A DISCRETE EVENT SIMULATION APPROACH TO PROTOCOL VALIDATION

Theodore Ono-Tesfaye and Pawel Gburzynski
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
E-mail: {theodore,pawel}@cs.ualberta.ca

KEYWORDS

Discrete event simulation, state exploration, protocol verification, protocol specification.

ABSTRACT

We present a system, descending from an event-driven simulator, for verifying probabilistic and timed properties of communication protocols. Our system was inspired by SPIN [Holzmann, 1991, Holzmann, 1992, Holzmann, 1997], and employs essentially the same efficient technique of state exploration, but it extends those ideas in the direction of expressing properties involving probabilities and strict timing (as opposed to the mere succession of events). This way, our approach is applicable to protocols in which the exact timing of events is critical for their correct operation, e.g., protocols for real-time multimedia applications, access protocols for shared-media networks, or reactive protocols driving distributed real-time equipment.

INTRODUCTION

Designers of communication protocols need to ensure that (i) the protocol performs well, and (ii) that it is free from errors. Discrete event simulation (DES) has become an indispensable and widely used tool for the performance evaluation of communication protocols. In contrast, and despite some recent progress [Daws and Yovine, 1995, Miller and Xue, 1996, Holzmann, 1997, Yuan et al., 1997], the use of automated tools for the validation of protocols is much less common. There are several reasons why the use of validation tools has lagged behind the use of performance evaluation tools like DES:

- Most validation tools are lacking in the level of realism they support. To realistically capture the properties of a protocol, it is necessary to be able to specify its timing aspects (e.g., delays between events) and its probabilistic aspects (e.g., probability of loss). Note that the most widely known validation tool, SPIN [Holzmann, 1992, Holzmann, 1997], supports neither time nor probabilities.

- Unlike most network simulation tools (OPNET, SMURPH [Gburzynski, 1996]) that are typically based on common programming languages, like C or C++, validation tools often employ special-purpose languages (e.g., PROMELA in SPIN, KRONOS [Daws and Yovine, 1995]). The use of these languages is rooted in the desire to achieve a level of mathematical precision not obtainable by using C or C++.

- Protocol validation is computationally difficult due to the *state space explosion problem*—the fact that the state space of a protocol is exponential in the number of variables and processes. Validation algorithms published in the literature often have unrealistic resource requirements. For example, the probabilistic validation algorithm proposed in [Baier et al., 1998] requires the manipulation of a Markov chain representing the entire state space of the protocol.

In this paper, we propose to address these problems by extending the DES paradigm to perform the functions of a protocol validator. We have implemented a prototype validation tool as a C++ library. This approach has several advantages: protocols can be modelled realistically, including all timing and probabilistic aspects; a single model can be used both for the performance evaluation via simulation and the validation of protocols; and the use of C++ eliminates the learning curve associated with special-purpose languages.

Of course, discrete event simulators have been used for validation (debugging) of protocols in the past, but this has generally happened in an ad-hoc fashion by running a large number of simulations with different random seeds, initial states, etc. Our approach is more systematic and combines the DES paradigm with methods already used by validation tools. The task is simplified by the inherent similarity between DES and protocol validation: in both cases, it is necessary to first specify the protocol (typically as a set of processes) and then execute it in some fashion. Throughout our work, the emphasis is on practical utility rather than on mathematical precision.

The rest of the paper is organized as follows. Section **MODEL** describes our protocol specification method and a logic for specifying properties of protocols. Section **ALGORITHMS** describes our model-checking algorithm and discusses some strategies for tackling the state explosion problem. Some results obtained with our validation tool are reported in section **EXPERIMENTS**. Finally, we summarize our conclusions.

MODEL

Protocol Models

In this section, we informally define two protocol modelling concepts: a low-level model and a high-level model. The low-level model is mathematically simple but inconvenient for the description of complex systems. Our logic and our algorithms operate at this level. The high-level model allows the modular specification of communication systems as sets of communicating processes. The semantics of the high-level model are given in terms of the low-level model.

