# Simple Network Protocol Simulation within Maude

Ian A. Mason

*University of New England*
iam@turing.une.edu.au

Carolyn L. Talcott

*Stanford University*
clt@cs.stanford.edu

## 1 Introduction

On the one hand network and communication protocols are complex and difficult to design, on the other hand it is important that network systems are robust and reliable. Thus it is desirable to have formal models augmented with tools that support simulation and testing that can be used by designers of new protocols, both in the early stages of design, as well as in later stages, where more rigorous formal analysis is important. Rewriting logic and the language Maude [4,9] with its support of executable specifications and its ability to incorporate a wide spectrum of formal methods [7] is an ideal framework for developing such tools. There is already considerable experience in modeling and analyzing specific protocols within Maude (see [7] for a summary).

In this paper we present the specification of a network model in Maude and some primitives for defining simulation strategies. The use of the model is illustrated with a simple HELLO sub-protocol taken from the IETF PIM-DM (Protocol Independent Multi-Cast-Dense Mode) RFC [6], and based on a pseudo-code specification [21]. The network model we present reflects the key aspects of the infra-structure on which typical communication protocols run. The model is designed so that we may execute isolated protocols as well as develop techniques for composing sub-protocols, to model the more complex protocols used in practice. The long term goal is to support simulation and formal analysis at many levels of detail.

Other approaches to modeling and analysis of network protocols include: discrete event simulation such as NS [3,13]; model checking (SPIN [11,12], FDR [20], Murphi [5]); and using general purpose proof systems such as PVS [18], HOL [10], or Isabelle [19]. An NS simulation runs code that can be quite close to actual implementation using traffic and network topology

*This is a preliminary version. The final version will be published in*
*Electronic Notes in Theoretical Computer Science*
*URL:* www.elsevier.nl/locate/entcs

generators. The result of an NS simulation is a trace of all packets produced, transported, dropped in the network, and any other diagnostic information directly instrumented into the protocol simulation code. The focus is on performance information and there is little support for analyzing other properties such as reliability or security properties. Verisim [1] combines NS and Java-Mac, to allow for expressing and checking properties of the network state and history, still based on simulation code. Model checking and theorem proving approaches work with more expressive specification languages, but analyse abstract protocol representations. Using general purpose theorem provers allows for reasoning about all computations and arbitrary network size and topology, but requires considerable time and expertise to carry out the formal proofs. Model checking is more highly automated, but for a more restricted set of properties of specific system configurations. The SPIN model checker provides support for both simulation and model checking. Combining theorem proving and model-checking can provide some of the benefits of each. A good example of this can be found in [2]. Our approach is based on executable Maude specifications of network services. Maude specfications are closer to implementation code, while being simpler to reason about. They can also be used as a step towards code generation, either using a special purpose tool or using a Maude compiler [in progress]. On the other hand the Maude specification has a formal semantics that serves as the basis not only for analysis of simulation results but also for reasoning both manual and automated about general properties of a specification and its instantiations. The current work is a first step towards developing flexible network simulation tools. Starting with a detailed representation of the computation chosen by a strategy we define simplification functions the keep information of interest, which may include selective information about packet transmission as well as information about the 'knowledge' represented by the network state and different point in the computation. Our approach to modeling real-time properties in rewriting logic follows that of [16], adding clocks and timers as attributes to objects and separating the transitions modeling passage of time from those modeling state change. Like part of the work on the Real-Time Maude tool [17] we are developing strategies for simulation. The Real-Time Maude tool aims at general purpose strategies for simulation and model-checking, using concepts like eager and lazy rules, firing all eager rules before time passes. Our work is currently focused one modeling properties of network protocols. The basic idea for our primitive strategy is to allocate 'gas' to each network entity, and time passes when each entity has used up its gas or has no transitions.

