errors, can increase the number of reachable states by orders of magnitude.

The problem of finding deadlocks in protocols has been shown to be PSPACE-complete at best (undecidable for unbounded message queues).[12,13,14,15] Though this result will not prevent the development of new validation methods, it should of course play an important role in their design. In general, a method that relies solely on a completely exhaustive search of a state space is doomed to fail. There will always be cases where only a partial probing of a complete state space will be feasible. The best we can do in such cases is not to prove the *absence* of errors with certainty, but their *presence*, should they exist, with high probability.[10]

The method introduced here improves over traditional validation methods in three ways:

- it increases the speed of analysis (on a medium size system) from about 50 to the order of 10^3 states per second,
- it increases the size of systems that can effectively be analyzed exhaustively from 10^4 to the order of $10^6 - 10^7$ states, and
- without performance penalty the same algorithm can perform partial searches in still larger systems, locating errors with high probability.

In section 2 we first consider the traditional state space exploration methods based on FSM models in more detail, and discuss the nature of the time and space complexity issues. Based on some general observations we then develop a *state vector model* that avoids a number of performance bottlenecks. State space handling in the state vector model is discussed in detail in section 4. Section 5 discusses a method to translate abstract protocol specifications into state vector descriptions and section 6 discusses the results of a first application of the new method to some larger protocols.

THE TRADITIONAL MODEL

It is fairly straightforward to translate a protocol specification into a finite state machine (FSM) model, where each asynchronous process is coded as a separate FSM. The model can be extended with message queues and variables.[10,16] If the size of the queues and the range of the variables is bounded the system remains finite and can in principle be analyzed exhaustively by enumerating the reachable states. In general the transitions in such systems are conditional, for instance on the presence of an input message or on some boolean condition of local variables. In each system state any number of transitions may be simultaneously executable: one or more possible moves in each process. The system is therefore nondeterministic. To analyze the system exhaustively the nondeterminism must be resolved and all possible moves must be explored in all possible orders. There are a few observations we can make.

- The state space is *sparse*. Compared to the number of potentially reachable states (the Cartesian product of the state sets of all asynchronous processes) the set of effectively reachable states is usually small. A ratio of only 1 reachable state for every 10^9 states in the product set is no exception.[10,17]
- The state space is *tightly connected*. Because of the nondeterminism, states are usually reachable via many different execution paths that differ only in the way the executions of the asynchronous processes are interleaved.
- Largely independent of the formal model in which the protocol is defined, the state generation process in an automated validator is a basic two step process: evaluate a *condition* and, depending on the outcome, perform an *action* (i.e. a state transition).
- Finally, by measurement we can show that the most time-consuming operation in a protocol validator is not state generation or analysis but state *storage* and state *comparison*: determining whether or not newly generated states, and thus all their successors, have been generated and examined before. Without such checks a validator will run substantially faster, but will end up analyzing the same states over and over again, generally more than canceling the effect of the speed-up.[10]

Let us first make more precise what is meant by the term ‘state.’ In a finite state machine model extended with variables and message buffers, a *system state* consist of three parts:

- a control flow state for each FSM in the model,
- the current values of all global and local variables, and
- the complete contents of all message queues.

To allow for state matching each system state is stored somewhere in memory where it can be retrieved quickly, for instance via hash functions. The larger the system is, the larger also the state descriptions become. For a new state to match a previously analyzed state all elements of this state composite

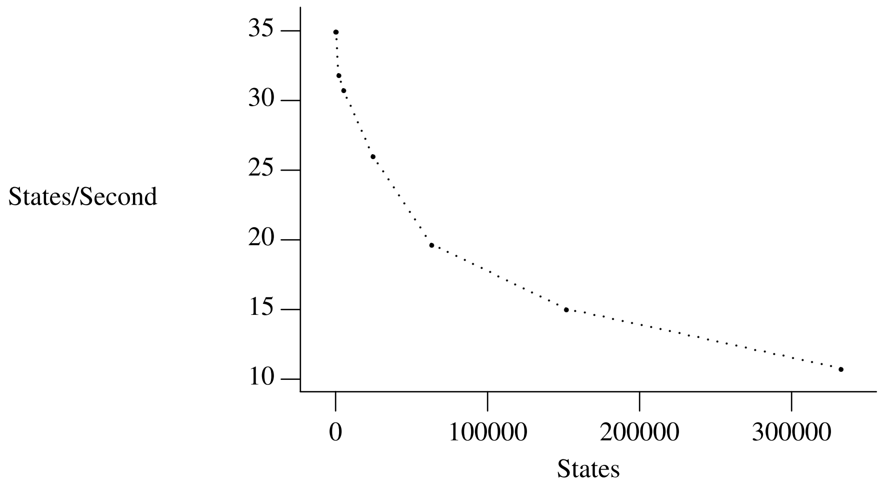

Figure 1: Overload Behavior in a Traditional Model

must be compared. Again, the larger the states are, the more time is required to compare them. Figure 1 illustrates this effect convincingly.[10] It shows the performance of a traditional validator for eight different validation runs on state spaces with between 28 and 332,527 system states. The data is from Figure 3 in Reference 10, corrected for a constant overhead factor (2 seconds) for the initialization of the state space.

For the example shown here each state required 40 bytes, so 332,527 states took up 13 Mbyte. Probing such a large state space is bound to slow down the analysis, even with the best of hashing methods. By virtue of an effective hashing policy the example validator from Figure 1, for instance, slows down by a factor of only 3 while the state space grows by a factor of 30.

Below, the four observations we made earlier will be used in the derivation of a more powerful model for automated protocol validation.

A STATE VECTOR MODEL

First we use the observation that the effective structure of a validator is not an FSM structure but a simple predicate/action model on state descriptions (state vectors). We define the vector model V as a collection of states and transition rules.

$$V = (S, T)$$

One member of the state set S is the *current state* s. Each state is completely specified by a vector of values.

