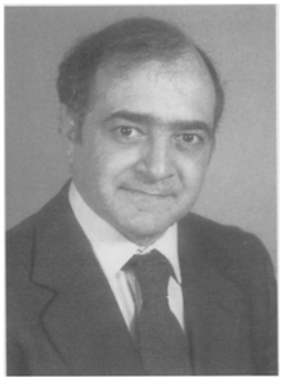
Distributed Computing (1989) 3:118–129

DISTRIBUTED COMPUTING

Probabilistic verification of communication protocols

N.F. Maxemchuk and Krishan Sabnani
AT&T Bell Laboratories, Murray Hill, NJ 07974, USA



Nicholas F. Maxemchuk received the B.S.E.E. degree from the City College of New York, NY, and the M.S.E.E. and Ph.D. degrees from the University of Pennsylvania, Philadelphia. He is the Head of the Distributed Systems Research Department at AT&T Bell Laboratories, Murray Hill, NJ, and has been at AT&T Bell Laboratories since 1976. Prior to joining Bell Laboratories he was at the RCA David Sarnoff Research Center in Princeton, NJ for eight years. Dr. Maxemchuk has been on the adjunct faculties of Columbia University and the University of Pennsylvania. He has been an advisor to the United Nations on data networking and has been on networking panels for the US Air Force and DARPA. He has served as the Editor for Data Communications for the IEEE Transactions on Communications, as a Guest Editor for the IEEE Journal on Selected Areas in Communications, and has been on the program committee for numerous conferences and workshops. He was awarded the RCA Laboratories Outstanding Achievement Award, the Bell Laboratories Distinguished Technical Staff Award, and the IEEE's 1985 and 1987 Leonard G. Abraham Prize Paper Award.

Krishan Sabnani received a BSEE degree from Indian Institute of Technology, New Delhi, India and a PhD degree from Columbia University, New York, NY. In 1981, he joined AT&T Bell Laboratories after graduating from Columbia University. He is currently working in the Distributed Systems Research Department of AT&T Bell Laboratories. His major area of interest is communication protocols. Dr. Sabnani was a co-chairman of the Eighth International Symposium on Protocol Specification, Testing, and Verification held in Atlantic City, NJ during June 1988. He is currently an editor of the IEEE Transactions on Communications and of the IEEE Transactions on Computers. He has served on the program committees of several conferences. He is also a guest editor of two special issues of the Journal on Selected Areas in Communications (JSAC) and the Computer Networks and ISDN Systems Journal, respectively.

Offprint requests to: K. Sabnani

Abstract. Complete behavior of a communication protocol can be very large. It is worth investigating whether partial exploration of the behavior generates reasonable results. We present such a procedure which performs partial exploration using most-probable-first search. Some of the ideas used in this procedure are based on a convolutional decoding procedure due to Jelinek and a performance evaluation procedure due to Rudin. Multiple trees of protocol behavior are constructed. Some results on estimating the probability of encountering an unexplored state in a finite run of a protocol are also presented.

Key words: Probabilistic analysis – Validation – Verification – Protocols – Probabilistic verification

1 Introduction

With the rapid expansion of computer networks and of the services provided, communication protocols used are becoming increasingly complex. For the proper operation of such networks, communication protocols must be free of logical errors. One way of eliminating these errors is to verify the protocols before implementing them. Verification of communication protocols has been an active topic of research for the past decade (v. Bochmann and Sunshine 1983; v. Bochmann 1975; Danthine 1983; Gouda 1984; Lam and Udaya Shankar 1984; Sabnani and Schwartz 1984; Zafiro-

pulo et al. 1983; Sabnani 1988). A majority of these attempts check the complete protocol design for some logical errors, such as deadlocks, livelocks, and functional errors. Complete verification of complex protocols is very hard (v. Bochmann and Sunshine 1983). It is worth investigating whether partial search of protocol designs generates reasonable results. Some recent attempts (West 1986; Rudin and West 1982) perform a partial check of protocol designs for logical errors.

In this paper, we present a novel way of probabilistically checking (or verifying) a protocol design. This procedure explores the most probable sections first for logical errors. Same ideas used in this work are similar to the ones used in Jelinek's convolutional decoding scheme (Jelinek 1969) and in the performance evaluation technique due to Rudin (1984). This procedure can locate a wide range of errors, such as deadlocks, livelocks, and functional errors. It always terminates if the number of reachable states is finite. But it may also terminate at several intermediate points. At these points, we can estimate the probability of encountering an unexplored state in a finite run. A run of length n means an interaction sequence with n state transitions.

In this paper, we focus only on data transfer protocols. The procedure given here can be easily generalized for general protocols. Using the procedure given here, one can make a statement that an arbitrary run of the protocol with length n (typically $n=10^9$) will have an unexplored state with a probability less than a small number such as 10^{-12}. It is done because we construct multiple trees. Each tree starts from a reachable state in which the transmitter can pick up a message from the local user. This idea is similar to the use of repetition sets (McNaugton 1966). When all reachable states of the protocol have been explored, the computation terminates. The idea of constructing trees was inspired by a performance evaluation procedure due to Rudin (1984). A similar procedure was also developed by Gouda (1978).

Our procedure can be viewed as a directed simulation in which the most likely states are investigated first and cycles through high probability states are investigated once. It explores the most probable section of the protocol once. It then explores the less probable sections of the behavior. On the other hand, a discrete event simulation examines the most probable states of the FSM repeatedly and the improbable sections of the FSM with a low probability. Therefore, the procedure given here explores the low probability sections much sooner.

In Sect. 2, we introduce a simple model for specifying protocols. A protocol is modeled as a collection of *f*inite *s*tate *m*achines (FSMs) interacting with one another using synchronous message exchanges of the type used in the programming language *Communicating Sequential Processes* (*CSP*) (Hoare 1978). Section 3 presents some intuitive background for this work. The details of our procedure are given in Sect. 4. Some examples illustrating this procedure are given in Sect. 5. Finally we make some concluding remarks in Sect. 6.