## 2 Concurrent Objects in Rewriting Logic

We briefly introduce the rewriting logic model for concurrent object systems that we use for our network model. Following the approach of Meseguer [15], the concurrent state of an object system, called a *configuration*, has the structure of a *multiset* made up of objects and messages. The associativity and commutativity of a configuration's multiset structure makes it very fluid. We can think of it as a *soup* in which objects and messages float, so that objects and messages can at any time come together and participate in a concurrent transition corresponding to a communication event of some kind.

An *object* in a given state is represented as a term $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where $O$ is the object's name or identifier, $C$ is its class, the $a_i$'s are the object's *attribute identifiers*, and the $v_i$'s are the corresponding *values*. The set of all the attribute-value pairs of an object's state is formed by repeated application of the binary union operator $\_ , \_$ which also obeys structural laws of associativity, commutativity, and identity; i.e., the order of the attribute-value pairs of an object is immaterial.

Particular systems are axiomatized by providing additional operations and equations, specifying, for example, the data operations on attribute values and the structure of messages, and by providing rewrite rules describing the system dynamics. These rules have the form

$$\begin{aligned} r(\bar{x}) \ : \quad & M_1 \ldots M_n \ \langle O_1 : F_1 \mid atts_1 \rangle \ldots \langle O_m : F_m \mid atts_m \rangle \\ \longrightarrow \ & \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\ & \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\ & M'_1 \ldots M'_q \\ & \mathit{if} \ \ \mathit{cond} \end{aligned}$$

where $r$ is the rule's label, $\bar{x}$ is a list of the variables occurring in the rule, the $M$s are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and $C$ is the rule's condition. That is, a number of objects and messages can come together and participate in a transition in which some new objects may be created, others may be destroyed, and others can change their state, and where some new messages may be created. We require that there be at least one object on the left hand side.

The Full Maude language [4,9] provides special syntax, *object-oriented* modules, to specify concurrent object-oriented systems. They are declared with the syntax `omod ... endom`. Within an object module, classes can be declared specifying attributes and their sorts, and subclass relations can be declared to extend an attribute set. In addition attributes that are not used can be omitted from the object terms in rules. Modules can be *parameterized* by parameter *theories*. The instantiation of a parameterized module is accomplished by *views* that map the sorts and operators of parameter theories to their corresponding instantiations in the module chosen as the actual parameter. Views are declared with the syntax `view ... endv`. View names can then be placed in the corresponding parameters of a parameterized module to instantiate the module. Maude provides several built in modules and theories, including the one sort parameter theory `TRIV`, and modules, `BOOL` for booleans with sort `Bool`, and `MACHINE-INT` for integers, with sort `MachineInt`. In this paper all of the views we use have the form `view <Sort> from TRIV to <MODULE> is sort Elt to <Sort> endv`. Indicating that the sort `Elt` of the module `TRIV` is mapped to the sort `<Sort>` of module `<MODULE>`. We leave these implicit in our discussion of the Maude specification.

# 3 The Network Model

A network as modeled in Maude consists of a set of nodes, interfaces, and local area networks (lans), each being a Maude object of the appropriate class. Since failure, and the ability to recover from such failure is central to network protocols, each type of object: node, lan, and interface has a boolean flag to indicate its status (up or down). An interface connects one node to one lan and has two buffers, one for outgoing packets (node to lan) and one for incoming packets (lan to node). A node also has a pair of buffers for each of its interfaces. These can be thought of as the other half of the interfaces buffers. This split is done to allow uniform and distributed movement of packets from node to interface to lan and back. It also means that node specific rules for processing packets can be expressed completely locally – using only the nodes local state. There are generic network rules for moving packets around and for modeling failure/crash of nodes, interfaces, and lans, and for reset of interfaces and lans. Node reset is specific to the node subclass.