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

A 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 $t \in T$ 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. When the Boolean function evaluates to *true* the action is executable. An algorithm for the symbolic execution of the state vector system is then quite simple. Transition rules are stored in an ordered set T. Two set operations are used: ‘card(T)’ returns the number of elements in T, and ‘elem(T, i)’ returns the i-th transition rule from T: a predicate function on the current state s ‘t.p(s)’ and an action ‘t.a(s).’ In pseudo C we can write a recursive procedure:

```
execute(s)
{       for (i=0; i<card(T); i++)  /* for all members  of set T      */
        {       t = elem(T, i);    /* the i-th element in set T      */
                if (t.p(s))        /* the transition predicate holds */
                        execute(t.a(s));        /* transition action */
}       }
```

State generation is fast, and not hampered by any higher level structure: the selection of a transition rule, and the evaluation of a predicate, possibly followed by an 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, and generalized flow control. It is important to note that the state vector and the predicate can be coded as mere bit-vectors or *numbers*. The predicate will test the state vector for certain properties, using bit masking to shield off parts of the vector that are irrelevant. In many cases more than one condition from the predicate can be checked in a single test (e.g. the required control flow state of a process and the contents of a message queue and the values of local variables). The action can also be coded as a vector with a bit mask, that assigns new values to portions of the state vector. Again, multiple assignments can be combined in a single action vector, making for a considerable gain in the speed of state generation.

All that needs to be added to complete the validator is the state handling: state analysis and state space maintenance. The state space handling is essential if we require that the complete validator be able to detect, for instance, the following types of errors.

- Absence of deadlock states. A deadlock state is defined as a system state where no further progress is possible, but that is not a proper end state. A proper end state is either user-defined or a system state where all processes have reached the end of their code.
- Absence of non-progress loops. A non-progress loop is an execution sequence that can be repeated indefinitely without passing through user-defined progress states.
- Absence of overrun errors. An overrun error happens when a process tries to append a message to a message buffer that is full. It indicates failure of a flow control scheme.
- Absence of improper terminations. Proper termination is a condition that is checked whenever an end state in the protocol is reached. It is a user-defined requirement on the state of queues and variables at the termination of a protocol run. Other than a deadlock state, an improper termination state is an intended state with the wrong attributes (e.g. not all data was successfully transferred from sender to transmitter).

If for the moment we restrict our attention to deadlock detection, state analysis will be trivial: a state without successors that is not a proper end state must be a deadlock. Note that it is not the validator's job to determine which states are proper end states. A preprocessor can make that determination and label state vectors appropriately (see PREPROCESSOR). The new validation algorithm is:

```
execute(s)
{       int ok=0;
        for (i=0; i<card(T); i++)  /* for all members  of set T      */
        {       t = elem(T, i);    /* the i-th element in set T      */
                if (t.p(s))        /* the transition predicate holds */
                {       ok = 1;       /* s is not a deadlock state   */
                        execute(t.a(s));        /* transition action */
        }       }
        if (!ok && !end_state(s))   /* there are no successor states */
                report_error(s);
}
```

Of course, the algorithm above does not necessarily terminate. To perform loop detection and loop analysis we must at least be able to determine if the system has returned to a previously analyzed state. This means that states must be remembered in a state space where old states can quickly be located and compared with newly generated states. Minimally we need to maintain a list of states on the single execution path that leads from the initial state to the current state. This minimal set is in fact already present in the above algo-

```
analyze()
{       struct {
                State state;    /* the saved state vectors           */
                int label;      /* return label of pseudo recursion  */
                int status;     /* saves current value of 'ok'       */
        } stack[N];             /* N defines the maximum search depth*/
        State s = initial;      /* initialize the current state      */
        int i, ok, depth = -1;  /* initial value: 1 step above root  */
down:
        ok=0;
        if (++depth >= N) goto done;    /* exceeded depth limit      */
        if (in_loop(s)) goto done;      /* s is already on the stack */
        i=0; stack[depth].state = s;    /* save s on the stack now   */
up:
        if (i < card(T))                /* the cardinality  of set T */
        {       t = elem(T, i);         /* the i-th element in set T */
                if (p(t, s))            /* transition predicate holds*/
                {       ok = 1;         /* s is not a deadlock state */
                        s = a(t, s);                /* transition    */
                        stack[depth].label = i+1;   /* for return    */
                        stack[depth].status = ok;   /* save old value*/
                        goto down;      /* the pseudo recursion step */
        }       }
        if (!ok && !end_state(s))             /* no successor states */
                report_error(s);
done:
        if (--depth < 0) return;             /* analysis is completed */
        s  = stack[depth].state;       /* restore the previous state */
        i  = stack[depth].label;       /* the pseudo return label    */
        ok = stack[depth].status;      /* restore old value          */
        goto up;
}
```

Procedure Analyze()

rithm, though not available for user probing: a complete backtrace of all states encountered in the recursion is hiding on the call stack.

By rewriting the recursive algorithm as an iterative one we can accomplish two goals: (1) the stack can be coded at user level as an explicit list of states and can be used for loop analyses, (2) we can avoid the overhead of the procedure calls implied by the recursion. This leads to procedure 'analyze()' shown below. Remember that except for the line with the test 'in_loop(s)' this is functionally equivalent to the version above. To gain extra speed 'in_loop(s)' can be a macro that expands into a simple scan through the stack-frames for a match on argument 's'. But we are not quite through yet.

The algorithm above implements a depth first search strategy in an expanding tree of system states. When run to completion, the algorithm will perform a completely exhaustive search of the state space. The stack acts as a little state space that is used to detect execution loops. States are created on the stack for every execution step *down* in the tree, and implicitly deleted for every step *up*. This means that if the same state occurs in two different execution paths, it and all subsequent states will be analyzed twice. This repeated work[10] can be avoided by maintaining a much larger data base of all reachable states ever encountered.