Low-level Model.
Our low-level model is a *timed probabilistic transition system (TPTS)*, which is a discrete-time Markov chain with additional labels e (event type) and t (delay) on each transition. The delay of a transition indicates the number of time units that the system spends in the origin state before making the (instantaneous) transition to the destination state. A TPTS can be depicted as a directed graph whose nodes are states and whose edges are the transitions with non-zero probabilities. These transitions are written as $g \xrightarrow{e,t,p} g'$. Figure 1 shows an example TPTS with 13 states. It represents a system where an activity (event ACT) results either in the production of data (event PRODUCE) or in the consumption of data (event CONSUME) after 5 time units. When data is produced, it is put in the buffer (PUT) with a delay of 1 time unit, and when it is consumed, it is extracted from the buffer (GET) after 1 time unit. The buffer has a capacity of 1 and an attempt to PUT into a full buffer is immediately followed by a FULL event. Similarly, an attempt to retrieve data from an empty buffer is followed by an EMPTY event.

High-level Model.
Our high-level model is that of a *probabilistic protocol* which consists of a set of communicating processes. Processes are finite state machines that respond to events. When a process is in state s and it receives an event, it changes to some state s' and generates a set A of new timed events. The probability P of generating a particular set of output events depends on the state s and the type of the received event. We write $s \overset{y/A/P}{\Longrightarrow} s'$.

The semantics of a protocol are given in terms of a TPTS whose states consist of the states of all processes and a list of pending events—the *event list*. The protocol works by repeatedly *executing* the earliest event in the list. In the initial global state, all processes are in their initial states, and the event list consists of the initial events scheduled at time 0. When an event is executed, its time is subtracted from the times of the remaining events and the event's destination process state changes according to the process transition relation. New events are generated by choosing one set of new events $A_i (i = 1, \ldots, L)$, according to the probability distribution of the transition, and appending it to the event list. Thus, the times of output events denote the delay between the current time and the new event, not the absolute event times.

To illustrate these concepts, we study a probabilistic protocol consisting of two processes, a simple data buffer with capacity 1 and a producer/consumer that utilizes the buffer. The buffer is modelled by the process in figure 2 (left). It has two states – 0 (empty) and 1 (full) and receives PUT and GET events that force transitions between these states. If the buffer process receives a PUT event while it is in the full state, it generates a FULL event to itself.[1] Conversely, if it receives a

[1] Exposing error conditions as events in this way makes it possible to specify EL formulas (see section **MODEL**) to reason about them.

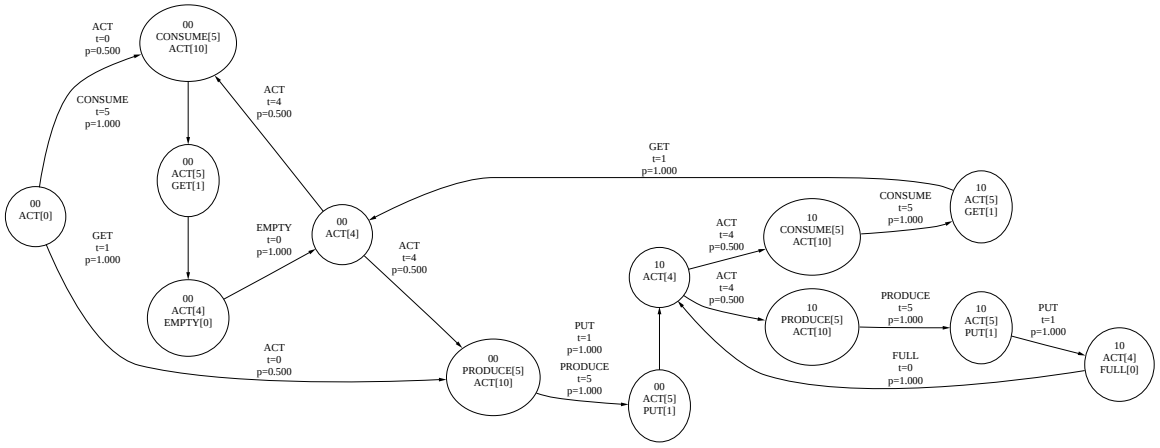

Figure 1: Producer/Consumer System

GET event while in the empty state, it generates an event of type EMPTY.

The producer/consumer is modelled by the process in figure 2 (right). When it receives an activity event ACT, it generates the next activity event with a delay of 10 time units and either a PRODUCE or CONSUME event to itself with a delay of 5 time units. PRODUCE or CONSUME are each chosen with probability 0.5. When the producer-consumer receives a PRODUCE event, it sends a PUT event with delay 1 to the buffer process; when it receives a CONSUME event, it sends a GET event to the buffer process. Note that the producer/consumer process has only one state. The 13-state TPTS corresponding to this protocol is shown in figure 1.