2 A model for specifying protocols

A protocol is defined as a collection of *mFSMs*: $F_1, F_2, \ldots, F_m$, where m is a positive integer. These machines interact with one another other using synchronous message exchanges similar to the ones used in the programming language *Communicating Sequential Processes* (*CSP*) (Hoare 1978). An output operation is denoted by *process1 !msg* (send *msg* to *process1*) and an input operation is denoted by *process2?msg* (receive *msg* from *process2*). The main semantic assumption about these input/output operations is that matching operations are executed simultaneously. In other words, if a process wants to execute an output, it has to wait until a corresponding process is ready to execute the matching input operation and vice-versa. For instance, if *process1 !msg* appears in *process2* and *process2?msg* appears in *process1*, they are matching operations that are executed simultaneously. Thus the input/output operations not only are a means for exchanging data between the processes but also as a means for synchronizing their execution.

Each FSM is given as a five-tuple (Σ, S, s_0, T, P) where Σ is an alphabet set consisting of the machine's input/output operations; S is a finite set of states; $s_0 \in S$ is its initial state; $T: S \times \Sigma \longrightarrow 2^S$ is a non-deterministic state transition function; and $P: S \times \Sigma \times S \longrightarrow [0, 1]$ is a probability function. P is a function of three parameters: s, the present state; l, the edge label or the CSP-type Input/output operation; and t, the next state. In other words, $P(s, l, t)$ is the probability of the state transition with label l from s to t. The function P for a deterministic state transition is equal to one. The function P for each choice of a non-deterministic state transition has a value less than 1. But, the sum of these probabilities for all non-deterministic transitions from one state with the same label must add up to 1. We further assume that each non-deterministic transition has the following restriction. Each non-deterministic transition has two choices: one choice (called the "high probability

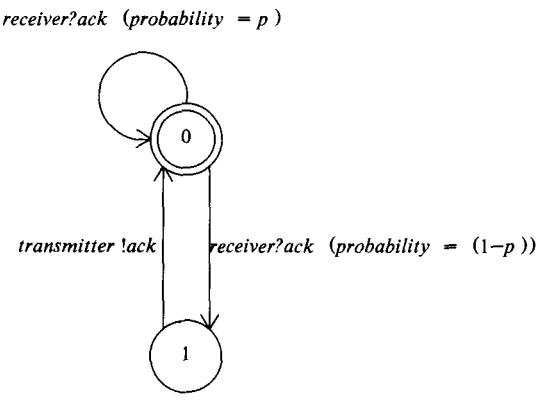


Fig. 1. The FSM model of a Lossy Communication Channel

choice") has a probability $(1-p)$ and the second choice (called the "low probability choice") has a probability p where $p \ll 1$. Such a model captures an important feature of a communication channel, namely the high probability transition corresponds to correct delivery of messages and the low probability corresponds to message loss. An example FSM with two states is given in Fig. 1 models a communication channel; it either delivers the sent message correctly with a probability of $(1-p)$, or loses a message with a probability of p.

We assume here that the errors introduced in the communication channels are independent. The communication channel loses a new message with a probability independent of whether it lost the previous message. For example, the communication channel shown in Fig. 1 returns to state O after losing a message. Since all the relevant past history of an FSM is captured in its current state, state O gives no information whether the previous message was lost or not. This assumption may not be valid for some types of communication channels, such as packet radio. In such cases, probabilities of such errors are not independent. Future research can develop partial verification procedures for such channels with dependent errors.

Estimating the value of p is not difficult in the model given here. In a more general model, estimating probabilities of improbable events, such as that of expiration of a time-out counter, would be difficult. But this is not a very serious problem as the designer of a simulator would have the same difficulty in setting up a simulator to predict the protocol's performance.

A *trace* for an FSM is defined as a sequence of labels of edges in a finite traversal of its underlying graph. The *behavior* of an FSM is defined as the set of its *traces*. Two traces for machines x and y match if the input/output operations in them match with one another. For example, the trace $x!m$; $x?n$ for FSM y matches with another trace $y?m$; $y!n$ for FSM x.

3 Intuitive explanation

Any infinitely long run of a protocol will encounter all reachable global states. A global state is a tuple of the states of all m FSMs. But, a finite run of the protocol may not encounter all states, especially the ones with low probability. Some runs of protocols are highly unlikely. For example, a run in which the communication channel loses the same message ten times is highly unlikely. For large protocols, the exploration of all global states requires a large amount of computational effort. If a partial exploration of the global FSM is done, its most probable states should be examined first.

A finite run of the protocol should encounter global states examined during the verification. If it does not, the probability of encountering an unexplored state should be very low. Therefore we must first check the most probable states of the global FSM during a protocol verification. The procedure given here does not perform a depth-first or a breadth-first search of the global FSM, but instead a most-probable-first search. This is done by computing the most probable successors of the most probable states.

We store only a small number of reachable states in a state table as described in the next section. The successors of all reachable states are computed until a terminating condition is satisfied (Theorem 2). This procedure is used because we want to estimate the probability of enocuntering an unexplored state in a finite run.

Our procedure will not explore some sections of the global FSM which have a very low probability of occurrence. These sections may contain some errors which can be of any type. The result of a probabilistic verification may be used to reduce the complexity of a protocol. Therefore, the states which are visited infrequently may be removed from the protocol. We believe that this approach will result in simpler protocol designs than those used in practice today. The protocols used in practice are very complex because they are designed to avoid all errors. There is no reason to make the protocols more reliable than the hardware or software which implements them. Therefore, states which are visited infrequently may be removed from the protocol. The procedure given here has two advantages:

- If the complete verification or exploration of the global FSM is difficult, then we can direct the

search so that we can explore the most probable sections of the global FSM. For a given amount of computational resources, the procedure given here does the best possible search.
- If a protocol error occurs in a highly unlikely section of the global FSM, then the recovery procedure such as adding a timer may not be necessary.

One recent attempt does a partial search of the global FSM (West 1986). But it does a random exploration of the global FSM, not a directed exploration as is done in our procedure. Another technique proposed by Rudin and West also performs partial exploration of the protocol behavior (Rudin and West 1982). The error modeling function is localized in a demon that could produce a predetermined number of errors. By incrementing the number of errors, larger sections of the protocol behavior can be explored. There has been some research work on verification of probabilistic algorithms (Pnueli and Zuck 1986). Such procedures do a complete verification of probabilistic algorithms.

