Check and Simulate: A Case for Incorporating Model Checking in Network Simulation

Ahmed Sobeih, Mahesh Viswanathan* and Jennifer C. Hou
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{sobeih,vmahesh,jhou}@uiuc.edu

Abstract

Existing network simulators perform reasonably well in evaluating the performance of network protocols, but lack the capability of verifying and validating the correctness of network protocols. In this paper, we have extended J-Sim — an open-source, component-based compositional network simulation environment — with the model checking capability to explore the state space created by a network protocol until either the entire state space is explored (if the state space is finite) or an error (e.g., a violation of a user-defined safety assertion) is discovered. We also exploit protocol-specific properties in the process of exploring the state space, to reduce the size of the state space and to guide the (best-first) search towards paths that can potentially locate errors in less time. As a proof of concept, we have demonstrated use of the J-Sim model checker in locating errors in an automatic repeat request (ARQ) protocol. As compared to the Maude LTL model checker, the J-Sim model checker can locate errors in a more timely manner and with shorter error traces.

1. Introduction

Modern data communication networks are extremely complex and do not lend well to theoretical analysis. With computer/network entities and techniques interacting and interfering with one another, optimization problems do not have a simple and regular structure that allows us to neatly fit it into the framework of established optimization theories. As a result, it may be more feasible to carry out simulation to study and evaluate the performance of network entities and protocols, and interaction among them. Several existing network simulators (e.g., ns-2 [23] and J-Sim [15]) provide an environment in which a network protocol designer can build a protocol prototype, validate/evaluate its performance with respect to pre-selected networking metrics (e.g., system throughput, packet delivery ratio, and end-to-end delay) and re-design/refine the protocol if needed.

One major deficiency of existing network simulators is, however, that they only evaluate the performance of network protocols in several scenarios, but can *not* usually exhaust all the possible scenarios. If all the corner cases do not appear (and hence cannot be investigated) in the scenarios studied, subtle errors in the protocol specification/implementation may not be easily located in the simulation, and may manifest themselves after the protocol has been implemented and deployed. This could lead to serious, and sometimes disastrous, problems. For example, a routing protocol that may suffer from routing loops may cause data packets to loop in the network and not reach their destinations. Another example arises in the area of network security, where "holes" for security attacks may only be discovered after protocol implementation and deployment, causing severe damage to computer systems.

In the current practice, to check whether or not a network protocol contains any errors, a prototype that was specifically made for validation purpose has to be built (e.g., an interactive theorem prover and/or a model checker). This process is time-consuming, error-prone and requires tremendous efforts. An interesting question is then whether or not we can employ a single, integrated tool to provide both the *performance evaluation* and *validation* of network protocols. With such a tool, only one prototype will be built and used for the two purposes.

Motivated thus, we have extended J-Sim [15] — an open-source, component-based compositional network simulation environment — with capability to explore the state space created by a network protocol until either the entire state space is explored (if the state space is finite) or an error (e.g., a violation of a user-defined safety assertion) is discovered. In this paper, we will document our experiences in carrying out this research task. Note that our objective is to locate errors in the network protocol itself rather than errors in the simulation code. The basic idea is then to execute

*Supported by DARPA/AFOSR MURI Award F49620-02-1-0325

0-7803-8509-8/04/$20.00

the simulation code in order to pinpoint errors in the network protocol. Specifically, we have implemented a model checker (written in Java so that it can be readily integrated with J-Sim), and incorporated this model checker into J-Sim. As a proof-of-concept, we have used the J-Sim model checker to locate errors in a simple automatic repeat request (ARQ) stop-and-wait protocol. This first step demonstrates the feasibility of integrating model checking with network simulation.

To carry out the above tasks, we have addressed several issues. First, we have laid a framework that enables the model checker to take control of the simulation of a network protocol in order to explore the entire state space, rather than just exploring one single execution path as J-Sim does. Second we ensured that the implementation of this framework did not require the core design and implementation of J-Sim to be altered. Third, we have incorporated search techniques that are protocol specific that reduce the size of the explored state space and hence the time in detecting an error. Specifically, we make use of *protocol-specific* properties to reduce the size of the state space (e.g., by reducing the number of possible transitions), thereby reducing the time and/or space needed to locate an error. We also explore how a model checker can make use of best-first search that exploits properties inherent to the network protocol being checked, in order to guide the search towards paths that can potentially locate errors in less time. As compared to the Maude Linear Temporal Logic (LTL) model checker [7, 10], our framework enables discovery of errors with shorter counterexamples and in quicker time. These results are encouraging because the performance of the Maude LTL model checker has been shown in [10] to be comparable to that of SPIN [14].

The rest of the paper is organized as follows. We provide in Section 2 preliminary information on J-Sim and formal reasoning. Then we provide an overview of our proposed framework followed by a detailed description of our design and implementation in Section 3. Following that, we present our performance results in Section 4. Finally, we provide a taxonomy of related work in Section 5 and conclude the paper in Section 6 with a list of research avenues for future work.

2. Background

2.1. Network simulation and J-Sim

As mentioned in Section 1, due to the fact that communication networks are extremely complex, it is not unusual that theoretical network analysis can be rigorously made only after leaving out several (sometimes subtle) details that cannot be easily captured in the analysis [4, 21, 24, 28]. Packet-level, event-driven simulation studies are usually used as a replacement to better study the performance of network components, protocols, and their interaction.