Because a process can generate more than one event in response to a single event, it is possible for a probabilistic protocol to result in an infinite TPTS. Also, processes can generate events with zero delay, and it is therefore possible for the resulting TPTS to have an infinite loop in which time never progresses—it *stutters*. In the rest of this paper, we will assume that protocols are well-behaved, in the sense that their TPTS representation is finite, stutter-free and has no sink states (i.e., states with no successors).

Event Logic

We consider events and their delays to be the only externally observable results of the execution of a TPTS. It is therefore useful to define the notion of *event sequence*: a list of events and delays $e = e_0 \xrightarrow{t_1} e_1 \xrightarrow{t_2} e_2 \cdots \xrightarrow{t_n} e_n$ is an event sequence starting at g_0 if there is a path $g_0 \overset{e_0,t_0,p_0}{\longrightarrow} g_1 \overset{e_1,t_1,p_1}{\longrightarrow} \ldots \overset{e_n,t_n,p_n}{\longrightarrow} g_n$ in the TPTS. It is straightforward to define a probability measure $\mathcal{P}$ on finite and infinite paths of a TPTS starting at some initial state g_0, see for example [Baier et al., 1998].

We present a simple temporal logic—event logic (EL)—for expressing properties of a TPTS. EL is based on the linear-time logic implemented by SPIN and borrows probabilistic real-time extensions similar to those used by the logic PCTL [Hansson and Jonsson, 1989]. PCTL itself is an extension of the branching time logic CTL. EL can express properties such as "event A is followed by event B within 100 time units with probability ≥ 0.50". The grammar for probabilistic EL formulas is as follows:

- the constants TRUE and FALSE and the event types are atomic EL formulas
- if f_1, f_2 are EL formulas, then $\neg f_1, f_1 \wedge f_2$, $f_1 \vee f_2$, and $f_1 U^{\leq t} f_2$ for $t \in N$ are EL formulas
- if f is an EL formula, then $P(f) \geq p$ for $p \in [0,1]$ is a probabilistic EL formula

The meaning of a formula $f_1 U^{\leq t} f_2$ is that f_2 becomes true within t time units and that f_1 will be true until f_2 becomes true. Satisfaction of non-probabilistic EL formulas is defined on events that are part of infinite event sequences. An infinite event sequence satisfies a formula if the formula is satisfied by all events of the sequence. For an event type e_i, an event sequence $e = e_0 \xrightarrow{t_1} e_1 \cdots e_i \cdots \xrightarrow{t_{i+1}} e_{i+1}$ and a non-probabilistic formula f we define

- if f is atomic, then $e_i \models f$ if $f =$ TRUE or $f = e_i$, otherwise $e_i \not\models_e f$

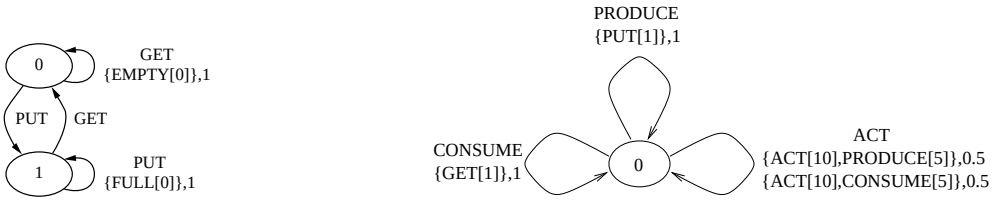

Figure 2: Buffer (left) and Producer/Consumer Process (right)

- if $f = f_1 \vee (\wedge) f_2$, then $e_i \models_e f$ if $e_i \models_e f_1$ or (and) $e_i \models_e f_2$
- if $f = \neg f_1$, then $e_i \models_e f$ if $e_i \not\models_e f_1$
- if $f = f_1 U^{\leq t} f_2$, then $e_i \models_e f$ if $e_i \models_e f_2$ or ($t_{i+1} \leq t$ and $e_{i+1} \models_e f_1 U^{\leq t - t_{i+1}} f_2$)
- $e \models f$ if for all $i \in \{1, 2, \ldots\}$: $e_i \models_e f$.

A probabilistic EL formula $P(f) \geq p$ is satisfied by a TPTS M if the measure of satisfying event sequences of M $\mathcal{P}(\{e : e \models f\})$ is greater than or equal to p. An algorithm that checks this is described in section .