This is a harder problem to solve. The traditional solution is to make an exhaustive list of all state vectors that have been encountered during the search and check newly created states against the states in the list. To speed up the state matching the vectors can be stored in a hash table, e.g. with all states mapping onto the same hash value stored in linked lists. For small protocols, with say up to a few thousand reachable states, this is trivial. For large protocols it is not. It leads to the behavior illustrated in Figure 1.

Consider a protocol with eight million reachable states, where each state vector is 400 bits long. Eight million states then take 400 Mbyte of memory, not counting any of the overhead required for implementing a convenient state space structure. 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 Mbyte 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 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.

THE BIT STATE SPACE

The state vector model points the way to an interesting solution to the state space maintenance 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 50 steps per second at best. 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 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 (see PREPROCESSOR) instead of at runtime. The state space matching problem remains. So far, without state space matching, the analyzer described above runs at a speed of 5000 steps/second on a VAX-11/750.

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 26 bits this gives us an arbitrary number between 0 and 67 million. Most machines easily have 8Mbyte (67Mbit) of main memory available, so we can allocate a array of 8 million bytes and let the *position* in that array (and within a byte) uniquely define a state. We never store the 26 bits required to describe the state: it is uniquely defined by the address.

Once a state has been analyzed we turn on a 1-bit flag in the array at the address defined by the state. If we match for a state we simply check that flag. There is *no* searching and *no* real matching: if the address defined by two states is equal the states are equal. Adding this to the analyzer described above only reduces the analysis speed from 5000 to between 1000 and 3000 steps per second: still a good two orders of magnitude faster than traditional validators. Analyzing 1 million states exhaustively now takes little more than eight minutes on a VAX-11/750.

Large State Vectors

If the state vector is larger than 26 bits we need a fast way to cast it into the required address range, using a good hash function. We have already observed that the state space is usually very *sparse* so this method does in fact work remarkably well over a wide range of protocols. For a bit vector of 400 bits, we need a reduction factor of 400/26 or 1:15. The sparseness of the state space is typically better, especially for larger protocols. The ratio of 1 reachable state in every 10^9 system states, for instance, allows a reduction factor of 1:29 ($2^{29} < 10^9$). Again, the address computed (the hash *value*) is used to identify a state, and the state vector itself (the hash *key*) is not stored. This method to build a data base by using the hashvalue and throwing away the key (the original state vector) was also used by Doug McIlroy[18] to compact dictionary information for a fast spelling checker. It was first described by Bob Morris in a key paper on hashing methodology.[19]

Figure 2 summarizes the method. A preprocessor translates the FSM model into a state vector model. Each state vector is used to calculate a 26-bit hash-key, which is used directly to index a large fixed size state array.

Since state vectors are not preserved, hash conflicts cannot be corrected. Note that when hash conflicts occur the method continues to work without error, although it may not perform a completely exhaus-

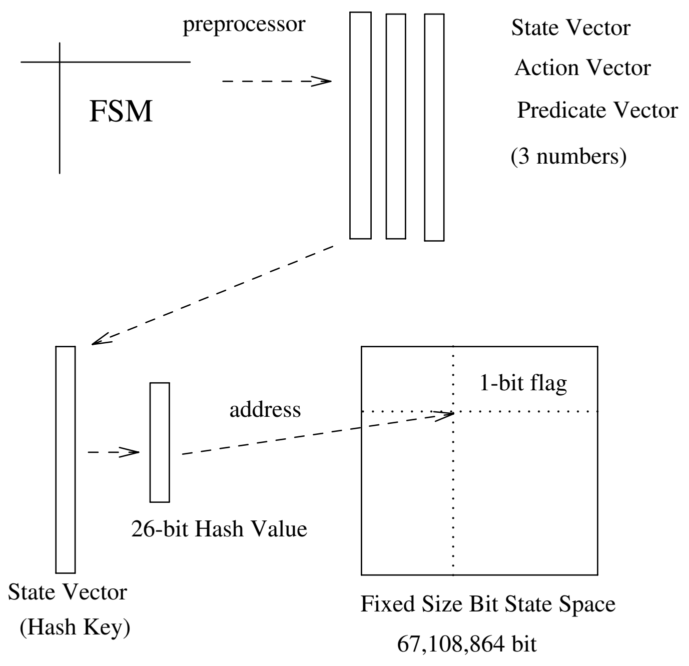

Figure 2: State Vector Model

tive search. A hash conflict causes the analyzer to assume erroneously that a state was previously analyzed and need not be analyzed again. It may truncate a search path too soon but will never cause spurious error reports. If the number of reachable states is N and the bit state space is 67 Mbit ($2^{26} = 67,108,864$ bits), we can expect hash conflicts in $100 * N/(67 * 10^6)$ percent of the cases. Some details on the effect of the hash conflicts on the coverage of the search are given in Appendix A. Note that a failure to visit a state may imply a failure to visit its successors. In principle, repeating validation runs with alternate hash functions, or alternate search strategies could each time recover a larger fraction of the states missed in a first run, thus allowing us to quickly approach an effective coverage of 100%. (An alternate hash function would move hash conflicts to a different part of the state space.) But even without duplicate runs, the method gives us precisely the behavior we need: it will analyze as many states as is feasible within the practical constraints of our system, without the penalty in performance incurred by a traditional method (cf. Figure 1). Since the state space is typically *tightly connected* we will still have a high probability of at least finding one or more variants of most error sequences. A completely exhaustive search usually traces many variants of the same errors, that differ only in the ordering of the events.