Most notable research efforts on network simulation include: NEtwork Simulation Testbed (*NEST*) [27], The REalistic And Large (*REAL*) [16], ns-2 (ns version 2) [1], and *J-Sim* [15]. *NEST* [27] is a general-purpose simulation package designed to simulate distributed networked systems and protocols. It provides a client-server based graphical environment for simulation construction and execution control. The REalistic And Large (*REAL*) [16] network simulator is a substantially improved, and faster, version of NEST and is designed specifically for studying different congestion and flow control mechanisms in TCP/IP networks.

Ns-2 [1] began as a variant of the REAL network simulator [16], and has evolved substantially over the past few years. It provides substantial support for simulation of TCP, routing, and multicast protocols. However, due to its special node structure, it is non-trivial, and sometimes difficult, to include other protocols/algorithms or accommodate new network architectures in ns-2. In addition, the not-so-structured software architecture and the mixture of compiled and interpreted classes make it difficult to understand and validate ns-2 code.

J-Sim [15] is an open-source network simulation and emulation environment. It is implemented on top of a component based software architecture, called the *autonomous component architecture (ACA)*, that closely mimics the digital-circuit design. The basic entities in the ACA are *components*, which communicate with one another via sending/receiving data at their *ports*. How components behave (in terms of how a component handles and responds to data that arrive at a port) is specified at system design time in *contracts*, but their binding does not take place until the time when the system is being "composed." With the separation of contract binding (at system design time) from component binding (at system integration time), a component can be individually implemented and tested independently. When data arrives at a port of a component, the component processes the data immediately in an independent execution context (e.g., *thread* in Java). The interference between different data handled simultaneously by the same component is thus minimal, and is only subject to synchronization and mutual exclusion (in order to ensure the integrity of shared data). In some sense, the ACA realizes the notion of *software IC* because of this *message-passing, independence execution* model [15, 31].

By closing the gap between hardware and software ICs, the ACA provides composability, extensibility and the loose coupling feature between individual components [30]. All of these features enable new components to be included into J-Sim in a plug-and-play fashion. For this reason, we have determined to choose J-Sim as the network simulator to be

augmented with the protocol verification and validation capability.

2.2. Formal reasoning

In general, there are two basic approaches towards formal reasoning of software and hardware systems: *theorem proving* and *model checking*. In theorem proving, a formal technique (e.g., deductive methods and induction) is used to prove that the implementation of a system under study meets its specification. On the other hand, model checking checks a finite state machine model of the system in order to verify whether a (safety or liveness) property holds. Model checking [6, 2], by state space exploration, starts from an initial state of the system and recursively generates successor system states by executing the transitions (e.g., events) of the system. This process continues until either the entire state space is explored or an error is discovered. (Hence, in systems of infinite state space, model checking is used for locating errors, rather than proving correctness.)

As compared to theorem proving, model checking has several important advantages. First, it can be built into existing tools and automated. Second, model checking does not require a deep understanding of complex mathematical concepts. Third, when the desired property fails to hold, a model checker provides an error trace of the sequence of events that led to the error, that helps in understanding why the error occurred and how it can be fixed. For all the above reasons, we determine to incorporate a model checker in J-Sim.

One of the major challenges of using model checking is the well-known *state space explosion* problem, i.e., the state space of the system can be so prohibitively large that a model checker may run out of memory. Several approaches to handling the state space explosion problem (e.g., partial order reduction, abstraction, just to name a few) can be found in [6].

3. Check and Simulate

In the proposed framework, a network protocol designer specifies the assertions that should be maintained at any time during the execution of a network protocol. A user-defined safety assertion is generally a function (written in Java) whose input is the system state and output is true/false. The meaning of that function depends on the network protocol itself. For example, if the protocol is a routing protocol, the function may be that no routing loop is formed. If it is a security protocol, the function may be that an attack does not take place. If it is a reliable unicast/multicast protocol, the function may be that the receiver(s) receive all the packets that the sender believes to have been received. After the assertion is properly specified and the state defined, the model checker starts from an initial state of the system and recursively generates successor system states by executing the transitions of the system. This process continues until either the entire state space is explored or an error (e.g., a violation of the user-defined assertion) is discovered.

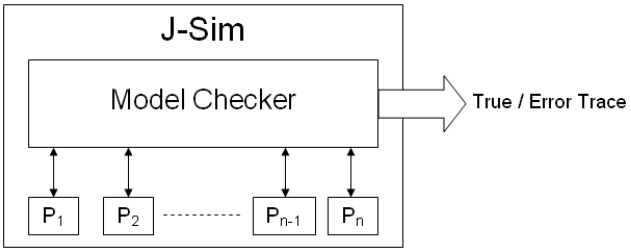

Figure 1. Overall framework of model checking in J-Sim.