## *3.1 Packets*

A network packet has at least a destination address, a packet type, contents, and a source address. The destination address could be that of an interface, or a pseudo address such as a multicast address. The constant `lanAddress` is introduced to serve as a lan broadcast address.

```
(fmod PACKET is
  including CONFIGURATION .
  sorts Packet Contents PType Address .
  subsort Oid < Address .
  op mkPacket : Address PType Contents Address -> Packet .
  op lanAddress : -> Address .
endfm)
```

## *3.2 Interface buffers and plugs*

We use two additional structures to assist in representing nodes. These structures are needed to represent interface connections from a nodes point of view (`InterfaceBuffer`) and from a lans point of view (`InterfacePlug`). An `InterfaceBuffer` is the pair of buffers together with the name (identifier) of the interface. The buffers are modeled as lists of packets using the parameterized `LIST` module instantiated with the `Packet` view. As mentioned above, all our views have the same form and are omitted here for brevity. The `InterfacePlug` is the name of the interface together with a flag. The flag in this case is used in the rules to indicate whether or not a packet on the lan wire has been seen by an interface. The module also contains operations `clear` that clear, or reset, these structures. Clearing an `InterfaceBuffer` resets each of its buffers to the empty one, while clearing an `InterfacePlug` resets its flag to `false`. For brevity we omit the equations describing `clear`.

```
(fmod  INTERFACES is
  including CONFIGURATION .   including BOOL .  including LIST[Packet] .
  sorts InterfaceBuffer InterfacePlug .

  op mkInterfaceBuffer : Oid  List[Packet]  List[Packet] -> InterfaceBuffer .
  op mkInterfacePlug : Oid  Bool  -> InterfacePlug .

  op clear : InterfaceBuffer -> InterfaceBuffer .
  op clear : InterfacePlug -> InterfacePlug .
  **** equations for clear omitted.
endfm)
```

### *3.3 Networks*

Then the `NETWORK` object module is introduced which defines the classes `Node`, `Interface`, and `Lan` and the generic operations and rules for networks. An `Interface` object consists of its status flag, a lan object identifier, a node object identifier, and the two unidirectional buffers. A `Node` object consists of its status flag, and a list of `InterfaceBuffer`. A `Lan` object consists of its status flag, a wire that may or may not contain a packet (the one currently traveling on the lan), and a list of interfaces that are attached to the lan. This is in the form of a list of `InterfacePlug`s. The `DEFAULT` parameterized module its parameter sort with a distinguished constant to represent an undefined or non-existent element.

```
(omod  NETWORK is
  protecting BOOL .  protecting MACHINE-INT .
  protecting INTERFACES .  including PACKET .
  protecting LIST[Packet] .  protecting DEFAULT[Packet] .
  protecting LIST[InterfaceBuffer] .  protecting LIST[InterfacePlug] .

  class Interface | up : Bool, lan : Oid, node : Oid,
                    n2l : List[Packet], l2n : List[Packet] .

  class Node | up : Bool, interfaces : List[InterfaceBuffer] .

  class Lan | up : Bool, wire : Default[Packet],
              interfaces : List[InterfacePlug] .
```

The object module also contains four operations necessary for formulating the rewrite rules. The operation `iCast` constructs a packet addressed to the lan broadcast address `lanAddress` of the given type and contents on the outgoing buffer of each of the given interfaces with packet source the particular interface associated with the buffer. `clear` is the extension of the reset operations mentioned above to lists of `InterfacePlug`s and lists `InterfaceBuffer`s respectively. Finally, the predicate `allTrue` is used to determine if every `InterfacePlug` in a list has seen the packet currently on the lan's wire. In other words if every `InterfacePlug` flag is set to `true`.

```
  op iCast : PType Contents List[InterfaceBuffer] -> List[InterfaceBuffer] .

  op clear : List[InterfaceBuffer] ->  List[InterfaceBuffer] .
  op clear : List[InterfacePlug] ->  List[InterfacePlug] .

  op allTrue :  List[InterfacePlug] ->  Bool .

*** equations and conditional equations omitted for brevity
```

The rewrite rules form several groups. There are rules that allow any interface `icrash`, node `ncrash` or lan object to crash `lcrash`, and there are rules that allow any interface `ireset` or lan object `lreset` to recover (reset) from a crash. The rules for the crashing of an interface or lan are generic, as are their reset rules. The crash rules set the `up` attribute to `false`, while the reset rules set it to `true`. In addition, reset rules give the other attributes default initial values clearing wires and buffers. The node rules however will be part of a particular protocol. Thus there will be specific reset/reboot rules for each subclass of node, representing the protocol at that node.