ALGORITHMS

Basic Algorithm

Our basic algorithm is a depth-first search of the protocol state space (the TPTS). It is similar to the approach described by Holzmann [Holzmann, 1991, Holzmann, 1997] in that the state space is constructed on-the fly and that a hash table of bits is used to detect previously visited states. This basic algorithm can be used in cases when it is not necessary to check the validity of an EL formula.

A state search algorithm needs to detect when it encounters a state that has already been visited. When the state space is too large to be stored in memory, Holzmann's bitstate hashing algorithm can be used. The algorithm uses a large hash table of bits to mark visited states: whenever a state is visited, a hash value of the state is calculated. This value is used as an index to the hash table, and the bit at the indexed location is set to 1. Thus, the validation algorithm can detect previously visited states by checking if the corresponding bit in the hash table has been set. This algorithm greatly reduces the memory requirements of the state space search, but there is a possibility that two different states hash to the same value. If the load factor of the hash-table is low, i.e., if the table is much larger than the number of states, the probability of missing a state is very low. Holzmann's algorithm uses two independent hash functions and two hash tables to further reduce the probability that two different states map to the same hash values. The memory requirement of this algorithm is 2 bits per hash table entry.

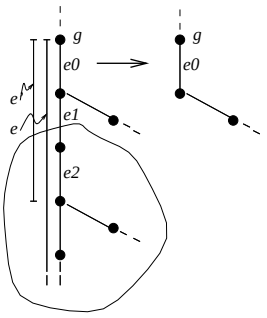

Figure 3: Example of Inner Loop

Model-Checking of Event Logic Formulas

The process of checking whether a TPTS satisfies an EL formula is called *model-checking*. In this section, we present an algorithm for calculating the measure of the event sequences of a TPTS M that satisfy an EL formula f. The basic approach is to search the tree of all paths of M and to remove paths that do *not* satisfy f. The measure of the removed paths is added to some global variable NS which is the output of the algorithm when it terminates. Like the model-checking algorithm in SPIN, we employ a nested depth-first search (DFS): an outer DFS and an inner DFS which operate as follows.

Inner DFS. From every state g that is visited by the outer DFS, the inner DFS determines for each infinite event sequence e starting at g, whether *the first event* e_0 of e satisfies the (non-probabilistic) formula f or not. Note that since EL formulas are finite and since every "until" operator U has a finite time horizon, we can determine if e_0 does not satisfy f by examining finite subsequences e' of e. If e_0 does not satisfy f in the context of e', e is removed from the search tree (figure 3). The inner DFS procedure returns the measure of the paths that were removed in this fashion.

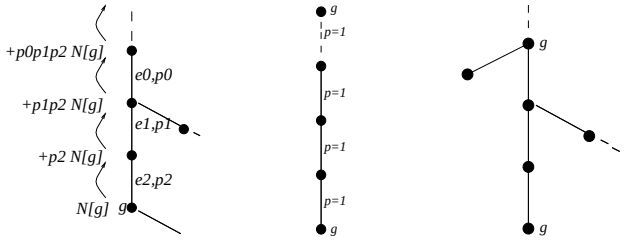

Figure 4: Propagation of N-values (left), Satisfying Cycle (middle), 0/1-Cycle (right)

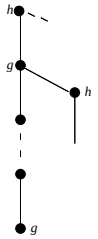

Figure 5: Outer Loop Scenario

Outer DFS. We now describe the outer DFS. This part of the algorithm must ensure that every non-satisfying path in the tree is measured exactly once. For every state g in the search tree, we define the $N(g)$ as the measure of infinite paths starting at g that do not satisfy f. The task, therefore, is to compute $N(g_0)$.

$N(g)$ is a real value $\in [0,1]$ and storing $N(g)$ for all states is too expensive. Because of this, the algorithm stores only limited information about what is known about $N(g)$ for a state g. This information can have one of four values:

- 0: $N(g)$ is 0, i.e., all path starting at g satisfy f
- 1: $N(g)$ is 1, i.e., none of the paths starting at g satisfies f
- 0/1: either all paths starting at g satisfy f or none of the paths starting at g satisfy f
- UNKNOWN: it is not known whether any of the three cases above apply or not — this is the default situation for unexamined states

Storing this information requires 2 bits per visited state, we will refer to it as $I(g)$. In addition to this limited information about the $N(g)$ of all states, the algorithm also stores the exact $N(g)$ of the nodes on the current path from the root of the search tree. $N(g)$ is initialized to 0 when g is first visited by the outer DFS.