4 Procedure

The procedure proposed here is a variation of the reachability computation procedures reported in the literature (v. Bochmann and Sunshine 1983; Zafiropulo et al. 1983). However, instead of constructing a directed graph, as in the earlier procedures, multiple tree diagrams are constructed. The reasons for this selection are given later in this section. We also describe how global states are divided into classes, a term described later in this section. Our procedure can detect a wide range of protocol errors, such as deadlocks, livelocks, and functional errors. We describe how these errors are located. The algorithm is given in Sect. 4.1, and an expression for the probability of encountering an unexplored state in a finite run as a function of the run's length is derived in Sect. 4.2. We also prove in Sect. 4.2 that the algorithm given in Sect. 4.1 eventually terminates. This procedure is illustrated by some examples in Sect. 5.

We have two alternative procedures for conducting state exploration and then estimating probability of encountering an unexplored state in arbitrarily long runs of a protocol. These two alternatives are: (1) examine all possible infinitely long sequences of protocol behavior; (2) select a small[1] set *RS* of global states and prove that whenever the protocol starts in one of these states, it always ends up in some other state contained within *RS* with a very high probability. Alternative (1) requires infinite computational and storage resources. We have used Alternative (2) here. The set *RS* chosen here for a data transfer protocol is the set of those global states in which the protocol can pick up a message from its user. A tree is built from the protocol's initial global state. Whenever a state belonging to the set *RS* is encountered for the first time during the execution of our procedure, we build another tree starting from that state. The idea is analogous to the use of repetition sets (McNaughton 1966; Sabnani 1988). The root of each tree is a global state in with the transmitter can pick up a message (or a member of *RS*). A tree is constructed by computing the successors of the root state.

The reachable states of the global FSM are divided into classes: *Class 1*, *Class 2*, and so on. The initial global state is a member of Class 1. A global state belongs to Class 1 if it is a successor of a Class 1 global state reached after performing a state transition in which two component processes make high probability matching transitions. Two matching transitions mean edges with matching Input/Output operations. Any global state in which the transmitter can pick up a message (or a member of *RS*) also belongs to Class 1. A *starting state* for Class 1 is a reachable state which is a member of RS. A *starting state* for Class 2 and higher is one which the protocol enters after a low probability transition.

A global state belongs to Class 2 if (a) it is a successor state of a Class 2 state with a state transition in which two component processes perform high probability operations or (b) it is a successor state for a Class 1 state with a state transition in which one component process makes a high probability choice and the other component makes a low probability choice.

Similarly, a global state is of Class j if (a) it is a successor state of a Class j state with a state transition in which both component processes perform high probability transitions, (b) it is a successor state of Class $j-1$ state with a transition in which only one component process makes a high probability choice, or (c) it is a successor state of a Class $j-2$ state with a state transition in which both components make low probability transitions.

The idea of dividing states into classes is inspired by the bin processing procedure used in Jelinek's convolutional decoding procedure. By dividing states into classes, we do not have to store

[1] If the the transmitter has x states and y states in which it can pick up the next message from the local user, then the size of *RS* is y/x of the global state space's size

probabilities in states. In addition, we do not have to store the tree stucture during the execution of our algorithm. The states belonging to each class are stored in an array. We could have chosen to assign actual probabilities to the global states. But it would make the required algorithm more complex than the one given here. For such an algorithm, the state with the highest probability will be expanded first. This procedure will be similar to the other convolutional decoding procedures (Peterson and Weldon 1972).

The procedure described here can check for deadlocks, livelocks, and functional errors. A deadlock is a global state which has no successors. Such states are located during state exploration.

A protocol has a livelock if it can be locked up in a small set of global stares in the same class and makes no progress towards its stated function. A livelock is a loop in the global FSM which does not contain a member of *RS* or a low probability transition. If the protocol is locked up in such a loop, then it can remain forever in this loop and make no progress towards its goal, namely data transfer. In any infinite execution, the protocol must pick up messages from the user infinitely often. If the protocol never picks up a message, then it can make no progress towards its eventual goal, namely data transfer. This is a necessary but not sufficient condition for ensuring progress. For example, the protocol can pick up messages but never deliver them. The safety property (that the protocol delivers messages correctly provided that it makes process) is checked using a procedure described later in this section.

If a loop in the global FSM contains a low probability transition, then the protocol will circulate in this loop n times with a probability less than p^n. An example is a loop in which the communication channel loses a message. Since the probability of message loss in a typical communication channel is very small, the protocol will eventually move out of this loop. Such a loop is not a livelock.

The presence of a livelock is detected by keeping track of the number of state transitions that the protocol has gone through in its path from a *starting state* in its present class to its present state in the current class. This count is stored in the global state. A global state is represented as $[(s_1, s_2, \ldots, s_m), count]$ where s_j is the state of the jth FSM F_j, and *count* is the number of state transitions (in the present class) which brought the protocol to its current state. Whenever *count* exceeds a maximum value *max*, the protocol has a livelock. An upper bound of *max* given in Theorem 1 is the number of reachable states excluding the reachable members of *RS*.

We can not use the typical graph-theoretic approaches used in the past (v. Bochmann and Sunshine 1983) because the complete graph representing the global FSM is not stored. Our procedure checks whether there is a run in which the protocol remains in one class and does not encounter a state belonging to *RS*. As shown in Theorem 1, the upper bound on the value of *max* is *R*, where *R* is number of all reachable states excluding members of *RS*.

Theorem 1. *If a protocol has a run with R state transitions which do not have a low probability transition or a member of RS, then it must have a livelock.*

Proof. Since the total number of reachable states excluding members of *RS* is *R*, at least one state must appear twice in this run. This means that the protocol has a loop which does not contain a member of *RS* or a low probability transition.

A protocol has functional correctness if it performs its expected services correctly provided that it makes progress (i.e. it does not encounter a deadlock or livelock). Functional correctness is analogous to the presence of safety properties (Hailpern and Owicki 1983). For checking functional correctness, we use a procedure similar to the one described in Sabnani and Lapone (1986). The services expected from the protocol are specified as a service FSM. Any undesirable behavior on the part of the component FSMs drives this service machine into an *error* state. Functional errors can be detected by treating the service FSM as a component FSM and checking whether the *error* state of the service FSM is reachable in the global FSM. An example service FSM shown in Fig. 10 (described in Sect. 5.3) models services for a simple data transfer protocol. If the protocol works correctly, then it should pick up a message from a user *tuser* and deliver it to another user *ruser*. If the protocol picks up two messages without delivering the first one, the protocol has a functional error. This error is detected because such behavior of the protocol leads this service FSM into its *error* state.