Figure 3 shows the performance of the state vector validator applied to the same protocol that was used for Figure 1. The Figure shows the results of nine different validation runs on state spaces with

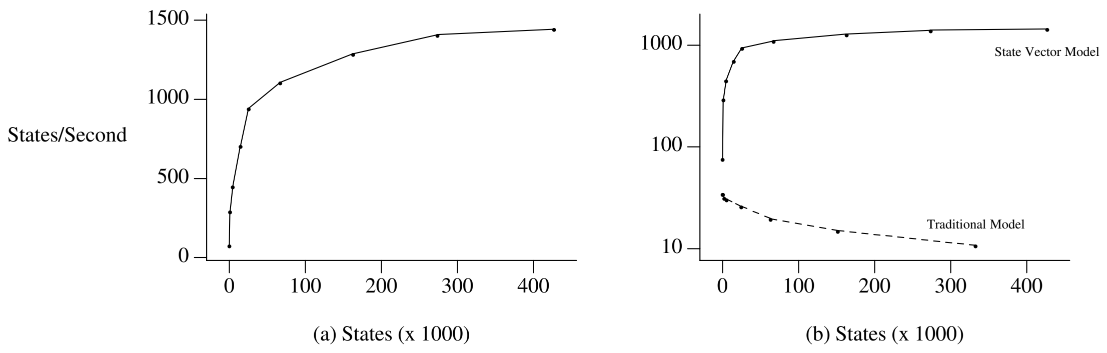

Figure 3: (a) Overload Behavior in the State Vector Model
(b) Comparison of State Vector and FSM Model (logscales)

between 54 and 426,578 system states. The best performance is obtained for large numbers of states that make full usage of the entire bit state space array: unquestionably a desirable behavior. The validations were run in a paging environment, which may contribute for the lower performance for small state spaces. Note that if the state space is small, the reachable states will be scattered over the large bit state space, initially causing random pages from the 8Mbyte state space array to be swapped into core and added to the working set of the program. Once the entire state space array is in core the validator will work at optimal speed.

THE PREPROCESSOR

To be able to use the validator, we need a preprocessor that translates formal protocol specifications into the state vectors and transition rules defined earlier. This is a fairly standard programming job: translating a specification from one formal language into another equivalent one. For the validator we have built, the preprocessor of *trace* (an FSM-based 'traditional' validator)[10,20] was adapted to produce a C program that encodes the analyzer for the protocol to be validated. The program is compiled and executed to perform the analysis. The input to the preprocessor is a formal protocol specification in the nondeterministic validation language *Argos*. A complete description of the input language can be found elsewhere.[10,20,21] Here a small example will suffice to illustrate the purpose and working of the preprocessor. Consider the following description of the traditional alternating bit protocol.[22]

```
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 otherwise. 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 double colon. The do statement is executable if at least one of the options is executable, 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. There are no variables in this example and for simplicity sequence numbers were made part of the message types (see however also the larger example in Appendix B). The state vector will look as follows:

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

It contains just four values: two for the control flow states of sender and receiver and two for the current contents of the one slot message buffer. Each process also has a message queue of one slot. Each queue can be in three different states: it can be empty or contain one of two different messages. The specification defines 6 and 4 distinct states for the sender and receiver process respectively. The state vector in our model will have a total length of

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

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 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).

In C the bit vector can be implemented as a structure of unsigned bitfields.

```
struct State {
        unsigned sp : 3;
        unsigned rp : 2;
        unsigned sq : 2;
        unsigned rq : 2;
};
```

For the state space indexing the bit structure can be interpreted as a number by refering to the state description through a union:

```
union Both {
        struct State state;
        int n;
} now;
```

The predicates can now simply read state information using the first field, as in *now.state.rq* to refer to the contents of the receiver queue. The state space handling is done by interpreting the same bits as a single integer through the second field *n*. To first test and then set the value of the *n* –th bit in state space array *SS* we can write:

```
unsigned char SS[N];    /* 8Mbyte state space array */
...
if (SS[n>>3]&(1<<(n&7))) ...
...
SS[n>>3] += (1<<(n&7))
```

The index, *n* shifted right by 3 bit positions, gives *n* divided by 8 (the number of bits per unsigned char or byte). The 3 least significant bits of *n* (*n*&7) are used to index one specific bit within the byte addressed.

If the state vector has more than 26 bits (the number of bits needed to address an 8Mbyte state space) its numeric value is first hashed into the address *n*, with a good hash function.

To allow a user to identify progress states (for loop analysis) and end states (to distinguish deadlocks from proper terminations) in the protocol description, the following labeling convention was introduced. An *Argos* labelname starting with the string ‘progress’ is taken by the preprocessor to indicate a progress state. Similarly, a labelname starting with ‘end’ indicates an end state. An example is given in Appendix B.

It is relatively easy for the preprocessor to generate the right code for detecting *deadlocks* (states where all predicates are false simultaneously and that are not labeled as end states), *non-progress loops* (execution loops that do not pass through the user-defined *progress* states), *overrun errors* (send actions directed at full buffers), or *improper terminations* (a check of a general user-defined criterion that must hold in all states that were labeled as *end* states).

The preprocessor itself is a 2000 line program written in C. The current version reads FSM descriptions generated by an existing compiler for Argos.[21] To translate the Argos description of the alternating bit protocol given above into the FSM description takes 1.9 seconds of CPU time. The FSM description is 42 lines of ASCII text. Converting that description into the state vector based validator takes another 0.8 seconds. The resulting validator is 300 lines of C text and is linked with some library routines (together worth another 150 lines of C). After compilation the exhaustive analysis of this example protocol takes 0.2 seconds. (All times given are for a VAX-11/750.)

To allow us to generate an efficient validator most validation parameters are processed at compile time and incorporated into (or deleted from) the validator generated. Loop analysis and checks for overrun errors, for instance, can be enabled or disabled at compile time. Also various different types of partial search heuristics can be defined by the user at compile time and define the structure of the validator program produced. Apart from the user selection of these validation parameters, the generation of the state vector protocol validator is completely automatic. To run the analysis, the validator produced by the preprocessor is simply compiled and executed.

APPLICATION

The analysis technique described here was applied to a number of larger protocol systems that have defeated attempts at exhaustive analyses in the past. One case is an experimental dataswitch control protocol (listed as *DSC* in the table below) with 4 asynchronous processes (with 31, 46, 7, and 6 control flow states), 5 messages queues, 33 different message types, and 7 variables. What complicated previous analysis attempts was the length of possible executions. The longest execution sequence turned out to be 2,743 steps long, and the total number of reachable states was 82,583. The complete analysis took just under 50 seconds on the VAX-11/750, and it traced a large number of errors that could be classified as cases of incomplete specification.

Another larger test case was the Universal Receiver Protocol (*URP*): a local standard for transport layer protocols on, for instance, AT&T Datakit VCS networks.[23] The protocol defines a standard receiver process that offers five different grades of service to transmitters, ranging from a raw byte stream, to a full fledged transport layer service with sliding window flow control, error detection, and error correction by retransmissions. For each grade of service an Argos model was developed (or derived from existing implementations) and tested in combination with the standard receiver. All validation runs were performed for both ideal transmission channels and loss-channels, with messages queues of 10 buffer slots each (there were 5 such queues in the model). The largest number of reachable states for the transport layer service was 5,954,172. The longest execution sequence was 66,924 steps. The validation showed convincingly that all grades of service could be realized reliably with the receiver as standardized.

As a final example, Appendix B shows an Argos model for the Cambridge Ring Protocol[24] (*CRP*). The version shown includes a *RESET* message that can be used by the sender to abort and reinitialize a session. It also includes a process named *ring* that models the transmission channel. Loss of messages, for instance, can be restricted in the analyses to queues *tosender* and *torecv*. In the version shown the channel may duplicate messages of the type *DATA*.

The specification also shows how an extra model process (*assertion*) can be used to validate more specific requirements about the expected or required order of events. In this case, process *assertion* checks that messages are always accepted by the receiver in the right order. It will cause an error (improper termination) if this user requirement is violated.

The specification as shown translates into a formal model with 4 processes of 10, 8, 16, and 19 control flow states, 6 messages queues, 4 of 6 slots each and 2 of 1 slot each, and 17 different types of messages (note that a message *a* sent to queue *b* is different from a message with the same name *a* sent to queue *c*). It takes 6.2 seconds to translate the Argos description into an FSM model, and another 4.0 seconds to derive the state vector based validator from that description. The validator generated is a C program of 1,412 lines plus a header file of 280 lines (including all the cross references to the original source to allow for legible trace-backs for error sequences). The state vector itself is 194 bits long when the buffer sizes are restricted to 2 slots, and 364 bits in the model as analyzed, with buffer sizes of 6 slots.

Exhaustive validation for this protocol shows that it behaves as advertised on both ideal and loss channels. Without any transmission errors there are just 3,241 reachable states, and the longest non-cyclic execution path is 176 steps. Adding the possibility of message loss on the queues *tosender* and *torecv* increases the number of reachable states to 6,490, with a longest sequence of 256 steps. No protocol errors result.