The inner DFS procedure is started at each node that the outer DFS visits. The value that the inner DFS procedure returns — the measure of paths that were removed by it — is propagated backwards along the current path to update $N(g)$ for the nodes on the path (fig. 4 left). A search path in the outer DFS is aborted if

1. the measure of non-satisfying paths starting at the current state g is known, i.e., $I(g) = 0$ or $I(g) = 1$ (the first two cases above), or
2. the current state g has already been visited on the current path from the root.

In case 1, the value is propagated upwards (fig. 4 left) just as if it had been computed again. Case 2 implies that a cycle has been detected. We must consider two subcases: (i) all transitions between the two identical states have probability 1 (fig. 4 middle), and (ii) there is at least one transition between the two states whose probability is not equal to 1 (fig. 4 right). Scenario (i) means that $N(g)$ is 0, since no non-satisfying path can be reached from g, and $I(g)$ is set to 0. In scenario (ii), $N(g)$ can only be 0 (if no non-satisfying path can be reached from g) or 1 (if at least one non-satisfying path can be reached from g, this means that each infinite path starting at g is non-satisfying with probability 1). $I(g)$ is thus set to 0/1.

If $I(g)$ is 0/1 and $N(g) > 0$ when the outer DFS has finished searching the subtree rooted in a state g, we can conclude that the final value of $N(g)$ must be 1. The difference $1 - N(g)$ must be propagated backwards along the path. It is important to note that $I(g)$ =0/1 and $N(g) = 0$ on the other hand does not imply that the final value of $N(g) = 0$. Figure 5 shows a situation where $N(g) = 0$, $I(g)$ =0/1 after the subtree rooted in g has seen searched, but the final value of $N(g)$ is not necessarily 0. This value can be either 0 if no no-satisfying path can be reached from state h, or 1 if at least one non-satisfying path can be reached from h. Note that the algorithm does not "miss" any non-satisfying paths — they will be detected during the search of the subtree rooted in h, and $N(h)$ will be correctly set to 1.

The algorithm stops when the entire tree rooted in g_0 has been searched. $N(g_0)$ is then the measure of paths that do not satisfy f, and conversely, $1 - N(g_0)$ is the probability that f is satisfied.

```
// Algorithm checkEL (outer DFS)
// global constants: TPTS M, formula f
// call as: outer_loop(g_0, g_0);
// N[g_0] will contain the measure of
// paths that do not satisfy f.
outer_loop(g, path) {
    if I(g) = 0 or I(g) = 1 then
        propagate I(g);
        return;
    else if g visited earlier in path
    then
        if all probabilities = 1
        then I(g) := 0;
        else I(g) := 0/1;
        return;
    N[g] := inner_loop(g);
    propagate N[g];
    for each state g' in M with
    p' = P(g, g') > 0
        outer_loop(g', path --p'--> g')
    if I(g) = 0/1 and N(g) > 0
    then
        propagate 1 - N[g];
        I(g) := 1;
        return;
}
```

Figure 6: Algorithm for Checking EL Formulas, Outer Loop

```
// Algorithm checkEL (inner loop)
// global constants: TPTS M, formula f
// returns the measure of all
// event sequences starting at g
// whose first event does no satisfy f.
inner_loop(g) {
    n := 0;
    for all event sequences e =
    e_0 e_1 e_2 ... starting at g
        if e_0 ⊭_e f
            n+ = P(e);
            remove e from
            the last branch-
            ing point on;
    return n;
}
```

Figure 7: Algorithm for Checking EL Formulas, Inner Loop

Probabilistic Search

When the size of the state space exceeds the limit of what can be handled within the available time and memory, it needs to be restricted in some way. When using the bitstate hashing method, the state space is naturally limited by the size of the hash table—the state graph is randomly truncated. Other ways to limit the state graph is to set a maximum depth threshold or a minimum probability threshold. When using a probability threshold, path probabilities are used as heuristics to guide the state space search towards more probable regions of the state graph. The principle is simple: we set a small probability threshold (e.g., 10^{-10}) and truncate a search path when the current path probability becomes smaller than the threshold.