There are three major design goals in building the framework of model checking into J-Sim (or any other network simulators):

1. The core implementation of J-Sim should not be modified.
2. Only a minimal modification to the J-Sim code that implements the network protocol is required.
3. The network protocol designer should not be overburdened with the details of the model checking process. He/she should only provide the protocol specification (in particular, the assertions that should be maintained in the course of verification and validation) and define the state of the protocol being model-checked, the set of events that may trigger state transition, and how the events are handled. Overall, the role of the network protocol designer in model checking should be to provide sufficient protocol specifics for the checking process to proceed.

To realize the above design goals, we have built a model checker (in Java) that checks a network protocol by executing the J-Sim simulation code of that network protocol *directly* and exploring the state space on the fly. Figure 1 illustrates the overall framework of incorporating model checking into J-Sim. As shown in Figure 1, the model checker interacts with instances of J-Sim classes, P_1, P_2, ... , P_n, that implement the entities comprising the network protocol being model-checked. The output of the model checker is either true (if the user-defined assertion is satisfied) or an error trace (i.e., a trace of transitions from the initial state to the state in which the user-defined assertion is not satisfied). We have built the model checker as an instance of a new component (*ModelChecking*) in J-Sim. The model checking procedure that explores the state space is implemented as one of the static member functions of the *ModelChecking* class. Figure 2 gives a pseudo-code description of the model checking procedure.

Definition of states Network simulation in J-Sim consists of a set of objects that make up a network (e.g., senders, receivers, routers, links, and protocols that run within each router/host). Each of these objects is an instance of a Java class in J-Sim. In order to explore the state space created by a network protocol, the notion of the "state" has to be adequately defined. A possible naïve definition of a state consists of all the data members of all the objects in the simulation. The problem with the above definition is the (perhaps) unnecessarily large size of the state space thus resulted. For the purpose of model-checking a network protocol, some objects may not be relevant and some data members may not be necessary. On the other hand, the model checking infrastructure should be laid in a sufficiently generic manner and not be tied to a specific network protocol. To this end, we have defined and implemented another class, *GlobalState*, which includes only the relevant state variables (e.g., state variables needed to generate the successor states and state variables needed to check whether a state satisfies the user-defined assertions).

As shown in Figure 2, the two major data structures are *NonVisitedStates* (which was implemented as a linked list storing the states that have not yet been visited) and *AlreadyVisitedStates* (which was implemented as a hash table storing the states that have already been visited). Figure 2 presents a stateful search that avoids visiting a state that has already been visited before (i.e., a state that already exists in *AlreadyVisitedStates*). Each state in the state space of a network protocol is an instance of the *GlobalState* class. For example, in a typical end-to-end network protocol, *GlobalState* may have two components: *ProtocolState* and *NetworkCloud*; the former represents the state of the end-hosts (e.g., the sender and the receiver) while the latter represents the data and control packets that are currently in transit in the network. It should be noted that the implementation of *GlobalState* differs from one network protocol to another; hence, it is the responsibility of the protocol designer to provide an implementation of *GlobalState*.

The model checking procedure, shown in Figure 2, keeps track of three instances of *GlobalState*; namely, *initialState* (the initial state of the network protocol), *currentState* (the current state being explored) and *nextState* (one of the possible successors of the current state). It should also be mentioned that the protocol designer should specify how the initial state of his/her network protocol can be constructed.

State transition After defining what a state is, the next step is to define what a "transition" is. In each state in the state space, some events (i.e., transitions) may or may not occur. Examples of events in a network protocol are: packet arrival, packet loss and timeout. It is also the responsibility of the network protocol designer to specify (a) the set of events that exist in the network protocol, (b) when each event occurs (e.g., a packet arrival event occurs at a node *n* only if the network contains a packet whose destination is *n*) and (c) how each event is handled (i.e., an event handler function that makes a transition from one state to another). Note that the network protocol designer has to write the event handler in order to have a working prototype of the network protocol in J-Sim even if he/she does not intend to model check the protocol.

To help the protocol designer in defining the events that trigger the state transition, we have made use of the *reflection* facilities [11] of the Java language and implemented a *Transition* class as follows:

```
class Transition
{
   /* check if the event can take place */
   java.lang.reflect.Method  EnablingFunction ;
   /* define how each event is handled */
   java.lang.reflect.Method  EventHandler ;
   ................
}
```

For each possible event in the network protocol, the protocol designer needs only to (1) create an instance of the *Transition* class, (2) use the *java.lang.Class.getMethod()* function to return a *Method* object that reflects the event's enabling function and event's event handler, and (3) use the *java.lang.reflect.Method.invoke()* function to invoke the enabling function and the event handler. This is shown in Figure 2 as *e.EnablingFunction()* (line 8) and *e.EventHandler()* (line 11) respectively.

The model checking process For each state being explored (*currentState*), the model checking procedure generates all the possible successor states (*nextState*) by executing the event handlers of the events that can occur in *currentState*. However, since an event handler is only invoked from the model checking procedure but actually executed inside the protocol entities themselves, the model checking procedure must first restore the state of the protocol entities to the state reflected in *currentState* before the execution of the event handler. This is achieved by the *CopyFromModelToEntities()* function call (Figure 2, line 9). After the execution of the event handler (Figure 2 , line 11), the *CopyFromEntitiesToModel()* function is called (Figure 2, line 12) to extract the new state information from the protocol entities and copy them to *nextState*. The model checking procedure then checks (via *nextState.verify()*) whether *nextState* does not satisfy the user-defined assertions (Figure 2, line 14). If *nextState* does satisfy them, it is added to *NonVisitedStates* (Figure 2, line 21) in order to be explored later; otherwise an error trace showing the sequence of states that led to the error is printed by calling the *printErrorTrace()* function (Figure 2, line 18). The *printErrorTrace()* function is a recursive function that traces the state space backwards until the initial state is reached.

