VALIDATION AND SIMULATION OF COMMUNICATION PROTOCOLS: AN INTEGRATED APPROACH

Theodore Ono-Tesfaye and Pawel Gburzynski*
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

We present a system for the simulation and validation of communication protocols. Our tool borrows ideas from SPIN [Holzmann, 1997], but in addition enables the modeling of timing and probability constraints because it is based on discrete event simulation. We describe a logic for specifying properties of communication protocols, and a model-checking algorithm for calculating the probability of such a property being satisfied by the protocol. Using a queueing system as an example, we show that the same model can be used to obtain performance measures by simulation and for model-checking.

Keywords: discrete event simulation, state exploration, protocol verification, protocol specification.

1 Introduction

The analysis of communication protocols encompasses the tasks of *performance evaluation* and *validation* (analysis for correctness). Due to the distributed nature of communication protocols and their resulting complexity, the use of automated tools for these tasks is highly desirable. Traditionally, the problems have been approached separately: discrete event simulation (DES) tools like OPNET or SMURPH are used for performance evaluation, while a variety of tools (e.g., SPIN [Holzmann, 1991, Holzmann, 1997], KRONOS [Henzinger et al., 1994, Daws and Yovine, 1995]) have been proposed for protocol validation.

In this paper, we seek to integrate the two tasks of performance evaluation and protocol validation by presenting protocol validation as an extension of a discrete event simulation tool. The advantages of implementing protocol validation as an extension of DES are that it allows for a realistic and natural modeling of protocols, including their timing and probability constraints, and that the same model can be used both for validation and performance evaluation. We describe a simple logic for expressing properties of sequences of events, and an algorithm for checking if a protocol satisfies such a property.

The fundamental problem that all validation methods face is the *state space explosion problem*—the fact that the state space of a protocol is exponential in the number of variables and processes. In those cases where protocols can not be exhaustively validated due to resource limitations, we propose probabilistic algorithms — based on Holzmann's efficient state space exploration [Holzmann, 1991, Holzmann, 1997] — for partial searches.

The rest of the paper is organized as follows. Section 2 describes our protocol specification method and a logic for specifying properties of protocols. Section 3 sketches 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 4. Finally, we summarize our conclusions in section 5.

2 Model

2.1 Protocol Models

In this section, we informally define two protocol modeling 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.

*e-mail: {theodore,pawel}@cs.ualberta.ca

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 $g \xrightarrow{e,t,p} g'$.

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 receives an event y, 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.

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).

2.2 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 $g_0 \xrightarrow{e_0,t_0,p_0} g_1 \xrightarrow{e_1,t_1,p_1} \ldots \xrightarrow{e_n,t_n,p_n} g_n$ 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 $e_i \models_e f$ if $f = \text{TRUE}$ or $f = e_i$, otherwise $e_i \not\models_e f$
- 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 $e_i \models_e f_1$ 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 3.2.

3 Algorithms

3.1 Basic State Space Search

Our basic algorithm is a depth-first search of the protocol state space. 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. No collision detection is performed and therefore, there is a small probability of mis-identifying a new state with one that was already visited. 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.

The bitstate hashing method is very efficient in that it requires only a few bits of storage per state. Nevertheless, the state space explosion problem means that industrial-sized protocols can often not be exhaustively validated, and the state space 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, a maximum time threshold, or a minimum probability threshold. When using a probability threshold, path probabilities are used as a heuristic 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. Time-limited and depth-limited searches can be handled in a similar manner.

3.2 Model-Checking of EL Formulas

We now briefly sketch 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. The inner DFS procedure returns the measure of the paths that were removed in this fashion.

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 keeps track of 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 by multiplying it with the transition probabilities and adding it to $N(g)$ for the nodes on the path. 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 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, and (ii) there is at least one transition between the two states whose probability is not equal to 1. 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. $I(g)$ 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. Note that $N(g) = 0$ and $I(g) =$0/1 does not imply that the final value of $N(g)$ $= 0$. 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.

3.3 Combining Model-Checking with Truncated Searches

As in the basic state space search (without model-checking), we propose limiting the state space search by probabilities (i.e., limiting the search to those states whose probability of occurring is greater than some threshold p), depth, or time.

To combine model-checking with such truncated searches, we first need to examine what the "measure of satisfying paths" means in the context of truncated searches. All paths in a truncated search are finite and have a probability which is the product of all transition probabilities along the path. The measure of satisfying paths is the sum of the probabilities of all paths that satisfy an EL formula f. So, in the case of a time-limited search with threshold t, the measure of satisfying paths is the measure of all paths with time-length $\leq t$ that satisfy f.