Table 1. Sample validations

protocol	state vector (size in bits)	number of processes	number of queues	channel type	reachable states	longest sequence
DSC	45	4	5	ideal	82,583	2,743
				loss	853,679	4,506
URP	500	3	6	ideal	3,343	320
				loss	5,954,172	66,249
CRP	364	4	6	ideal	3,241	176
				loss	6,490	256
				duplication	10,988,822	50,000
				duplication	11,300,371	1,000,000
CRP’	364	4	6	ideal	946	123
				loss	2,340	156
				duplication	1,131	123
				dupl+loss	2,652	156

Introducing duplicate messages, however, does cause errors and dramatically increases the number of reachable states. Note that a repeated message is always acknowledged with a retransmission of the last sent message by both sender and receiver. This means that as soon as a duplicate of a message has entered the system it will create a series of extra duplicates; call them ‘secondary duplicates’. If such a ‘secondary duplicate’ sparks off yet another duplicate, also that message will trigger a new response. The secondary *DATA*, for instance, triggers a secondary *RDY*, which triggers a ‘ternary duplicate’ *DATA*, and so on ad infinitum. Buffer overflows and buffer lockups are then readily possible and the corresponding error sequences are faithfully traced and reported by the validator, although no completely exhaustive search seemed feasible. The validation with message duplication on otherwise ideal transmission channels was run with search depth limitations up to one million steps (!). Most sequences turn out to be shorter than 50,000 steps: a search with that depth restriction generates near 11 million reachable system states. Allowing sequences up to one million steps long increases that number by less than 3%. Approximately 9,000 variants of the errors caused by message duplication were generated.

A solution to the problem seems relatively simple: the response to a message that is recognized as a duplicate can be deleted. The amended protocol (CRP’) turns out to have fewer reachable states: 2,652, with a longest execution sequence of 156 steps. The analyzer, however, now reports 35 non-progress loops. Some occur after the loss of a *RDY* message. Consider the case when the very first *RDY* message is lost. The sender may time out arbitrarily often, retransmitting a message *DATA*. If a *RESET* message is lost, the receiver may end up in a cycle timing out arbitrarily often, retransmitting the *RDY* message.

Most of the above validations were run with a 67Mbit (8 Mbyte) state space on a VAX-11/750. The larger tests were run with a 268Mbit (32 Mbyte) state space on a VAX/8550.

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 more general use. With the validation techniques discussed in this paper protocol descriptions generating in the order of $10^6 - 10^7$ system states are analyzed conveniently in minutes of CPU time on a medium size machine (VAX-11/750). This should cover a large fraction of the protocols one would be interested in in practice.

The method smoothly transfers from exhaustive searching to partial searching for very large state spaces. The search algorithm performs consistently well, even under overload conditions.

The state vector method is based on a number of observations on the nature of the protocol validation problem. First, we observed that the basic validation cycle is a simple two step predicate/action process. Secondly, we noted that the state space for large protocols is *sparse* which allows us to make use of a hashing discipline in which hash values are used and hash keys ignored. Thirdly, we noted that state storage and state comparison are the most time-consuming operations for large state spaces. The hashing discipline used avoids these bottlenecks almost entirely. Finally, the connectivity of the state space was exploited to preserve a high probability of tracing design errors with partial state space searches.

It should be possible to built a validation *kernel* of this type that works with many different formal protocol description techniques: the only requirement is that a preprocessor can be constructed that produces a general state vector description. As a first step towards a more general input language the modeling language Argos and the preprocessor were extended with primitives for both synchronous (i.e. rendez-vous) and asynchronous (i.e. buffered) communication. The two types of communication can now be used together in a single protocol specification.

APPENDIX A: COVERAGE

In this appendix we study the coverage of the new search discipline, using a bit space space with hashed lookups. We will do so in three different ways. First the coverage is estimated analytically, using some simplifying assumptions about the structure of the state space. Secondly we measure the performance on a small test protocol that builds a state space that conforms as closely as possible to these assumptions. The assumptions we make, however, lead to a pessimistic view of the coverage. Especially the *connectivity* of the state space is not taken into account in these first two estimates. Therefore, in a third test, we measure the coverage for an unedited practical protocol system that had been subjected to more serious validations with various protocol analyzers before.

A.1 An Analytical Estimate

Consider a state space of 2^{26} bits, and an exponentially expanding state space tree of depth *N*. If the number of nodes at a distance *n* from the root of the tree is assumed to be 2^n, the total number of states in this tree will be

$$s_N = \sum_{i=0}^{N} 2^i$$

The fraction of the state space utilized if no hash conflicts would occur and all states can be stored is

$$\frac{\sum_{i=0}^{N} 2^i}{2^{26}}$$

A randomly selected state from the tree will have distance *n* from the root with probability

$$\frac{2^n}{\sum_{i=0}^{N} 2^i}$$

The number of (successor) states that would be missed if the hash value calculated for that state would conflict with the hash of a previously generated state is

$$\sum_{i=0}^{N-n} 2^i$$

The probability of a hash conflict caused by adding a state to the tree, when the state space already contains *m* states is

$$\frac{m}{2^{26}}$$

While *m* grows, the probability of conflicts will grow, but not the average damage done per conflict. The average number of states missed by adding a new state to a state space of size *m* can then be approximated as

$$p_m = \frac{m}{2^{26} \cdot \sum_{i=0}^{N} 2^i} \cdot \sum_{n=0}^{N} \left[2^n \cdot \sum_{i=0}^{N-n} 2^i \right]$$

It is now fairly straightforward to calculate the total expected number of lost states *p* in a state space of depth *N* algorithmically. The following pseudo-C loop will do the job.