Independent of the protocol are the rules for propagating packets around the network. Examples of this are rules that move a packet from the appropriate interface to the corresponding buffer on the node `i2n` and `n2i`, a similar transfer going in the other direction. Packet flows from the interface to the lan via `i2l`, and from the wire of the lan to the interfaces on the lan via a pair of rules. `l2i-on` puts the packet in the appropriate interface buffer when the interface is up (not removing the packet from the wire). It ignores interfaces that are not up, though it does record the fact that they have been considered for this packet (by using the flag of the `InterfacePlug`, and finally a rule, `clear`, to remove the packet from the lan wire after its has been propagated to every up interface, and dropped for every down interface on the lan.

### *3.4 Time and Timers*

To handle the use of timers uniformly a class of timer objects is defined and a timerset attribute is added to nodes that contains the nodes current set of timers. First `TIME` and `TIMER` modules are introduced. For simplicity and executability, `Time` is modeled by `MachineInt`:

```
(fth TIME is
 protecting BOOL .  protecting MACHINE-INT .
 sort Time .
 subsort MachineInt < Time .
endfth)
```

Note that, following [16], all that is required of a `TIME` module is a sort

`Time`, a constant `0`, and operations `+`, `-`, and `<` satisfying certain basic properties, and it is straightforward make this parametric.

A timer is an object with a boolean attribute `on` that indicates whether the timer is running, and a attribute `time` which when running gives the time left. Passage of time is modeled by the `delta` operation that decrements the `time` attribute of a timer that is on.

```
(omod TIMER is
  including BOOL .   including TIME .
  class Timer |  on : Bool, time : Time  .
  op delta : Object Time -> Object .
  var xid : Oid .   vars xTime0 xTime1 : Time .
  eq  delta(< xid : Timer | on : true, time : xTime0 >, xTime1) =
           < xid : Timer | on : true,
              time : (if (xTime0 < xTime1) then 0 else xTime0 - xTime1) > .
  eq  delta(< xid : Timer | on : false, time : xTime0 >, xTime1) =
           < xid : Timer | on : false, time : xTime0 > .
endom)
```

### *3.5 Timed Networks*

The `TIMED-NETWORK` module introduces `TimedNode`s which extend the `Node` class by incorporating a clock `clock` representing the current local time, and a set of timers `timerSet`. We also incorporate into the `TIMED-NETWORK` module an ability to create *new* identifiers and new timers by the operations `newId` and `newTimer`. The attribute `idcounter` is used to create a fresh identifier for each new timer object.

```
(omod TIMED-NETWORK is
  including NETWORK .  including TIMER .  including SET[Object] .

  class TimedNode |  idcounter : MachineInt,
                     clock : Time,
                     timerSet : Set[Object] .
  subclass TimedNode < Node .

  op newId : Oid MachineInt -> Oid .
  op newTimer : Oid MachineInt Bool Time -> Object .
```

A `System` sort in a `TIMED-NETWORK` module is a wrapper around a `Configuration`. Following the approach of [16], the passage of time is given by lifting the `delta` operation on timer objects, to corresponding operations on sets of objects, and `Configurations`.