Implementation We have encountered two major implementation problems in the course of incorporating the

```
Procedure modelCheck() {
    /* The following are static data members of the ModelChecking class:
       AlreadyVisitedStates, NonVisitedStates, initialState, currentState, nextState */
 1. AlreadyVisitedStates = { } ;
 2. NonVisitedStates = { initialState } ;
 3. while ( | NonVisitedStates | > 0 ) {
 4.    currentState = NonVisitedStates.removeFirst() ;
       /* Explore currentState only if it has not been visited before */
 5.    if ( currentState does not exist in AlreadyVisitedStates ) {
 6.       AlreadyVisitedStates = AlreadyVisitedStates ∪ { currentState } ;
 7.       for ( all possible events e ) { /* for all events that may take place */
 8.          if (e.EnablingFunction() returns true) { /* if e can take place */
 9.             CopyFromModelToEntities(currentState) ;  /* Copy the relevant state information from
                                                           currentState to the protocol entities */
10.             nextState = currentState ;  /* Start with nextState equal to currentState */
11.             e.EventHandler() ;  /* Invoke e's event handler */
12.             CopyFromEntitiesToModel(nextState) ;  /* Copy the new relevant state information from
                                                         the protocol entities to nextState */
13.             if (nextState does not exist in AlreadyVisitedStates) {
14.                if ( nextState.verify() == false ) { /* Check if the user-defined assertion(s) were violated */
15.                   Print("modelCheck: FOUND ERROR STATE ") ;
16.                   nextState.printState() ;  /* Print the error state */
17.                   Print("ERROR TRACE ") ;
18.                   printErrorTrace(nextState) ;  /* Print the error trace */
19.                   exit ;
                   }  /* end if the user-defined assertion(s) were violated */
20.                else {
21.                   NonVisitedStates = NonVisitedStates ∪ { nextState } ;
                   }  /* end else */
                }  /* end if nextState does not exist in AlreadyVisitedStates */
             }  /* end if e can take place */
          }  /* end for all possible events e */
       }  /* end if currentState has not been visited before */
    }  /* end while NonVisitedStates is not empty */
22. Print("No Error States were found. Ending model checking") ;
23. exit ;
}
```

Figure 2. Stateful on-the-fly model checking procedure.

model checker into J-Sim: one is related to how network protocol entities communicate with each other, with the model checker in between; and the other is related to the ACA timers. We describe below each of them and how we solved them while keeping our design goals met.

Without model checking, protocol entities communicate with each other via ports. However, when the network protocol is model-checked and the model checker is used as shown in Figure 1, the protocol entities need to communicate with each other via the model checker. Initially, we simply connected the ports of each protocol entity to those of the model checker, but then found that protocol-specific data/control messages generated by the protocol entities during the execution of an event handler may not be forwarded to the model checker at the required time. This is because the model checker does not wait until the protocol entities finish executing an event handler. This may cause the model checker to exclude some of the new state's information in *nextState*. We solved this problem by setting the ports that are involved in the interaction between the model checker and the protocol entities to the *function-call execution model* instead of the default *independence execution model* [30]. In the function-call execution model, the model checker *waits* until the protocol entities finish executing an event handler; therefore, this solution ensures that all the new state's information will be included in *nextState*. Although this solution requires modest modification to the J-Sim simulation code of the network protocol, we believe that this modification is minimal. Alternatively, one may make the modification in a subclass of the J-Sim class of a network protocol entity, thus keeping the original parent J-Sim class unmodified.

The second problem is related to the ACA timers. Without model checking, a protocol entity that uses an ACA timer sets the timer to a pre-determined time interval. Upon timer expiration, a timeout event is triggered if the timer is still active. If the network protocol is to be model-checked, the model checker should explore all the possible transitions from a given state, and should not be limited to a single timeout value for each timer[1]. Instead, the model checker should trigger the timeout event when that event may occur in the real world. For example, a typical retransmission timer in a reliable unicast protocol may expire at any time as long as there is a pending data message that has been sent but not yet acknowledged. This problem can be easily solved by modifying the core implementation of J-Sim. However, this violates our first design goal. Therefore, we chose to make this modification also at a subclass of the J-Sim class of a network protocol entity.

[1]We assume that setting of the interval of a timer may differ from one run of the protocol to another; otherwise, this approach may suffer from excessive false positives.

4. Evaluation and results

As a proof-of-concept of this innovative idea of integrating model checking with network simulation, we have used the J-Sim model checker to model-check an automatic repeat request (ARQ) protocol. In what follows, we first summarize the ARQ protocol and then present our experimental results. For the purpose of benchmarking, we have also compared the performance of the J-Sim model checker against the Maude LTL model checker.