The basic bitstate hashing method does not work for a probabilistic search. When the validation algorithm searches the event graph, it keeps track of the probability of the path from the root until the current node; we call this probability the *current probability*. A state can be be visited several times on event paths with different current probabilities. For instance, assume that a state is first visited with a probability p, and then with probability q. Then, the second path through the state should not be aborted if $q > p$, because the second path may extend deeper into the search graph before reaching the probability threshold.

One possible solution for this is to make the probability part of the state whose hash value is calculated. This would result in states always being classified as different if they are on event paths with different probabilities. In our model, we use a different solution: instead of using just one bit in the hash table to indicate that a state has been visited, we use n bits. The n bits indicate the largest current probability for a state that hashes to that hash table entry. Our implementation uses $n = 8$ and stores the order of magnitude of the probability, without the minus sign, e.g., $p = 1.3 \times 10^{-22}$ becomes 22. Probabilities $< 10^{-255}$ are stored as 255. A search path is aborted if the validator finds that the same state has already been visited with a higher current probability.

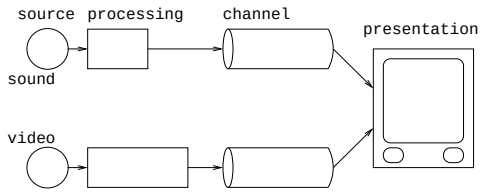

Figure 8: Multimedia Stream Model

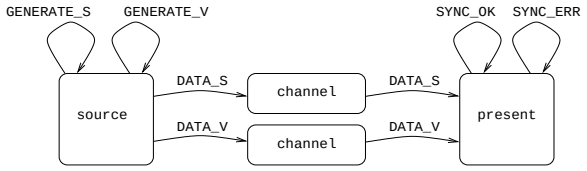

Figure 9: Processes for Multimedia Stream Model

EXPERIMENTS: MULTIMEDIA SYSTEM

We investigate a simple multimedia transmission/presentation system (figure 8). A generic multimedia system consists on n streams of periodic data, each of which may have different periods, processing delays and transmission channels. These streams are merged at the presentation device. In our model, we have $n = 2$ streams: sound and video data. The periods of the two streams are p_s and p_v, respectively. If g is the greatest common divisor (gcd) of the two periods, then the joint period of the two streams is $(p_s p_v)/g$, because $(p_s p_v)/g$ is divisible by both p_s and p_v. In order to check the synchronization of the two streams, they are tagged with sequence numbers as follows: the stream with the shorter period—sound in this case—is the reference stream. Its packets are tagged with numbers $0, 1, \ldots$ up to the end of the joint period, and each time period of length p_s is associated with the tag of the packet sent at the beginning of the period. The packets of the stream with the longer period are tagged with the tags associated with the periods that are within $p_s/2$ of the packet time. Figure 10 illustrates this for the case of $p_s = 12$ and $p_v = 40$. The joint period is then 120.

Using the sequence numbers, synchronization errors of the video packets relative to the reference sound stream can be detected at the presentation

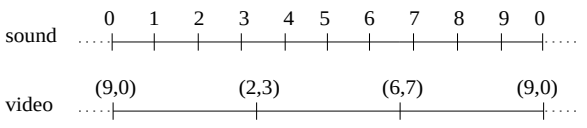

Figure 10: Packet Tags for Two Streams

Sound Del	Video Del	Channel Del	No. States	f
10	10	60	55	TRUE
10	10	[50,60]	1079	TRUE
10	[8,12]	[50,60]	32467	FALSE
10	[7,13]	[50,60]	84838	FALSE

Table 1: State Space and Synchronization, without Synchronizer

device without using timers. A video packet is considered to be out of sync if none of its tags correspond to the tag of the sound packet preceding it.

The implementation of this example is straightforward and uses three process types: the source, the presentation device and a channel process. Figure 9 shows the processes and the events exchanged among them. The presentation device process generates SYNC_OK and SYNC_ERR events to itself in response to video packet receptions. Because EL formulas are expressed in terms of event types, exposing synchronization errors as events makes it possible to write EL specifications about them.