Using our model-checking algorithm in the limited search case presents us with problems because the algorithm assumes that all paths are infinite — this assumption is used in the way that cycles are handled. Therefore, when performing a truncated state space search, the model-checking algorithm must disable cycle-detection and search paths until the limit (given by depth, time, or probability) is reached.

We now consider the time-limited search as an example; the probability-limited and depth limited searches are analogous. Since the search tree of a protocol is typically exponential in t, simply searching the entire tree is inefficient, and we want to avoid re-visiting states as much as possible. In the time-limited search without model-checking, an array, indexed by the hash value of a state, indicates the smallest t for which a subtree rooted in that state has already been searched for errors. If the state is re-visited with a greater t than that given by the array (indicating that the current state is lower in the search tree), the state need not be re-visited, because the subtree rooted in the current state is included in the subtree that was already searched. If, on the other hand, the state is re-visited with a smaller t than that given by the array (indicating that the current state is higher in the search tree), then the state must be re-visited because the search tree rooted in the current state is larger than the one searched before.

To extend this approach to model-checking a formula f, we must consider how the depth of a state affects its $I()$-value. Recall that the $I(g)$ for a state g is either 0 (meaning that all paths starting at g satisfy f) , 1 (meaning that none of the paths starting at g satisfy f), 0/1 (meaning that g is part of a cycle), or UNKNOWN (the default).

Now, in a time-limited search, assume that the subtree rooted in g is searched, all paths starting at g are found to be non-satisfying, and $I(g)$ is set to 1 as a result. The situation then is similar to the case without model-checking: if the state g is re-visited with a lower t (higher in the tree), then the subtree need not be searched again because all paths starting at this g will inevitably be found to be non-satisfying as well. If the state g is re-visited with a higher t (lower in the

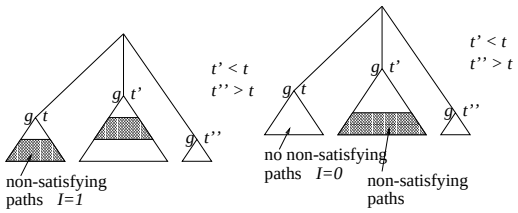

Figure 1: Re-visiting States in Limited Searches

	time-limited	probability-limited	depth-limited
$I = 0$	smallest t	largest p	smallest d
$I = 1$	largest t	smallest p	largest d

Table 1: Hash Array Contents for Limited Search

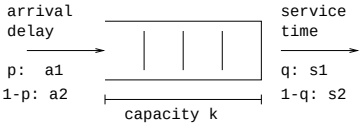

Figure 2: Simple Queue

k	p	a_1	a_2	q	s_1	s_2	¬OVER
5	0.1	5	10	0.8	5	10	FALSE
15	0.1	5	10	0.8	5	10	FALSE
5	0.1	5	20	0.5	7	8	FALSE
5	0.1	10	20	0.5	7	8	TRUE
15	0.1	5	10	0.8	1	2	TRUE

Table 2: Model-checking Results for ¬OVER

tree), the subtree needs to be searched again because paths starting at the new g are shorter an are non necessarily non-satisfying. Figure 1 (left) illustrates this reasoning.

This observation needs to be reversed for the case $I(g) = 0$: if g is re-visited with a lower t (higher in the search tree), then the subtree in g needs to be searched again; and if g is re-visited with a higher t (lower in the search tree), then the subtree need not be searched again (figure 1 right). Thus, our hash array needs to indicate for each g the "best" position for which a subtree rooted in g has been searched already; this can be either the smallest t (if $I(g) = 0$) or the largest t (if $I(g) = 1$). Table 1 summarizes what the array's contents for the different types of limited searches.

4 Experiments

We have implemented a prototype validation tool as a C++ library. [Ono-Tesfaye and Gburzynski, 1999] describes the modeling and validation of a multimedia stream synchronization system using our tool. In this section, we study a simple queueing system (figure 2). It consists of a packet queue with capacity k and a server. The interarrival time between packets is either a_1 (with probability p) or a_2 (with probability $1 - p$). Similarly, the service time for a packet is either s_1 (with probability q) or s_2 (with probability 1-q). The system is modelled using three event types: INIT (initialization), ARRIVAL (packet arrival), EOS (end-of-service). An additional event type — OVER — is generated whenever a buffer overflow occurs.