The protocol has a functional error if the behavior[2] of the component FSMs taken together is not a subset of the service FSM's proper behavior. The proper behavior of the service FSM corre-

[2] Recall from Sect. 2 that the *behavior* of an FSM is defined as the set of its traces (where a trace is defined as the sequence of labels of edges in a finite traversal of the graph representing the FSM)

sponds to all traversals of its underlying graph which do not cover its *error* state. For example, the proper behavior for the machine shown in Fig. 10 is (*transmitter!m;receiver?m*)*, where A^* means any number (including zero) of repetitions of A. If the composition of the component FSMs and of the service FSM contains a global state with its *error* state of the service FSM, then the protocol has a functional error.

If the global FSM has a global state with the *error* state of the service FSM as its component, then the composition of component FSMs has a trace which matches with a trace of the service FSM which goes through its *error* state. The sequence of events corresponding to such a trace is not permitted by the service FSM.

The termination conditions for our procedure are described next. Suppose that the set of *starting states* for Class j be given as S_j. If S_{j+1} is contained in the union of similar sets for all lower classes $\left(\bigcup_{k=1}^{j} S_k\right)$ and all states in Classes 1 to j have been exhausted, then all reachable states have been explored. Exhausting a class means that the successors of all states have been computed until the class is empty. The remaining states are either members of *RS* or of the next class.

Theorem 2. *If S_{j+1} is a subset of $\bigcup_{k=1}^{j} S_k$ and all states in Classes $1-j$ have been exhausted, then all reachable states have been explored.*

Proof. After all states in any class are exhausted, the remaining states are either members of *RS* or of next class. All states belonging to *RS* are the starting states for Class 1. Since all states in all classes including Class 1 are exhausted, the remaining states whose successors have not been computed must belong to Class $j+1$. Since S_{j+1} is contained in $\bigcup_{k=1}^{j} S_k$, the successors for all reachable states have been computed and these successors are other reachable states which have been already examined.

We store all starting states for the classes explored in a state table called *STATET*. Before expanding a new class (say $j+1$), we check whether its starting states (S_{j+1}) are already contained in this state table. If they are contained in this table, then we terminate the search. Compared to a traditional reachability computation (v. Bochmann and Sunshine 1983), the search through the state table is smaller because only a subset of states namely the starting states are stored in a state table. In a typical reachability computation, the search has to be done each time a new successor state is computed. This time is substantial if the number of reachable states is large such as one million (not an unusual number for large protocols). On the other hand, our procedure recomputes successors for some reachable states which are encountered again. Such repetitive work is not done in a traditional reachability computation.

4.1 Algorithm

The states of Class 1 are stored in an array $ST[1]$, the states of Class 2 in $ST[2]$, and so on. The key idea in the procedure given below is that we compute successor states for all states in the arrays: $ST[1]$, $ST[2]$, ..., $ST[i-1]$, where i is the upper bound on the number of classes to be explored. If there are α states left in $ST[i]$ at the end of computation, then the procedure given in Sect. 4.2 can be used to estimate the probability that a finite run of the protocol has an unexplored state. The value of i, the upper bound on the number of classes of states which will be examined, must be chosen by the protocol designer.

If a state is reached several times in a class (say k), we need to compute its successors only once. For doing it, we compress all instances of such a state into one entry of $ST[k]$. But we must store the number of times a state is reached because this count is essential for computing the probability of encountering an unexplored state in a run of arbitrary length. An entry henceforth called a *pair* is represented as $\{s, c\}$ where s is a global state of the form $[(s_1, s_2, \ldots, s_m), count]$ and c is the count described above.

A list *STATET* is used to store all starting states in the classes already explored. A list *STATETABLE* stores those global states belonging to *RS* which have been reached earlier in the analysis and whose successors have been computed. Note that *STATETABLE* is contained in *STATET*.

Step 1. We initialize the array $ST[1]$ to contain the initial global state and the other arrays to null.

$ST[1] := \{[(s_{1i}, s_{2i}, \ldots), 0], 1\}$;
$ST[2] := ST[3] := \ldots := ST[i-1] := ST[i] := ST[i+1] := null$;
$STATETABLE := null$;
$STATET := null$;
$k := 1$ where k is the current class of states being explored.

Step 2. If $ST[1]$ is empty, go to Step 4. If $ST[1]$ is non-empty, then set k to 1. Select a pair $\{g_s,$

$y\}$ from $ST[1]$. Let $g_s=[(s_1, s_2, \ldots), x]$. Remove this pair $\{g_s, y\}$ from $ST[1]$. If g_s is a member of *RS*, then add it to *STATETABLE* and to *STATET*.

Step 3. Let the set of outgoing edges from s_j in the FSM F_j be E_j. We will compute all the successors of g_s, which can be reached after performing one matching operation. It is done using the procedure described below.

For each edge $e_1 \in E_1$, do the steps described in this paragraph. Let the label of e_1 be $F_j?x$ or $F_j!x$. Check whether the set E_j for the FSM F_j contains an edge with the label $F_1!x$ or $F_1?x$. If it does, then it is a matching transition corresponding to $F_j?x$ or $F_j!x$. Let the edge e_1 connect state s_1 to t_1 and let edge e_j connect state s_j to t_j. This means that the global state $t_s=[(t_1, s_2, \ldots, t_j, \ldots), x+1]$ can be reached from the state g_s. If the global state t_s is a member of *RS* (or the transmitter can pick up a message in this global state), then add the pair $\{[(t_1, s_2, \ldots, t_i, \ldots), 0], 1\}$ to the array $ST[1]$ provided that it is not already a member of *STATETABLE*.

For adding a pair $\{g_s, y\}$ to an array $ST[k]$, we use the following rule. If no pair in $ST[k]$ contains the state g_s, then add the pair $\{g_s, y]$ to $ST[u]$. If $ST[u]$ contains such a pair $\{g_s, z\}$, then change this pair to $\{g_s, z+y\}$.