Finally the predicate `noTimeOuts` is introduced. `noTimeOuts` is true of a configuration, or set of objects, if there is no running timer whose time has run out (i.e. reached 0).

```
  sort System .
  op mkSys : Configuration -> System .
  op delta : Set[Object] Time -> Set[Object] .
  op delta : Configuration Time -> Configuration .
  op noTimeOuts : Configuration -> Bool .
```

```
  op noTimeOuts : Set[Object] -> Bool .
  **** equations concerning operations omitted.
```

## 4 The PIMDM Hello example

The neighbors of a network node are those nodes that are connected to one of the lan's to which the given node is connected. Since nodes may go offline, and new nodes may connect, this is a dynamically changing set. The Hello sub-protocol of the PIM-DM protocol is used to determine the current neighbor nodes. Each node periodically broadcasts a `hello` message on each lan to which it is connected. When a `hello` message is received, the sender is added to the list of neighbors and an associated timer is set. If the timer times out before another `hello` message is received the neigbor is assumed to be disconnected and is removed from the neighbors list. If the timeout interval and the resend interval are suitably tuned to the network conditions, then after some initial startup period each node will have a reasonably good approximation of its set of neighbors.

The `HELLO` object module introduces the `Hello-Node` class and gives the rules for processing `hello` messages and timeouts. A `Hello-Node` extends the `TimedNode` class by incorporating several collections of timer objects. Each `Hello-Node` has a single `helloTimer`, as well as a timer for each of its neighbor nodes on the network. The timers being objects, are usually referred to by their identifiers. Thus the `helloTimer` attribute is actually an object identifier. The neighbor timers, or more accurately their identifiers, are kept in a list, while the identifiers of the neighbor nodes are kept in a separate list. The association of timer (identifier) with neighbor node is by the position in the list. Thus the $i$th timer in the timer list corresponds to the $i$ neighbor node in the node list. The node stores all the actual timer objects in the `timerSet` attribute provided by the `TimedNode` superclass.

The protocol makes use of `hello` packets, and two time constants `helloTimeOut`, and `helloTime`. The `helloTimeOut` is the time a node waits before it decides to remove it from its current neighbor list, while the `helloTime` is the time between its multicasting of `hello` packets. Thus if `helloTimeOut` is substantially larger than `helloTime` a node can safely assume that it is justified in removing a neighbor when the associated timer goes off.

```
(omod HELLO is
  including TIMED-NETWORK .  including LIST[Oid] .  including SET[Object] .

  class Hello-Node | helloTimer : Oid,
                     neighborTimers : List[Oid], neighbors : List[Oid]  .

  subclass Hello-Node < TimedNode .
  subsort Oid < Address  .  subsort Time < Contents .
  op hello : -> PType .
  op helloTimeOut : -> Time .  eq helloTimeOut = 105 .
  op helloTime : -> Time .  eq helloTime = 30 .
```

### *4.1 The* `Hello-Node` *reset*

The first important rule is the reset rule for `Hello-Node`, a reset for a hello node does three things: (1) the node clears both it's neighbor lists (i.e sets both `neighborTimers` and `neighbors` to be empty); (2) it throws out any neighbor timers, leaving only its `helloTimer`, which is reset to `helloTime`; (3) it then broadcasts a `hello` packet with contents `helloTimeOut` on all its interfaces, using the `iCast` operation.

```
rl [nreset] :
   < xid : Hello-Node | up : false, interfaces : xNIL, helloTimer : xidHT,
                       neighborTimers : xNTL, neighbors : xNL,
                timerSet : xTS U < xidHT : Timer | on : xBool, time : xT> >
  xConf
   =>
   < xid : Hello-Node | up : true,  helloTimer : xidHT ,
                       interfaces : iCast(hello, helloTimeOut, xNIL) ,
                       neighborTimers : nil, neighbors : nil,
             timerSet : < xidHT : Timer | on : true, time : helloTime > >
  xConf  .
```

### *4.2 The incoming* `hello` *packet rule*

If a `Hello-Node` receives an incoming `hello` message on one of its interfaces, it does one of two things: (1) if it already has a timer associated with the neighbor, then it simply resets that timer to the contents of the `hello`, which is the value of `helloTimeOut` for the sender; (2) if it doen't have a timer, then it creates a new timer associated with this neighbor (via the two lists `neighborTimers` and `neighbors`) and adds this timer to the timer set, incrementing the `idcounter` appropriately.