4.1. Automatic Repeat reQuest (ARQ)

Automatic Repeat reQuest (ARQ) is a well-known error control protocol that has several variations. The simplest form of the ARQ protocol is stop-and-wait ARQ in which the sender sends a single data packet and then waits for a positive acknowledgment (ACK) before it advances to the next data packet. The receiver only replies with an ACK if the data packet is correctly received. As either the data packet or the corresponding ACK may be lost/corrupted in transit, after the sender sends a data packet, it sets a retransmission timer to a pre-determined value. If no ACK is received before the retransmission timer expires, the sender simply retransmits the data packet.

For the receiver to distinguish between a data packet that is sent for the first time and a retransmission, a sequence number is included in the header of each data packet. For stop-and-wait ARQ, it is sufficient that the sequence number be 1-bit (i.e., either 0 or 1) because the only ambiguity is between a data packet and its immediate predecessor and successor, but not between the predecessor and successor themselves [29]. For similar reasons, each ACK should also contain a sequence number. In the common practice, the sequence number in the ACK is the sequence number of the next expected data packet rather than the sequence number of the data packet that has been recently received. It should be mentioned that setting of the timeout interval at the sender is very important, and is a trade-off between premature timeout and prolonged retransmission.

Stop-and-wait ARQ ensures that every data packet sent by the sender will eventually be received correctly by the receiver and that the receiver will get the data packets in order, i.e., it is an in-order reliable unicast protocol. Although stop-and-wait ARQ described above is fairly simple, it is the fundamental basis of the error control mechanism in TCP (which uses a more complex sliding-window ARQ).

Before model-checking the ARQ protocol in J-Sim, we injected an error as follows. Suppose a protocol designer implements stop-and-wait ARQ in J-Sim, but does not include the sequence number in the ACK. In this case, the protocol designer may obtain an output trace similar to that shown in Figure 3 in a typical J-Sim simulation run.

```
TCL0> Sender: sending DataMessage 0
Receiver: receiving EXPECTED DataMessage: 0
Receiver: sending ACKMessage.
Sender: receiving ACKMessage
Sender: sending DataMessage 1
Receiver: receiving EXPECTED DataMessage: 1
Receiver: sending ACKMessage.
Sender: receiving ACKMessage
Sender: sending DataMessage 0
Receiver: receiving EXPECTED DataMessage: 0
Receiver: sending ACKMessage.
Sender: receiving ACKMessage
Sender: sending DataMessage 1
Receiver: receiving EXPECTED DataMessage: 1
Receiver: sending ACKMessage.
Sender: receiving ACKMessage
Sender: sending DataMessage 0
.....................
```

Figure 3. A sample output trace of a stop-and-wait ARQ protocol.

(Note that in order to produce the output shown in Figure 3, we simulated the ARQ protocol (with no sequence numbers in the ACK packet) in J-Sim by implementing four new J-Sim classes, namely, *Sender*, *Receiver*, *DataMessage* and *ACKMessage*.) Repeating the same experiment several times with different network topologies between the sender and the receiver may also produce the same output. This may lure into believing that this version of ARQ operates correctly. However, as will be demonstrated later in this section, not including a sequence number in the ACK may lead to an error.

4.2. Model checking the ARQ protocol

Recall that the first step in model checking is to specify the properties to be verified. An important safety property for any reliable unicast protocol is that the receiver does not miss any data packet that the sender believes to have been received by the receiver. In a reliable unicast protocol (e.g., the TCP protocol), the sender typically temporarily saves all the transmitted but not yet acknowledged data packets until it receives an ACK that acknowledges receipt of these data packets. In an ARQ protocol that uses a 1-bit sequence number, this safety property translates to the requirement that the difference between the total number of distinct data packets transmitted by the sender (*sender_NumDistinctDataMsgSent*) and the total number of distinct data packets received by the receiver (*receiver_NumDistinctDataMsgReceived*) is always less than or equal to 2. In LTL notation, this safety property can be defined as follows:

```
□ ( (sender_NumDistinctDataMsgSent
      - receiver_NumDistinctDataMsgReceived
    ) <= 2)
```

The next step is to specify what constitutes a state (i.e., provide an implementation for the *GlobalState* class mentioned in Section 3). For an end-to-end protocol such as

stop-and-wait ARQ, a sufficient definition of the global state is a tuple that has two components; namely, the protocol state (instance of *ProtocolState* class) and the network cloud (instance of *NetworkCloud* class). The *NetworkCloud* class models the network as a black box that contains the data packets and the ACK packets. The *ProtocolState* class contains data members that represent the state of the protocol entities; namely, the sender and the receiver. It should be noted that the *ProtocolState* class does not have to include all of the data members of the *Sender* and *Receiver* classes; instead, it just needs to include the relevant ones; e.g., *sender_NumDistinctDataMsgSent* and *receiver_NumDistinctDataMsgReceived*.

Next, the protocol designer needs to specify the initial state. A reasonable initial state is the state in which the sender has just sent the first data packet (denoted as *D0*), the receiver is expecting *D0* and the network contains *D0*. As mentioned above, the protocol designer should also specify the set of possible events for the network protocol, when each event occurs and how each event is handled. In the context of stop-and-wait ARQ, there are five events: arrival of a data packet, arrival of an ACK packet, timeout, loss of a data packet and loss of an ACK.