Let the probability of state transition from s_1 to t_1 in F_1 be denoted as $p_1=P(s_1, el_1, t_1)$ and that of the transition from s_j to t_j in F_j as $p_2=P(s_j, el_j, t_j)$, where el_1 and el_j are matching input/output operations $[(F_j?x$ and $F_j!x)$ or $(F_j!x$ and $F_j?x)]$. If both of these transitions are of high probability ($p_1 \geq (1-p)$ and $p_2 \geq (1-p)$), then the state t_s belongs to Class 1. Add this pair $\{[(t_1, s_2, \ldots, t_j, \ldots), x+1], y\}$ to $ST[1]$. If the count $x+l$ exceeds *max*, then the protocol has a livelock. If one of these transitions has a low probability ($p_1 \leq p$ and $p_2 \geq (1-p)$ *or* $p_1 \geq (1-p)$ and $p_2 \leq p$), then the state t_s belongs to Class 2. Add this pair $\{[(t_1, s_2, \ldots, t_j, \ldots), 0], y\}$ to $ST[2]$ using the rule given above. If both of these transitions have low probabilities ($p_1 \leq p$ and $p_2 \leq p$), then the state t_s is of Class 3. Add this state $\{[(t_1, s_2, \ldots, t_j, \ldots), 0]\}$ to $ST[3]$ using the rule given above.

Repeat Step 3 for each machine F_z, where $z=2$ to m. If the state g_s has no successors, then report it as a deadlock. Go to the beginning of Step 2.

Step 4. Increment k by 1. If the members of $ST[k]$ and similar arrays for higher classes are contained in *STATET*, then terminate the algorithm since all reachable states have been examined. Otherwise, add each member of array $ST[k]$ to *STATET* if it is not already contained in *STATET*.

Step 5. If k is equal to i, then terminate the computation. Recall that i is an upper bound on the classes which must be explored. Then, count the number of states in $ST[i]$ and use the analysis given in the next section to estimate the goodness of coverage.

Pick a pair $\{g_s, y\}$ from $ST[k]$. Remove this pair from $ST[k]$. Let $g_s=[(s_1, s_2, \ldots), x]$. Use the same procedure as described in Step 3 to compute the successor global states of g_s. Go back to Step 2.

4.2 Analysis

The algorithm given in the previous section always terminates provided that the number of reachable states is finite, as proved below.

Theorem 3. *Algorithm given in Sect.* 4.1 *will terminate eventually assuming that the number of reachable states is equal to RT if i, the upper bound on the number of classes to be explored, is equal to* ∞ *(infinity).*

Proof. If this algorithm does not terminate in Step 4, $ST[k]$ must contain at least one entry which is not already contained in *STATET*. Note that before expanding any state in Class k, the global states contained in $ST[k]$ constitute S_k. In Step 4, all elements of $ST[k]$ are added to *STATET*. So, the number of states contained in *STATET* will grow by at least one after expanding each class. Exhausting states in Class k means computing their successors until the remaining states are members of either the next class or *RS*. In the worst case, after expanding the states in Class *RT*, all the reachable states will be contained in *STATET* and the set of starting states for Class $RT+1$ will be contained in *STATET*, which contains all reachable starting states. Therefore, this algorithm must terminate after examining at most *RT* classes.

It can be easily concluded from this theorem's result that the number of classes which must be explored before the algorithm terminates (after exploring all reachable states) can be very large. The upper bound on the number of classes which must be explored is *RT*, the number of reachable states. This number can be as high as one million for complex protocols. It may be necessary to stop the computation before examining such a large number of classes. In such a case, there will be some states whose successors have not been computed. Then, an estimate for the global state cover-

age of the exploration already done is desirable. Such an estimate is chosen to be the probability of encountering an unexplored state in an arbitrarily long run of the protocol (for example, transport of n messages for a data transfer protocol).

Suppose that at the end of exploration, there are α states of Class i for which the successors have not been computed. This number α is computed by summing the counts for all pairs in $ST[i]$. Then, the probability of reaching one of these states is p_{worst} is equal to αp^{i-1} starting from a global state belonging to RS.

The storage of the protocol is defined as the maximum number of the packets that can be stored in all the component processes of the protocol, such as the transmitter, the communication channel, and the receiver. Suppose $n+s$ is the number of messages delivered to the transmitter and the storage of the protocol is s packets, then the probability of delivering n packets correctly is at least $(1-p_{worst})^n$ provided that the protocol has functional correctness. A data transfer protocol has functional correctness if it delivers messages to the receiver in correct sequence provided that the protocol does not have any deadlock or livelock. Whether a protocol is functionally correct can be checked by using a procedure described briefly earlier.

5 Example: the alternating bit protocol

We use a simple version of the Alternating Bit Protocol (ABP) as an example. The version used here uses half-duplex channel which can store at most one message or an acknowledgement. This protocol is specified as a collection of processes in Sect. 5.1. Its global state exploration trees are given in Sect. 5.2. A variation of this protocol which has a functional error is analyzed in Sect. 5.3.

5.1 Specification of the ABP

The ABP is specified as a collection of four processes, shown in Fig. 2. These processes *transmitter*, *receiver*, *comch*, and *timer* constitute the protocol. Two external processes *tuser* and *ruser* model the users of the protocol. As an abbreviation, we will use an edge labeled by $a*b$ to denote an edge labeled by an input/output operation a followed by an edge labeled by an input/output operation b. We will also use an edge labeled by $a+b$ to denote two parallel edges (edges connecting the same two states), one labeled by a and one labeled by b. We now give more details about the processes shown in Fig. 2.

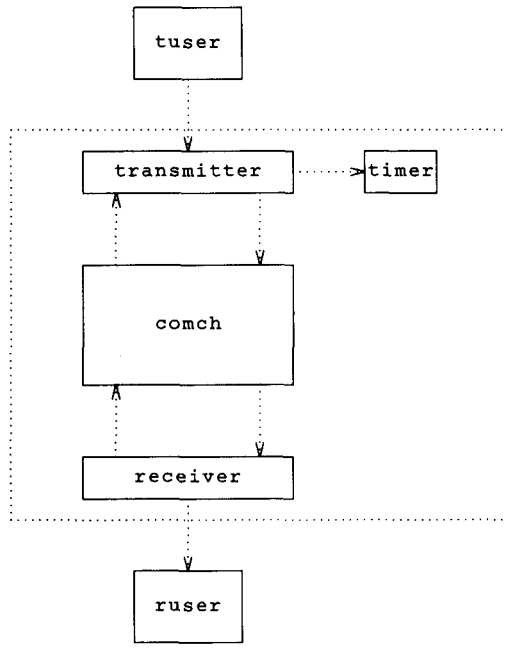


Fig. 2. Component processes of the Alternating Bit Protocol

5.1.1 The transmitter and timer. The *transmitter* receives messages from a user process *tuser* and assigns them sequence numbers *0* and *1*, alternately. It sends the message just received along with its assigned sequence number to the *receiver* via the communication medium *comch*. It then waits for an acknowledgement. If it does not receive an acknowledgement within a certain period of time called the time-out period, it retransmits the message. The *transmitter* has a slave process called *timer*. The *timer* indicates expiration of the time-out period.