This complex rule makes use of three operations `HelloOnNeighbors`, `HelloOnTimers`, and `HelloOnTimerSets` which we omit due to space considerations.

```
rl [nhello] :
  < xid : Hello-Node | up : true, idcounter : k, helloTimer : xidHT,
    interfaces : xNIL1 @ mkInterfaceBuffer(
                         xidN,
                          mkPacket(lanAddress,hello,xHelloTime,xidI) @ xinQ,
                          xoutQ) @ xNIL2,
       neighborTimers : xNTL, neighbors : xNL, timerSet : xTS >
  xConf
   =>
  < xid : Hello-Node | up : true,
      idcounter :  if ( length(xNL) == length(HelloOnNeighbors(xidI,xNL)) )
                     then k else ( k + 1 ) fi,
      interfaces : xNIL1 @ mkInterfaceBuffer(xidN, xinQ  ,xoutQ) @ xNIL2 ,
      helloTimer : xidHT,
      neighborTimers : HelloOnTimers(xidI,xNL,xNTL,xid,k) ,
      neighbors : HelloOnNeighbors(xidI,xNL) ,
      timerSet : HelloOnTimerSet(xidI,xHelloTime,xNL,xNTL,xid,k,xTS)  >
  xConf .
```

### *4.3 The Timeout rules*

The remaining two rules concerning the protocol correspond to when either of the two types of timers timeout. When the `HelloTimer` times out the node merely resets the `helloTimer` (to `helloTime`), and then broadcasts a `hello` packet with contents `helloTimeOut` on all its interfaces, using the `iCast` operation.

```
 rl [nhelloTimeOut] :
    < xid : Hello-Node | up : true, helloTimer : xidT, interfaces : xNIL,
                         neighborTimers : xNTL, neighbors : xNL,
            timerSet : xTS U  < xidT : Timer | on : true, time : 0 > >
   xConf
    =>
    < xid : Hello-Node | up : true,  helloTimer : xidT ,
                         interfaces : iCast(hello,helloTimeOut,xNIL) ,
                         neighborTimers : xNTL,  neighbors : xNL,
           timerSet : xTS U  <xidT : Timer | on : true, time : helloTime > >
   xConf .
```

If a neighbor timer times out, then that neigbour is flushed from both neighbor lists, as is its timer from the timer set.

```
crl [nNeighborTimeout] :
    < xid : Hello-Node | up : true,  helloTimer : xidT, interfaces : xNIL,
                         neighborTimers : xNTLa @ xidNT @ xNTLd,
                         neighbors : xNLa @ xidN @ xNLd,
            timerSet : xTS U < xidNT : Timer | on : true, time : 0 > >
   xConf
    =>
    < xid : Hello-Node | up : true, interfaces : xNIL, helloTimer : xidT ,
                         neighborTimers : xNTLa @ xNTLd,
                         neighbors : xNLa @ xNLd,
                         timerSet : xTS  >
   xConf
    if not( < xidNT : Timer | on : true, time : 0 > in xTS) .
```

### *4.4 The passage of time*

Finally the rules for the passage of time are added. The `ctick` rule is conditional on there being no timeouts in the system. Thus time outs must be acted upon before time can be allowed to pass. The `tick` rule allows time to pass unconditionally. Application of either of these rules must be controlled by network simulation strategies that provide a value for the time variable `xTime`.

```
crl [ctick] : mkSys(xConf) => mkSys(delta(xConf,xTime)) if noTimeOuts(xConf) .

rl [tick] : mkSys(xConf) => mkSys(delta(xConf,xTime))  .
```

## 5 Network Simulation Strategy Primitives

The simplest way to test a network protocol is to define the initial configurations of interest (network topology and initial protocol parameters) and execute these using Maude's default execution strategy. This already gives some information about the protocol and particularly about your specification. However, this does not account for all the other possible executions due to different choices in which rewrite rule to apply at each given stage. By choosing different execution strategy's for Maude one can model a range of different network conditions and rates that different nodes might progress without changing the underlying specification.