4.3. Use of protocol-specific abstractions in defining events and facilitating best-first search

As mentioned in Section 1, to reduce the size of the state space and to expedite the checking process, it is desirable to use properties that are specific to a network protocol to explore the state space and/or to guide the search towards paths that can potentially locate errors in less time. In the context of stop-and-wait ARQ, we have made use of the following protocol-specific properties:

1. ACKs are not lost. The effect that an ACK is lost is equivalent to the effect that the corresponding data packet is lost in the perspective of the sender. With this assumption, the model checker can focus on only four events rather than five.

2. The network between the sender and the receiver is modeled as a network cloud without specifying the structure of the network. This is reasonable because the ARQ protocol is essentially an end-to-end protocol and the safety properties that are verified depend only on the state of the two end points.

3. In conjunction with property 2, since the network is modeled as a cloud, we do not have to distinguish all the possible causes of packet loss (e.g., router/link failure, congestion, packet corruption as a result of transmission errors). This allows us to consider packet loss as a one single event regardless of the cause of packet loss thereby reducing the total number of events.

4. Data packets are not reordered inside the network. As a packet arriving earlier than expected will be discarded by the receiver, a packet reordering event can be treated as a packet loss event.

In addition to the properties mentioned above, we also make use of best-first search in order to explore states that may potentially lead to an error state first. Again, protocol-specific metrics are exploited to specify how a state is considered "better" or more likely to lead to an error state.

A suitable best-first strategy for the ARQ protocol presented above is to consider a state s_1 better than another state s_2 if the quantity $(sender\_NumDistinctDataMsgSent - receiver\_NumDistinctDataMsgReceived)$ in s_1 is greater than the corresponding quantity in s_2. This is because the greater this quantity, the more packets are believed to be received but actually they are not; in some sense, the sender is "making more progress" than the receiver.

4.4. Effect of protocol-specific abstractions and search strategy

In order to evaluate the effect of exploiting protocol-specific abstractions in defining events and in guiding best-first search, we present in this section experimental results obtained using breadth-first/best-first search with and without the first protocol-specific property (ACKs are not lost).

Using breadth-first search together with all of the protocol-specific abstractions, we obtain the error trace given in Figure 4. In State 3, the retransmission timer expires prematurely causing the sender to retransmit *D0*. In State 4, the receiver discards the duplicate *D0* and retransmits the ACK. In State 5, the sender receives an ACK and transmits *D1*. In State 6, the sender receives the other ACK and since the ACK does not carry a sequence number, the sender thinks it is acknowledging the receipt of *D1* sent in State 5. In State 7, *D1* is lost; i.e., the receiver will miss *D1* although the sender believes that *D1* has been received by the receiver. In State 8, the receiver transmits another ACK. Finally, State 9 represents a state from which the ARQ protocol can resume without the sender and the receiver noticing that an error has happened. In other words, the ARQ protocol fails. Without using the first protocol-specific abstraction, we obtain the same error trace but in longer time, because the model checker has to handle more events. The first column of Table 1 gives the time needed to locate the error (averaged over 5 runs) in both cases.

Using best-first search together with all of the protocol-specific abstractions, we obtain a *longer* error trace (as expected); however, it was obtained in less time because the best-first strategy successfully guides the search towards the states that may potentially lead to an error state. The second

```
TCL0> modelCheck: FOUND ERROR
        Sender: Last Seq. No. Sent: 1
        Receiver: Expected Seq. No.: 1
        Network: D1

ERROR TRACE
State 1
        Sender: Last Seq. No. Sent: 0
        Receiver: Expected Seq. No.: 0
        Network: D0
State 2
        Sender: Last Seq. No. Sent: 0
        Receiver: Expected Seq. No.: 1
        Network: ACK
State 3
        Sender: Last Seq. No. Sent: 0
        Receiver: Expected Seq. No.: 1
        Network: D0,ACK
State 4
        Sender: Last Seq. No. Sent: 0
        Receiver: Expected Seq. No.: 1
        Network: ACK,ACK
State 5
        Sender: Last Seq. No. Sent: 1
        Receiver: Expected Seq. No.: 1
        Network: D1,ACK
State 6
        Sender: Last Seq. No. Sent: 0
        Receiver: Expected Seq. No.: 1
        Network: D1,D0
State 7
        Sender: Last Seq. No. Sent: 0
        Receiver: Expected Seq. No.: 1
        Network: D0
State 8
        Sender: Last Seq. No. Sent: 0
        Receiver: Expected Seq. No.: 1
        Network: ACK
State 9
        Sender: Last Seq. No. Sent: 1
        Receiver: Expected Seq. No.: 1
        Network: D1
```

Figure 4. Error trace obtained using both breadth-first search and protocol-specific properties.

column of Table 1 gives the time needed to locate the error (averaged over 5 runs).

4.5. Comparison with Maude LTL model checker

To compare the performance of the J-Sim model checker with the Maude LTL model checker, we have first written a specification of the ARQ protocol in Full Maude as an object-oriented module in rewriting logic and then used the Maude LTL model checker to discover the error and the error trace (called *counterexample* in Maude).