The transmitter is specified by a finite-state machine given in Fig. 3. We denote by $m0$ a data message with the alternating bit set to 0 and by $m1$ the message with the alternating bit set to 1. Similarly, *ack* denotes the acknowledgement.

The *timer* is modeled as a FSM with two states-*0* (off) and *1* (on) (Fig. 4). It moves from its initial state 0 to state *1* when it receives a *start* signal from the *transmitter*. During its transition from state *1* to state *0*, it either sends an *out* signal to the *transmitter* or receives a *cancel* signal from the *transmitter*.

5.1.2 The receiver. Process *receiver* delivers the messages in the proper sequence to the user process *ruser* and sends the acknowledgements to the *transmitter*. The *receiver* is modeled as an FSM with 6 states (Fig. 5). If the *receiver* in its initial state *0* receives a message with sequence number *0* (*m0*), it delivers that message to *ruser* and sends an ack-

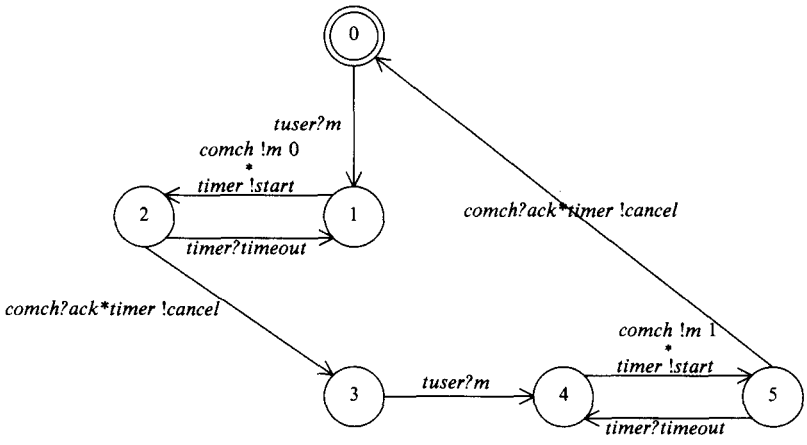


Fig. 3. The *transmitter*

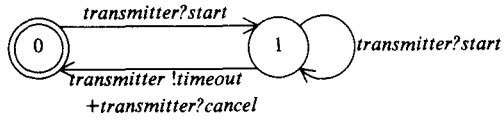


Fig. 4. The *timer*

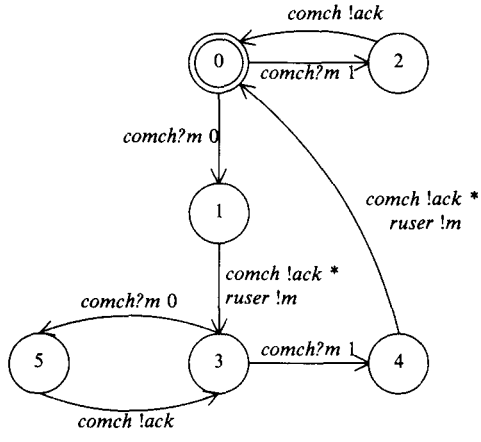


Fig. 5. The *receiver*

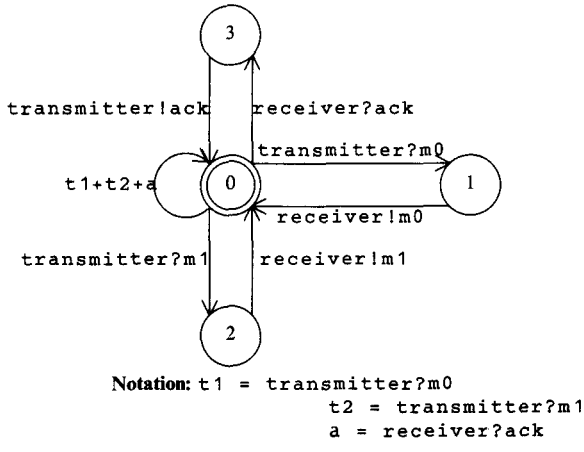


Fig. 6. The Communication Channel

nowledgement *ack* to the communication channel. On the other hand, if the *receiver* in the state *0* receives a message with sequence number *1*, it only sends an acknowledgement *ack* to *comch*. Similarly, if the *receiver* receives a message with sequence number *1* in the state *4*, then it delivers the message to *ruser* and sends an acknowledgement *ack*. But if the receiver receives a message with sequence number *0*, it only sends an acknowledgement *ack* to *comch*.

5.1.3 The Communication Medium comch. The communications medium is half-duplex and has one component: *comch*. It both transports data messages from the *transmitter* to the *receiver* as well as transports acknowledgements from the *receiver* to the *transmitter*. It is assumed that *comch* can lose messages with a probability of p, but does not corrupt messages that are delivered. This assumption can be ensured by using an error-detection code.

The message medium *comch* is specified as an FSM with four states (Fig. 6). In its initial state *0*, it receives one of the three message types: *m 0*, *m 1*, *ack*: from the *transmitter* or the *receiver*. *comch* either delivers a message correctly with a probability of $(1-p)$ or loses it with a probability equal to p.

We further assume that the contention for this medium is resolved fairly between the *transmitter* and the *receiver*. If the medium *comch* picks up a message from the *transmitter*, it will next pick up an acknowledgement from the *receiver* (provided that the *receiver* has an acknowledgement to send); and viceversa.