Using Maude's reflective capability, such strategies can be readily defined and executed in Maude. As a first step in developing a simulation tool, we have defined a small set of primitives for simulation of networks based on our `Timed-Network` model. The basic problem is to figure out what and how much to do between applications of the tick rule that advances time. Our primitives have been designed to make this decision programmable and allow the designer to experiment with different choices.

We extend Maude's META-LEVEL module, a functional module that provides syntax and operations for representation and manipulation of modules, terms, and sort relations, as well as basic rewriting primitives: meta-reduce (for equational rewriting); meta-apply (for controlled application of specific rewrite rules); and meta-rewrite (for using the default execution strategy).

In object configurations we are interested in controlling the application of rewrite rules based on the objects involved in order insure that each object is allocated a fair share of *cycles*. We define the function

```
op cast : Term -> Set[Term] .
```

which given the representation of an object configuration returns the set of (representations of) object identifiers of that configuration. To control the number of rewrites for a given object we define a sort `OwG` (Object with Gas) and constructor

```
op wGas : Term MachineInt -> OwG .
```

where `wGas(oidT,nGas)` indicates that the object with identity given by `oidT` has gas allocation `nGas` (can participate in `nGas` rule applications.)

We restrict attention to object modules in which each rule involves at least one object and consider only configurations in which each object has a unique identity. Thus the identities of the objects involved uniquely determine the matching substitution for a rule application. In the case of our network model there is, for each rule, a principle object that determines the match, even though some rules involve more than one object (interface-lan, and interface-node rules). Thus we introduce the sort `RuleScheme` consisting of pairs `rs(rid,vid)` where `rid` is a rule identifier and `vid` is the variable appearing in the rule that names the principle object. A rule instance is a pair `ri(rs(rid,vid),oidT)` consisting of a rule scheme and an object id (representation) giving the match for vid.

The operations Enabled and Fire are the basic operations for defining strategies for object configurations.

```
op Enabled : Module Term Term RuleScheme -> Bool .
op Fire : Module Term Term RuleScheme -> Term .
```

Enabled(M, cfT, oidT, rs(rid, vid) ) tests whether, in the module M, the rule instance ri(rs(rid,vid),oidT) is enabled in the configuration cfT. If so, Fire(M, cfT, oidT, rs(rid, vid) ) is the result of firing (applying) the above rule instance. (Note that enableness and firing are also the primitive notions used to define notions of fairness in such object systems.)

We use a sort Path to represent information of interest extracted from a Maude computation. For example path(steps,cfT) is a path where steps is the sequence of rule instances applied and cfT is the final configuration. This form of path is just one possible choice. More information could be kept, the final configuration could be further abstracted, and so on according what is most useful.

Using these basic primitives we define a simple strategy for bounded application of rules.

```
op tryObs : Module List[RuleScheme] Set[OwG] Set[OwG] Bool Path -> Path .
```

Working in module M with initial configuratin cfT, tryObs(M, rsL, OwGs, mt, false, path(nil,cfT)) applies the rule schemes in the list rsL decrementing allotted gas for the principle object of each rule application and remembering the sequence of rule instances, until nothing more can be done — either no more rules are enabled, or all of the objects are out of gas. This is done by repeatedly mapping through the set of objects, OwGs, looking for a rule enabled for that object and applying it if one is found. The second set of objects-with-gas is used to keep those that still have gas left and the boolean is used to remember if any rule has fired. The path argument is used to remember the rule instances fired and the current configuration.

This strategy can be used for any object module meeting our requirment that each rule involves at least one object and each rule has a principle object. The notion of rule scheme can easily be generalised to allow for a list of variables to match ids in the case that there are several objects with no one of them uniquely determining a matching substitution. We have used this strategy to run to run test simulations of the HELLO protocol.