In order to make a fair comparison, we have included the same set of protocol-specific abstractions in the Maude specification. Furthermore, we have also considered the two cases of whether or not the first property that ACKs are not lost is used. However, the Maude LTL model checker was not able to handle the infinite state space of the ARQ protocol and returned a "Segmentation Fault" (because of following a depth-first search and running out of memory). In order to remedy that, we enforce the state space to be finite by limiting the total number of data packets (both original and retransmissions) that the sender can send.

Tables 1–2 give, respectively, the execution time needed to locate the error and the length of the error trace. As shown in Table 2, the error traces obtained by the J-Sim model checker are shorter than those obtained by the Maude LTL model checker. The is because the Maude LTL model checker may sometimes oscillate between data packet loss and data packet retransmission on sender timeout events (which counteract each other). This causes the error trace to be longer (Table 2) and the time needed to locate the error to be larger (Table 1) than those obtained by the J-Sim model checker. The situation becomes even worse in the case of larger state spaces (because of more of such oscillations) where the J-Sim model checker significantly outperforms the Maude LTL model checker both in terms of the time overhead and the length of the error trace[2]. Finally, and most importantly, use of J-Sim saves the network protocol designer from having to build another prototype in Maude.

5. Related work

Conventional model checkers (e.g., SPIN [14], SMV [20], Murphi [9]) require that the system be first specified using a high-level modeling language. The process of describing the system in a high-level modeling language is a time-consuming, painstaking, and error-prone process. To deal with this, there has been some work (e.g., [25], Java PathFinder [13], and Bandera [8]) on translating programming languages (e.g., Java) into the input modeling languages of several conventional model checkers. The idea is to automatically extract an abstract model out of an application written in Java and then use conventional model checking to analyze this abstract model. However, this may not be always feasible, as it requires that each language feature of Java must have a corresponding one in the destination modeling language.

Conventional model checkers have also been used in formal reasoning of distributed systems. In [33], SMV is used to verify three cache coherence protocols used in distributed file systems; however, each cache coherence protocol has to be modeled using the SMV input language and then SMV checks that model rather than the actual implementation. In [18], the process of writing the model is automated by using an extensible compiler, xg++, that can automatically extract a model (described in the Murphi input language) from the original implementation code. Compared to [18], our goal is not to extract a model, but instead to model check the J-Sim simulation code itself. Teapot [5] is a domain-specific

[2]It should be noted, however, that Maude has another command, *search*, that uses a breadth-first search and can thus give the shortest path to the counterexample.

Table 1. Time needed to locate the error (in msec.). The number between brackets is the limit on the total number of data packets sent by the sender. The result in the last column is N/A because the Maude model checker did not produce a counterexample and returned the "Segmentation Fault" message.

	J-Sim Breadth-first Search (∞)	J-Sim Best-first Search (∞)	Maude LTL (10)	Maude LTL (100)	Maude LTL (1000)	Maude LTL (∞)
Without protocol-specific property no. 1	16.8	12.4	23.4	32.2	119	N/A
With protocol-specific property no. 1	14	10	20.2	26.4	113.8	N/A

Table 2. Length of the error trace (in number of states).

	J-Sim Breadth-first Search (∞)	J-Sim Best-first Search (∞)	Maude LTL (10)	Maude LTL (100)	Maude LTL (1000)	Maude LTL (∞)
With protocol-specific property no. 1	9	11	19	199	1999	N/A

language for writing cache coherence protocols, and offers further improvement over [33] and [18]. The Teapot compiler can translate a protocol specification to both executable C code and code that can be input to Murphi, and hence potential discrepancies between the specification and the actual executable code can be eliminated. Our approach differs from Teapot in that it does not require an input protocol specification and does not generate an output executable code. Furthermore, since the model checker is built in J-Sim, there is no need to use an existing model checker such as Murphi.

As mentioned above, the J-Sim model checker checks a network protocol by executing the J-Sim simulation code of that network protocol *directly* and exploring the state space *on the fly*. This is inspired by previous work on model checking the implementation code directly (e.g., CMC [22] and Verisoft [12]) for C and C++. Although CMC has been applied to model check implementations of networking code (namely, the AODV routing protocol [26]), our approach differs from CMC in two aspects: (a) our goal is to model check the network protocol while it is being designed (using J-Sim) rather than after it is implemented; and (b) we focus on Java rather than C or C++.

A more recent version of Java PathFinder [32] performs model checking at the bytecode level. However, this involves building a new Java Virtual Machine JVM^{JPF}, which is called from the model checker, to interpret bytecode generated by a Java compiler. Compared to [32], our approach has the important advantage of not requiring any modifications to the Java Virtual Machine. As far as formal analysis of network simulation is concerned, the only existing work is Verisim [3], which was developed based on a collection of pre-existing tools, i.e., ns-2 [23] and the MaC monitoring and checking framework [17]. Verisim replaces the monitor component of MaC by ns-2 and uses the checker component of MaC to verify user-defined properties on traces produced by ns-2. It should be noted, however, that not all errors may manifest themselves in a trace because ns-2 does not explore all possible execution paths during a simulation run.