```
m = 0; p = 0;
do {
        p = $p + p_m$;
        m = m+1;
} while (m+p < $s_N$);
```

The calculated coverage percentage, based on these values of *p*, is given in the table below.

Table A1. Calculated coverage

N	p	reachable	reached	coverage
10	0.07	1023	1023	100
11	0.31	2047	2047	100
12	1.37	4095	4094	99.96
13	5.99	8191	8185	99.92
14	25.91	16383	16357	99.84
15	111.23	32767	32656	99.66
16	473.08	65535	65062	99.27
17	1986.38	131071	129085	98.48
18	8169.92	262143	253973	96.88
19	32442.86	524287	491844	93.81
20	121632.53	1048575	926942	88.40
21	419430.06	2097151	1677721	80.00
22	1305608.74	4194303	2888694	68.87

The most important observation to make is that, even though the number of hash conflicts increases only linearly with the size of the state space, the effective coverage will decrease faster, due to the tree structure.

A.2 Measurement of A Small Test Protocol

The following example process, though not a very realistic one, builds precisely the binary tree assumed in the calculation.

```
queue dummy[23];
proc bin
{
        do
        :: dummy!m1
        :: dummy!m0
        od
}
```

Table A2. Process bin

N	reachable	reached-1	reached-2	coverage-1	coverage-2
10	1023	1023	1023	100	100
11	2047	2047	2047	100	100
12	4095	4083	4095	99.71	100
13	8191	8118	8191	99.11	100
14	16383	16127	16375	98.44	99.95
15	32767	31940	32724	97.48	99.87
16	65535	63275	65363	96.55	99.74
17	131071	124738	130300	95.17	99.41
18	262143	244227	259786	93.17	99.10
19	524287	422034	511179	80.50	97.50
20	1048575	806978	979642	76.96	93.43
21	2097152	1473057	1893436	70.24	90.29
22	4194304	2644261	3558739	63.04	84.85

The results of a single analysis run are shown in the column reached-1 and coverage-1. The measured coverage for this example is smaller than the estimate. For 4 million reachable states the coverage is 63.0% instead of the calculated 68.8%. Most likely this is caused by the tendency of the hash conflicts to accumulate in one part of the search tree, towards the end of the depth first search.

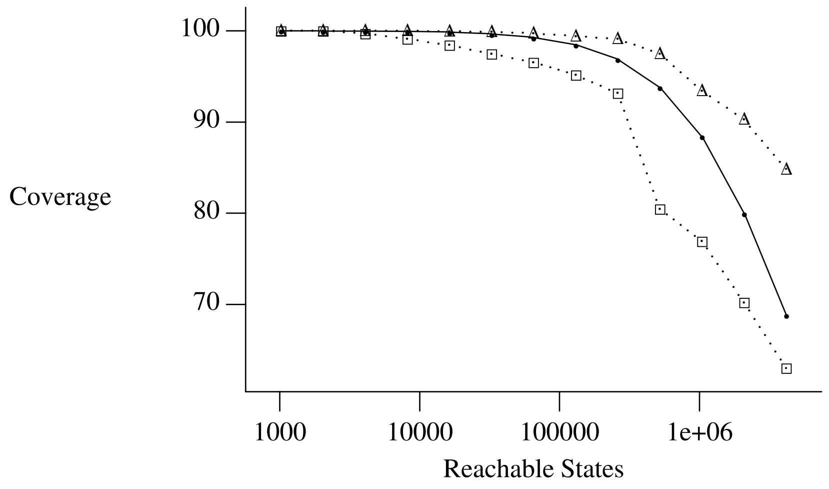

Figure A.1: Coverage

We can try to capitalize on this effect by repeating the analysis run while reversing the order of the depth-first search. The expectation is that the hash conflicts will now accumulate in a different part of the tree. The combined coverage of two such runs is shown in the table under reached-2 and coverage-2. The coverage of the run for 4 million states has improved from 63.0% to 84.8% in the second run, this time well above the calculated value of 68.8%. For comparison, Figure A.1 shows the information from the first two tables in graphical form. The drawn line is the calculated coverage. The dotted line with squares is the coverage for the artificial protocol after a single run, and the line with triangles shows the coverage after two runs.

The test protocol is atypical though, and may lead to a more pessimistic estimate of the coverage than is achieved in practice. The example, for instance, specifies just one process with only send operations. It builds a completely disconnected state space (every state occurs precisely once) while in practice state spaces were assumed to be tightly connected as a result of the nondeterminism among concurrent processes.

A.3 A Real-Life Protocol

Four extra runs were done with a 'real' protocol (a data-switch control protocol with 4 processes and 5 message queues) to compare the coverage that is achieved in practice with the above results. In each case the protocol was first run with the bit state space with uncorrected hash collisions, and then with a complete state space (with a considerable penalty in run-time), without the hash functions.

Table A3. Coverage in practice

reachable	reached	coverage
71411	70785	99.12
121493	120111	98.86
266293	262644	98.63
801570	779012	97.19

In all four runs the measured coverage is well above both the calculated coverage and the measured coverage on the artificial test protocol. In a first run it is even comparable to the coverage after the second run of the other protocol. This effect is illustrated in the graph in Figure A.2. The drawn line represents the coverage for a single run of the 'real' protocol and the dotted lines again give some points from the coverage of two earlier runs for the test protocol.

Note that for this protocol the true number of reachable states cannot be calculated, like we did for the smaller test protocol, but has to be measured with a separate exhaustive run, not utilizing a bit state space. Beyond the roughly 800k states of the last run, the computation of the actual number of reachable states became prohibitively expensive and had to be abandoned. The last point in the graph, for instance,

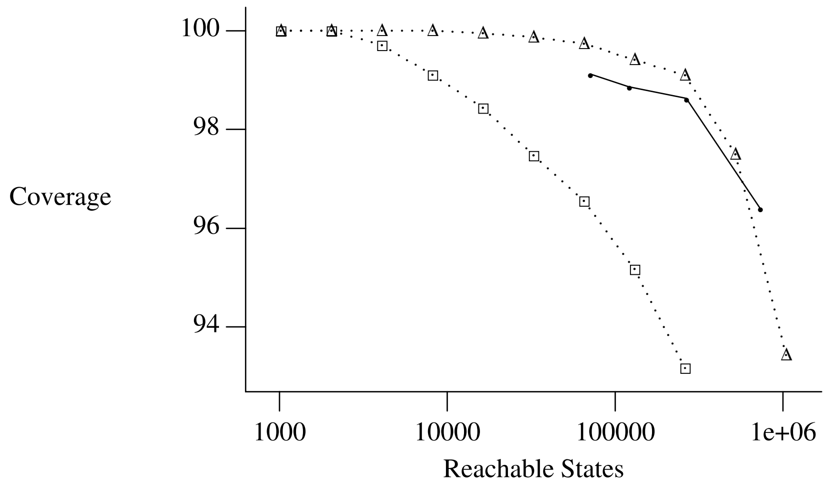

Figure A.2: Relative Coverage

required over six hours of VAX-11/750 time for the complete run with 100% coverage, and just under 10 minutes for the run with the bit state space giving a 97.2% coverage.

There is a trade-off between run-time and coverage with the new search method based on the bit state space. The choice between the two, however, will not be hard to make in practice: in the above case, we give up 2.8% of the coverage in return for a reduction of the run time by no less than 99.7%.

APPENDIX B

Process ‘ring’ models the behavior of the transmission channel. Loss errors are simulated on the two queues tosender and torecv.

Process ‘assertion’ is included in the model to verify that messages cannot be accepted out of order; the process will block on the condition ‘(m==n)’ if an out-of-sequence message was accepted, and cause a termination error. The ‘progress’ criterion used, for loop analysis, is the acceptance of a message by the assertion process.

```
/*
 *      Argos[20] Model of the Cambridge Ring Protocol[24]
 */

#define QSZ     6       /* length of all message queues */

queue sender[QSZ], recv[QSZ];

proc ring
{       queue tosender[QSZ], torecv[QSZ];
        pvar m;

end:    do      /* user-defined end state */
        :: tosender?RDY(m)    -> sender!RDY(m)
        :: tosender?NOTRDY(m) -> sender!NOTRDY(m)
        :: torecv?DATA(m)     ->        /* duplication error */
                if :: recv!DATA(m) :: recv!DATA(m); recv!DATA(m) fi
        :: torecv?NODATA(m)   -> recv!NODATA(m)
        :: torecv?RESET       -> recv!RESET
        :: tosender?RESET     -> sender!RESET
        od
}
```