We can check if the queueing system is free from overflows, i.e., we can check for the EL formula ¬OVER. This is not very meaningful, however, as a simple analysis shows that the system is free from overflows if and only if both possible service times s_1 and s_2 are less than the shortest interarrival delay. If there is an s_i that is greater than one of the interarrival times a_j, then there always a non-zero probability that enough successive service times have duration s_i and enough successive interarrival times have duration a_j, eventually filling up the buffer. The model-checking results in table 2 confirm this view.

Therefore, in the case of a queueing system like ours, it is more meaningful to look at probabilistic performance measures. Certain questions regarding the performance of a protocol (e.g., the average delay of a queue) can be answered both by simulation and by checking for a property like "the probability of the queue length being greater than x is less than y". We want to compute the probability v of observing at least one buffer overflow within t time units of starting the system. This value can be quickly estimated by running a few simulations, or obtained exactly by using the validator to calculate the measure of paths with lengths $< t$ that satisfy the property ¬OVER.

We set $p = 0.9, a_1 = 5, a_2 = 10, q = 0.1, s_1 = 5, s_2 = 10$ and obtain the overflow probability for different values of t. Figure 3 shows the results obtained by validation and by simulation. The simulation results were obtained for 100 simulation runs; the figure shows the 95% confidence interval. In all cases, the (exact) overflow probabilities calculated by the validator are within the simulation confidence intervals.

Both approaches give quick results for small values of t. As t grows larger, the state space of the validator

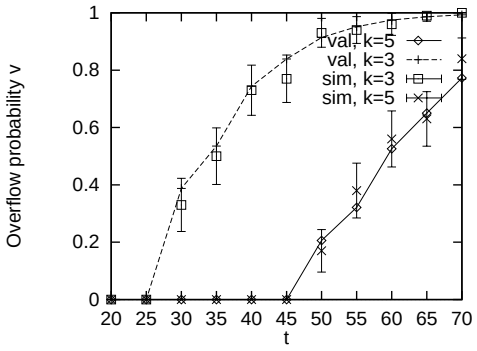

Figure 3: Overflow Probability, Simulation and Validation

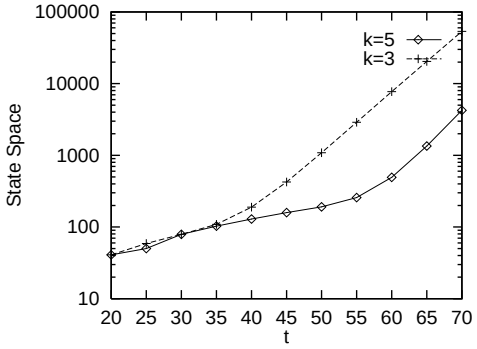

Figure 4: Size of State Space for Time-Limited Search

explodes (figure 4, the y-axis is log-scaled) and it becomes more feasible to estimate v via simulation. The reason for the inefficiency of the validator lies in our time-limited validation procedure (section 3.3) which works by assigning values 0 or 1 to states that are roots of subtrees that are satisfying or nonsatifying, respectively. In other words, for a state to be assigned $I = 1$, it must *inevitably* lead to an overflow event before the time limit t, and for a state to be assigned $I = 0$, it must be impossible for a state to lead to an overflow before the time limit. In the queueing system, such states are very rare — in most cases, it is possible but not inevitable that an overflow occurs, and the validator is forced to search a search tree whose size is exponential in t. Note that this reasoning does not extend to unlimited searches — such searches are generally much quicker: in the case $k = 5$, for example, the validator obtains the result that all paths are non-satisfying after searching only 35 states!

5 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. A major advantage of our approach is that the same model can be used both for the performance evaluation and for the validation of protocols.

We have described a simple logic to express properties of communication protocols, and a novel algorithm for efficiently calculating the probability of such a property being satisfied by a protocol. Using a queueing system as an example, we showed that our system is indeed an effective tool for analyzing the correctness and performance of a communication system. Further research should focus on extending these results to larger systems.

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. *Proc. PROBMIV 98.*

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

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

[Henzinger et al., 1994] Henzinger, T. A., Nicollin, X., Sifakis, S., and Yovine, S. (1994). Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244.

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

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

[Ono-Tesfaye and Gburzynski, 1999] Ono-Tesfaye, T. and Gburzynski, P. (1999). A discrete event simulation approach to protocol validation. *Proc. Euorpean Simulation Multiconference '99*, pp. 149–156.