5.2 State exploration

In this section, we present the results of state exploration for the protocol just described. We only ex-

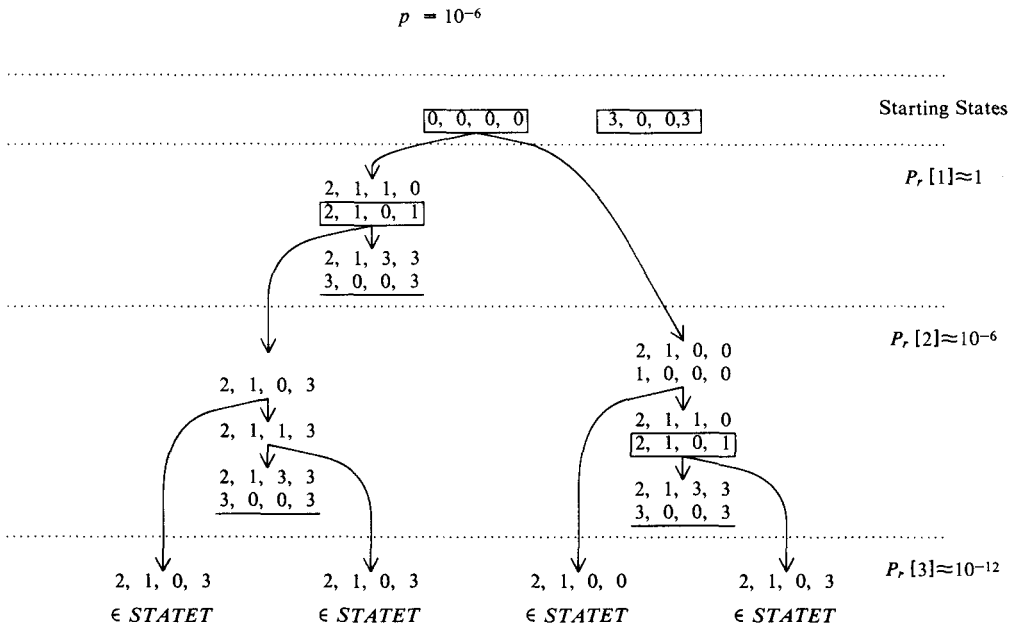


Fig. 7. Tree Diagram for the ABP

plore the states of its global FSM. No attempt is made to locate errors in this protocol.

To simplify the state exploration, we assume that the *roundtrip* delay from the *transmitter* to the *receiver* is smaller than the time-out period. Later, we will summarize the results for this protocol without making this simplifying assumption.

We also assume that the value of p is equal to 10^{-6}. It is p's value for the following realistic example. Bit errors are randomly distributed, the bit error rate is 10^{-9}, and the message length is always a constant equal to 1000 bits. The probability of an error in the message delivery is $1-(1-10^{-9})^{1000}$ or 10^{-6}. This value is also an upper bound for the probability of acknowledgement loss because the length of an acknowledgement will be typically much smaller than 1000 bits.

This protocol's global state is given as $[(s_1, s_2, s_3, s_4), count]$ in which s_1 is the *transmitter*'s state, s_2 the *timer*'s state, s_3 *comch*'s state, s_4 the *receiver*'s state, and *count* is the number of state transitions which took the protocol to its present state. Since we are conducting state exploration (not verifying it), only the four-tuple (s_1, s_2, s_3, s_4), will be used to represent any global state in the rest of this section.

The set *RS* for this protocol is the set of those global states in which the protocol can pick up a message from the user *tuser*. It is formally given as $\{(s_1, s_2, s_3, s_4)$: $s_1=0$ *or* 3; $s_2=0$ *or* 1; $0\leq s_3\leq 3$; $0\leq s_4\leq 5\}$.

Figure 7 shows the tree expansion from the initial global state (*0, 0, 0, 0*). The leaves of this tree are the Class 4 states from the array $ST[4]$ or a global state (*3, 0, 0, 3*) in which the *transmitter* can pick up a message from the user process *tuser*. We can build a similar tree starting from the global state (3, 0, 0, 3). The execution of the algorithm given in Sect. 4 is shown in Fig. 8.

Starting from the initial state (*0, 0, 0, 0*), the protocol can get into two states: (*2, 1, 1, 0*) and (*2, 1, 0, 0*). The first one is a Class 1 state and is stored in $ST[1]$. We first compute the successors of (*2, 1, 1, 0*). It has one successor: a Class 1 state (*2, 1, 0, 1*). This Class 1 state has two successors: one Class 1 state (*3, 0, 0, 3*), and one Class 2 state (*2, 1, 0, 3*). In the Class 1 state (*3, 0, 0, 3*), the *transmitter* can pick up a message. We build another tree starting from this state similar to the one shown in Fig. 7. In the tree shown in Fig. 7, Class 2 has two starting states: (2, 1, 0, 0) and (2, 1, 0, 3). They are added to *STATET*. At this time, *STATET*=$\{$(2, 1, 0, 0), (2, 1, 0, 3)$\}$. After exhausting states in Class 2, we are left with one Class 1 state (*3, 0, 0, 3*), a member of RS, and two Class 3 states: (2, 1, 0, 0) (three occurrences) and (2, 1, 0, 3) (one occurrence). Both of these starting states for Class 3 are already contained in *STATET*. Therefore, all reachable states have been explored.

We have not compressed several instances of one state in a single entry called *pair* in Fig. 8. Our choice of not compressing several instances of a state into one pair simplifies the explanation of the algorithm given earlier in Sect. 4.1.

We analyzed the same protocol without making the roundtrip propagation delay assumption.