```
proc assertion
{       queue accept[1], ack[1];
        pvar n = 0; pvar m;

end:    do
        :: accept?data(m) ->    ack!ok; /* handshake */
progress0:                      (m==n) -> n = (n+1)%4
        :: accept?reset   ->    ack!ok;
progress1:                      n = 0
        od
}
proc sender
{ pvar n = -1;
  pvar m;
                goto I;
E:              do
                :: sender?RDY(m) ->
                        if
/* E-rep */             :: (m == n) -> torecv!DATA(n)
/* E-exp */             :: (m == (n+1)%4) -> n = (n+1)%4;
                                if
/* buffer  ready */             :: torecv!DATA(n); goto E
/* buffer !ready */             :: torecv!NODATA(n); goto N
/* no more data  */             :: goto done
                                fi
                        fi
/* N-exp */     :: sender?NOTRDY(m) -> (m == (n+1)%4) -> goto I
                :: sender?timeout -> torecv!DATA(n)
                od;

N:              do
/* E-rep */     :: sender?RDY(m) -> (m == n) -> torecv!NODATA(n)
                :: torecv!DATA(n); goto E
                :: goto done
                od;

I:              do
/* E-exp */     :: sender?RDY(m) -> (m == (n+1)%4) ->
                                n = (n+1)%4;
                                if
                                :: torecv!DATA(n); goto E
                                :: torecv!NODATA(n); goto N
                                :: goto done
                                fi
                :: sender?NOTRDY(m) -> (m == (n+1)%4) -> skip
                :: sender?timeout -> torecv!DATA(n); goto E
                od;

done:           torecv!RESET;
                do
                :: sender?RESET -> n = -1; goto I
                :: sender?any -> skip   /* ignore */
                :: sender?timeout -> torecv!RESET
                od
}
```

```
proc receiver
{ pvar n = 0;
  pvar m;
                goto N;
E:              do
                :: recv?RESET -> goto done
                :: recv?DATA(m) ->
                        if
/* E-rep */             :: ((m+1)%4 == n) -> tosender!RDY(n)
/* E-exp */             :: (m == n) ->  n = (n+1)%4;
                                accept!data(m); ack?ok;
                                if
                                :: tosender!RDY(n); goto E
                                :: tosender!NOTRDY(n); goto N
                                fi
                        fi
/* N-exp  */    :: recv?NODATA(m) -> (m == n) -> goto I
                :: recv?timeout -> tosender!RDY(n)
                od;

N:              do
                :: recv?RESET -> goto done
/* E-rep */     :: recv?DATA(m); ((m+1)%4 == n); tosender!NOTRDY(n)
                :: tosender!RDY(n); goto E
                od;

I:              do
                :: recv?RESET -> goto done
                :: recv?DATA(m) -> (m == n) ->
/* E-exp */                     n = (n+1)%4;
                                accept!data(m); ack?ok;
                                if
                                :: tosender!RDY(n); goto E
                                :: tosender!NOTRDY(n); goto N
                                fi
                :: recv?NODATA(m) -> (m == n) -> skip
                :: recv?timeout -> tosender!RDY(n); goto E
                od;

done:           tosender!RESET;
                accept!reset; ack?ok;
                n = 0;
                goto N
}
```

REFERENCES

1. Bochmann, G. and Sunshine, C.A., "Formal methods in communication protocol design," IEEE Trans. on Comm., Vol. COM_28, No. 4, April 1980, pp. 624-631.

2. Bochmann, G., "Methods and tools for the design and validation of protocol specifications and implementations," Publ. #596, Dept. d'Informatique, Universite de Montreal, Oct. 1986, 56 pgs.

3. Zafiropulo, P., "A new approach to protocol validation," Int. Conf. on Communications, June 1977, Chicago.

4. Zafiropulo, P., "Protocol validation by duologue matrix analysis," IEEE Trans. on Commun., Vol. COM-26, August 1978, pp. 1187-1194.

5. West, C.H., "An automated technique of communications protocol validation," Int. Conf. on Communications, June 1977, Chicago.

6. West, C.H., "Automated validation of a communications protocol: the CCITT X.21 recommendation," IBM Journal of R&D, Vol. 22, No. 1, Jan. 1978, pp. 60-71.

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

8. Zafiropulo, P., West C.H., Rudin, H., Cowan, D.D., and Brand D., "Toward analyzing and synthesizing protocols," IEEE Trans. on Comm., Vol. COM-28, No. 4, (1980), pp. 651-661.

9. West, C.H., "Applications and limitations of automated protocol validation," Proc. 2nd IFIP WG 6.1, Int. Workshop on Protocol Specification, Testing and Verification, Idyllwild, Ca., May 1982, North-Holland Publ., pp. 361-373.

10. Holzmann G.J., "Tracing Protocols," AT&T Technical Journal, Vol. 64, No. 10, December 1985.

11. West, C.H., "General Technique for communications protocol validation," IBM Journal of R&D, Vol. 22, No. 4, July 1978, pp. 393-404.

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

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

14. 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.

15. Apt, K.R., Kozen, D.Z., "Limits for automatic verification of finite-state concurrent systems," Inf. Processing Letters, Vol. 22, No. 6, May 1986, pp 307-309.

16. 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.

17. Holzmann G.J., "On Limits and Possibilities of Automated Protocol Analysis," Proc. 7th IFIP WG 6.1, Int. Workshop on Protocol Specification, Testing and Verification, Zurich, Switzerland, (May 1987), North-Holland Publ.

18. McIlroy, M.D., "Development of a Spelling List," IEEE Trans. on Comm., Vo.. COM-30, No. 1, Jan. 1982, pp. 91-99.

19. Morris, R., "Scatter Storage Techniques," CACM, Vol. 11, No. 1, Jan. 1968, pp. 38-44.

20. Holzmann, G.J., "Manual for the protocol analyzer 'trace,' AT&T Bell Laboratories, Computing Science Technical Report No. 134, February 1987, 27 pgs.

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

22. 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.

23. Holzmann, G.J., "Reachability Analysis of the Universal Receiver Protocol," AT&T Bell Labs, Internal Report, March 25, 1987, 18 pgs.

24. Needham, R.M., Herbert A.J., "The Cambridge Distributed Computing System," Addison-Wesley Publ., London, 1982.