Maude [19] is a reflective language and system that supports both equational and rewriting logic specification and programming. Maude is extremely powerful and can be used to create *executable* specifications for a wide range of applications (e.g., other languages, theorem provers, concurrent systems). In fact, Maude can even be used to build language extensions for Maude itself. For example, Full Maude is implemented in Maude as an extension of (Core) Maude [7]. Concurrent object-oriented systems can be specified in Full Maude by means of object-oriented modules that can be executed and also model-checked with a Linear Temporal Logic (LTL) model checker. We have compared the performance of the J-Sim model checker against that of Maude, and shown that the former outperforms the latter in model checking stop-and-wait ARQ.

6. Conclusions and future work

In this paper, we have made a case for incorporating model checking in network simulation. We have built a model checking framework in J-Sim and experimented J-Sim, together with the model checking framework, with respect to the capability of locating errors in a stop-and-wait ARQ reliable unicast protocol. Experimental results have shown that the model checking framework is able to find safety property violations within acceptable time. Further-

more, protocol-specific abstractions that are inherent to network protocols expedite the process of locating such violations. Use of a best-first search strategy also significantly reduces the time needed to find these violations. As compared to the Maude LTL model checker, the J-Sim model checker can locate errors in a more timely manner and with shorter error traces.

We have identified several research avenues for future work. First, we intend to experiment the J-Sim model checker with more complex network protocols (e.g., AODV in mobile ad hoc networks and/or directed diffusion in wireless sensor networks). Second, we intend to extend the model checking framework to check general LTL formulae. Third, we will study how the model checking framework can model-check network protocols that involve complex interactions between various timers. Fourth, we will further reduce the intervention of the protocol designer with model checking by automatically (i) extracting the specification of the global state from the code of the user-defined assertions and (ii) building the initial state of the network protocol from the constructor functions of the J-Sim classes of that protocol. Finally, we will investigate a modified version of the best-first search that instead of deterministically exploring the best state first, makes a randomized choice that is biased towards the best states. The error traces will still be longer than those obtained by breadth-first search but may be shorter than those obtained by best-first search.

References

[1] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. Improving simulation for network research. Technical Report 99-702, University of Southern California, 1999.

[2] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Verlag, 2001.

[3] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: formal analysis of network simulations. *IEEE Trans. on Software Engineering*, 28(2):129–145, February 2002.

[4] F. Calí, M. Conti, and E. Gregori. Dynamic tuning of the IEEE 802.11 protocol to achieve a theoretical throughput limit. *IEEE/ACM Trans. on Networking*, 8(6):785–799, December 2000.

[5] S. Chandra, B. Richards, and J. R. Larus. Teapot: a domain-specific language for writing cache coherence protocols. *IEEE Trans. on Software Engineering*, 25(3):317–333, May-June 1999.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual*, July 2003.

[8] J. Corbett, M. Dwyer, J. Hatcliff, C. Păsăreanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite state models from Java source code. In *Proc. of ICSE'00*.

[9] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proc. of IEEE ICCD'92*.

[10] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In *Proc. of WRLA'02*.

[11] D. Flanagan. *Java In A Nutshell*. O'Reilly, 1997.

[12] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. of ACM POPL'97*.

[13] K. Havelund. Java Pathfinder, a translator from Java to Promela. In *Proc. of SPIN'99*.

[14] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[15] J-Sim. http://www.j-sim.org/.

[16] S. Keshav. REAL: A network simulator. Technical Report 88/472, University of California, Berkeley, 1988.

[17] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Proc. of ECRTS'99*.

[18] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In *Proc. of ISCA'01*.

[19] Maude. http://maude.cs.uiuc.edu/.

[20] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[21] V. Misra, W. Gong, and D. Towsley. Stochastic differential equation modeling and analysis of TCP windowsize behavior. In *Proc. of Performance'99*.

[22] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proc. of OSDI'02*.

[23] Ns-2. http://www.isi.edu/nsnam/ns/.

[24] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proc. of ACM SIGCOMM'98*.

[25] D. Y. Park, U. Stern, J. U. Skakkebæk, and D. L. Dill. Java model checking. In *Proc. of IEEE ASE'00*.

[26] C. Perkins, E. Royer, and S. Das. Ad hoc on demand distance vector (aodv) routing. IETF Draft, January 2002.

[27] D. J. Schwartz, Y. Yemini, and D. Bacon. NEST: A network simulation and prototyping testbed. *Communications of the ACM*, 33(10):63–74, October 1990.

[28] S. Shakkottai and R. Srikant. How good are deterministic fluid models of internet congestion control? In *Proc. of IEEE INFOCOM'02*.

[29] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall International Inc., 1996.

[30] H.-Y. Tyan. *Design, Realization and Evaluation of a Component-based Compositional Software Architecture for Network Simulation*. PhD thesis, Department of Electrical Engineering, The Ohio State University, 2002.

[31] H.-Y. Tyan and J. C. Hou. *JavaSim*: A component-based compositional network simulation environment. In *Proc. of Western Simulation Multiconference, CNDS'01*.

[32] W. Visser, K. Havelund, G. Brat, and S.Park. Model checking programs. In *Proc. of IEEE ASE'00*.

[33] J. M. Wing and M. Vaziri-Farahani. Model checking software systems: a case study. In *Proc. of ACM SIGSOFT'95*.