First, we do a state space search for different delay values to check for non-temporal errors like deadlocks and unspecified receptions. No EL formula is needed for this test. Given the simplicity of the system, it is not surprising that no such errors are detected. Table 1 summarizes the results for different delay values. Next, we check for synchronization errors. We want to specify that all video packets are in sync and arrive within some bounded delay, i.e., they result in the generation of a SYNC_OK event within some time limit t. In EL, this is expressed the formula

$$f := \text{GEN\_V} \;\rightarrow\; \text{TRUE}\; U^{\leq t}\; \text{SYNC\_OK}$$

where t is set to be greater than the sum of the maximal processing and channel delays. The results for this test are in the last column of table 1.

In addition to the basic system, we consider a multimedia system with a simple synchronization device (fig. 11). The synchronizer works by buffering packets of both streams until both buffers are at least half full. Once both buffers have exceeded this threshold, it retrieves packets with the same period as the sources and continues buffering any incoming packets. The buffers are dimensioned to be twice the size of the joint period of the two streams (=

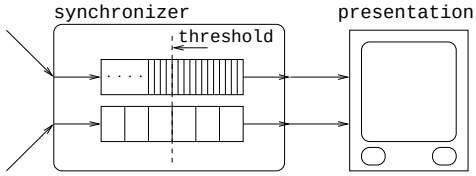

Figure 11: Synchronizer

Sound Del	Video Del	Channel Del	No. States	f
10	10	60	124	TRUE
10	10	[50,60]	206852	TRUE
10	[8,12]	[50,60]	1139013	TRUE
10	[7,13]	[50,60]	1104382	TRUE

Table 2: State Space and Synchronization, with Synchronizer

$2 \times 120 = 240$ in this case), so the buffer sizes for the sound and video streams are 20 and 6, respectively.

Running the same tests as above, we find that the EL formula is satisfied in all cases. In fact, the synchronization buffer prevents synchronization errors completely, so the system satisfies the simpler EL formula $\neg$ SYNC_ERR. Note that the addition of the synchronizer results in a considerably larger state space. Due to memory constraints, the fourth experiment (final row in table 2) could not be completed as an exhaustive search—we performed a probabilistic search with threshold 10^{-15} instead.

CONCLUSIONS

We have presented a protocol validation technique as an extension to discrete event simulation (DES). Compared with other methods of protocol validation, ours takes a more practical approach and allows protocol designers to specify protocols using the familiar DES paradigm. Two methods for dealing with the state explosion problem—an adaptation of Holzmann's state hashing and probabilistic search—were discussed. To demonstrate the validity of our approach, we described some experiments using a prototype implementation of our ideas. Similar methods can be applied to other standard DES tools.

Due to the state space explosion problem, we do not expect to be able to exhaustively search the state spaces of industrial-sized protocols, and thus, our tool is not suitable for rigorous correctness proofs. However, we believe that the methods introduced here promise to enhance the usefulness of discrete-event simulators as error-detection tools. It is our expectation that the advantages of our approach will lead to increased use of automated validation tools like these and, ultimately, to better protocols.

REFERENCES

[Baier et al., 1998] Baier, C., Kwiatkowska, M., and Norman, G. (1998). Computing probability lower and upper bounds for LTL formulae over sequential and concurrent Markov chains. *Proceedings of PROBMIV 98.*

[Daws and Yovine, 1995] Daws, C. and Yovine, S. (1995). Two examples of verification of multirate timed automata with KRONOS. *Proceedings, IEEE Real-Time Systems Symposium '95.*

[Gburzynski, 1996] Gburzynski, P. (1996). *Protocol Design for Local and Metropolitan Area Networks.* Prentice Hall.

[Hansson and Jonsson, 1989] Hansson, H. and Jonsson, B. (1989). A framework for reasoning about time and reliability. *Proceedings, IEEE Real-Time Systems Symposium '89*, pages 102–111.

[Holzmann, 1991] Holzmann, G. J. (1991). *Design and Validation of Computer Protocols.* Prentice Hall.

[Holzmann, 1992] Holzmann, G. J. (1992). Protocol design: Redefining the state of the art. *IEEE Software*, pages 17–22.

[Holzmann, 1997] Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, pages 279–295.

[Miller and Xue, 1996] Miller, R. E. and Xue, Y. (1996). Bridging the gap between formal specification and analysis of communication protocols. *Proceedings, 15th Annual Phoenix Conference on Computers and Communications*, pages 225–231.

[Yuan et al., 1997] Yuan, J., Shen, J., Abraham, J., and Aziz, A. (1997). On combining formal and informal verification. *Computer Aided Verification 97*, pages 376–387.