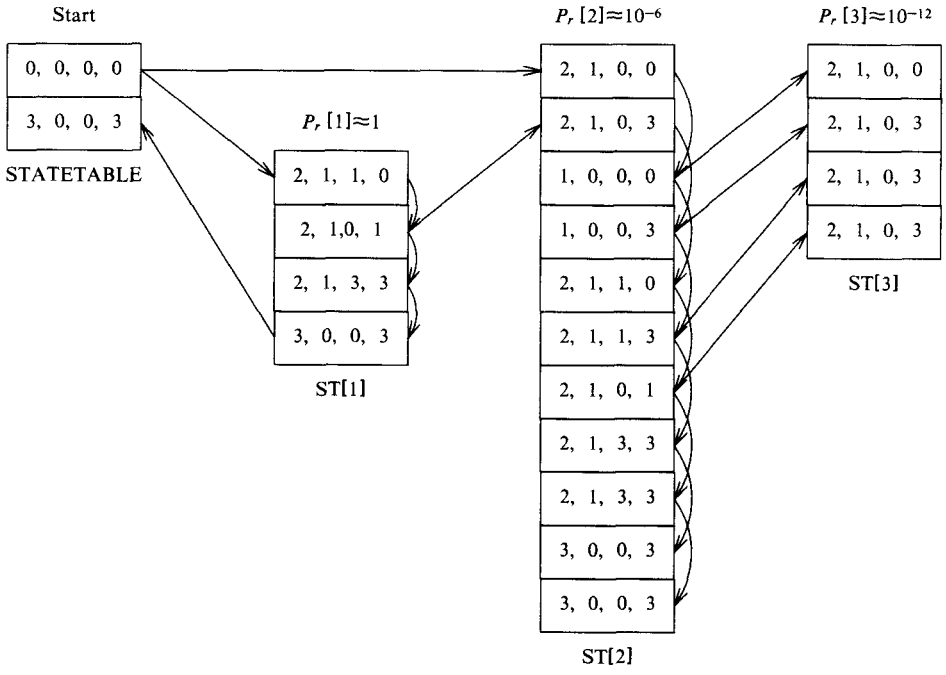


Fig. 8. *Tree Search*

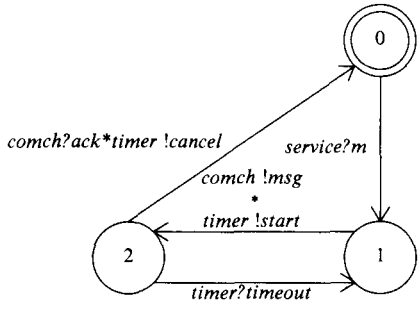


Fig. 9. The altered *transmitter*

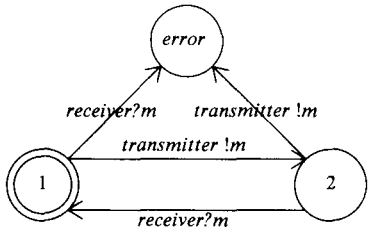


Fig. 10. The Service FSM

All of its reachable states are explored after examining Class 1–3 states.

5.3 *Another example*

The protocol described in the previous section is changed so that the messages from the transmitter to the receiver do not carry sequence numbers. Each message from the transmitter to the receiver is denoted as *msg*. The altered *transmitter* is shown in Fig. 9. We will show that this protocol has a functional error.

The service FSM for this protocol shown in Fig. 10 delivers a message to the transmitter. Then, the receiver must deliver the same message to the service FSM before the transmitter picks up the next message. If the transmitter picks up two messages before the receiver delivers any one of them, then the protocol has a functional error.

The state exploration of this protocol using algorithm given in Sect. 4.1 shows that several reachable global states contain the *error* state of the service FSM as their component. This shows that this protocol does not provide the expected service.

6 Conclusions

We believe that the work presented here will have the following impact on the protocol research:

- The protocol verification can be simplified because it checks the most probable portions of the protocol.
- The protocol designs can be simplified because a designer may decide to ignore errors which may happen once in a long time, such as five years.
- Multilayered protocols can be verified together. Redundant portions (the portions that take care

of errors in different layers) can be identified and removed.

The technique described here has been implemented in a computer program. It has been used to verify an ISDN protocol *Q.931* and the ISO Transport Class 4 Protocol. An open research problem for the future is developing partial verification procedures for other frameworks, such as programming languages and temporal logic. A problem for future work is generalizing the procedure given here for communication channels with dependent errors.

References

Bochmann G v (1975) Logical verification and implementation of protocols. Proc 4th Data Communications Symposium, Quebec, Canada, pp 8.5–8.20

Bochmann G v, Sunshine CA (1983) A survey of formal methods. In: Green PE (ed), Computer networks and protocols (May 1983) Plenum Press, New York, pp 561–578

Danthine AAS (1983) Protocol Representation with Finite-State Machines. In: Green PE (ed), Computer networks and protocols (May 1983) Plenum Press, New York, pp 579–606

Gouda MG (1978) Protocol machines: towards a logical theory of communication protocols. Doctoral thesis, University of Waterloo, Ontario, Canada

Gouda MG (1984) Closed covers: to verify progress of communicating finite state machines. IEEE Trans on Software Engineering (November 1984) SE-10:846–855

Hailpern B, Owicki S (1983) Modular verification of communication protocols. IEEE Trans Commun 31:56–68

Hoare CAR (1978) Communicating sequential processes. Commun ACM 21:666–677

Jelinek F (1969) Fast sequential decoding algorithm using a stack. IBM J Res Dev 13:675–685

Lam SL, Udaya Shankar A (1984) Protocol verification via projections. IEEE Trans Software Eng (July 1984) SE-10:325–342

Lapone A, Sabnani K, Umit Uyar M (1988) An algorithmic procedure for checking safety properties of protocols. (to appear in IEEE Trans Commun)

McNaughton R (1966) Testing and generating infinite sequences by a finite automation. Inf control 9:521–530

Maxemchuk NF, Sabnani K (1987) Probabilistic verification of communication protocols. In: Rudin H, West C (eds), Protocol specification, testing, and verification, vol VII. Elsevier Science Publishers, North-Holland, pp 307–319

Peterson WW, Weldon EJ Jr. (1972) Error correcting Codes, 2nd edn. MIT Press, Cambridge, Mass

Pnueli A, Zuck L (1986) Verification of multiprocesses probabilistic protocols. Distrib Comput 1:53–72

Rudin H (1984) An improved algorithm for estimating protocol performance. In: Yemini Y, Strom R, Yemini S (eds) Protocol specification, testing, and verification, vol IV. Elsevier Science Publishers, North-Holland, pp 515–525

Rudin H, West CH (1982) A validation technique for tightly coupled protocols. IEEE Trans Comput 31:630–636

Sabnani K (1988) An algorithmic technique for protocol verification. IEEE Trans Commun 36:924–931

Sabnani K, Lapone A (1986) PAV – protocol analyzer and verifier. In: Sarikaya B, Bochmann G v (eds) Protocol specification, testing, and verification, vol VI. Elsevier Science Publishers, North-Holland, pp 29–34

Sabnani K, Schwartz M (1984) Verification of a multidestination selective repeat protocol. Comput Networks 8:463–478

West C (1986) Protocol validation by random state exploration. In: Sarikaya B, Bochmann G v (eds) Protocol specification, testing, and verification, vol VI. Elsevier Science Publishers, North-Holland, pp 233–242

Zafiropulo P (1983) Protocol analysis and synthesis using a state transition model. In: Green PE (ed) Computer network and protocols (May 1983) Plenum Press, New York, pp 645–670