In a timed object module, the tryObs strategy deals with what happens between ticks. To control the passing of time and the interleaving of ticks and other rules we define Tick, that applies the tick rule once, round that applies the tryObs strategy and then the Tick strategy, and finally rounds the iterates round some finite number of times.

```
op Tick : Module RuleScheme MachineInt Path -> Path .
op round : Module List[RuleScheme] NzMachineInt RuleScheme MachineInt Path
           -> Path .
op rounds :
  Module List[RuleScheme] NzMachineInt RuleScheme MachineInt MachineInt Path
```

```
-> Path .
```

Tick(M,rs(rid,vid),nTime, path(steps, cfT)) is the result of applying the tick rule, rid, of timed module M with nTime as the value of the time variable, vid. round(M, rsL, nGas, rs(rid,vid), nTime, path(steps,cfT)) first applies tryObs with rule-scheme list rsL and initial object-with-gas set obtained by computing the cast of cfT and allocating each object initial amount of gas, nGas. It then applies Tick to the resulting path, with tick rule rs(rid,vid) and time increment nTime. rounds(M, rsL, nGas, rs(rid,vid), nTime, nRounds, path(steps,cfT)) repeats round, nRounds times and returns the resulting path. This is a very simple strategy, but effective for simulating network configurations as we will indicate in the next section. Again it can easily be elaborated with mechanisms for assigning different amounts of gas to different classes of objects, or for prioritizing the list of rules.

As a first simple example, consider a network with one lan and three nodes connected to it, each by a single interface. Let rsL be the list of generic node rule schemes (omitting crash and reset) appended to the list of hello rule schemes. Then with nGas 5 and nTime 10 each node knows its neighbors after 2 rounds, but with nGas 2 it takes three rounds for this to be accomplished.

To allow for runtime adjustment of the timer settings we defined a function setHT such that setHT(M, sendT, holdT) adds equations defining the constants of a HELLO module representing the interval to wait before sending a hello message to be sendT, and defining the time to wait for a next hello from a given neighbor to be holdT.

Now we can experiment with different choices of initial configuration, timer settings, amount of work between ticks, and and speed (all times being relative to a choice of unit). The function

```
harounds(sendT,holdT, nGas,nTime,nRounds, hcfT)
```

starts with initial configuration hcfT of Hello nodes, interfaces and lans, working in a HELLO module, setHT(HELLO-TEST,sendT,holdT), with timer settings given by sendT and holdT. It executes for nRounds, allowing each entity nGas rewrites and advancing the clock by nTime after each round. (In fact the round function was modified to fire enabled rule instances dealing with timeouts until none remain, not counting these in the gas allotment, then considering the remaining rules as described above.)

To test the ideas, we considered three inital configurations, looking for situations in which a neighbor timer expires, and for situations in which a node does get to know all of its neighbors in the time of the run. The three configurations are:

- Hnet3 — the 3 node network described above
- Hnet6 — a network with 6 Hello nodes, one lan, with each node having one interface connection to the lan.

- `Hnet33` — a network with 5 Hello nodes and 2 lans. 3 nodes are connected to each lan with one of the nodes be connected to both lans.

We fixed the send interval **`sendT`** to be 10 and considered values of **`holdT`** ranging from 10 to 20. (The PIM-DM RFC proposes 30 and 105 respectively. The longer holdtime makes no difference in the absence of other network traffic.)

In the `Hnet3` case the following experiments were performed:

| | sendT | holdT | nGas | nTime | nRounds | NTO? | Know? |
|---|---|---|---|---|---|---|---|
| 1. | 10 | 15 | 2 | 2 | 15 | no | yes |
| 2. | 10 | 11 | 2 | 2 | 10 | no | yes |
| 3. | 10 | 15 | 1 | 2 | 10 | no | yes |
| 4. | 10 | 15 | 5 | 2 | 15 | no | yes |
| 5. | 10 | 10 | 2 | 2 | 10 | yes | yes |

The first five columns are labeled by the parameters to the run, the `NTO?` column records whether or not there were neighbor timeouts, and the `Know?` column records whether or not all Hello nodes came to know their neighbors. In each of these runs, every Hello node came to know its 2 neighbors and only in the **`holdT`** 10 case did a neighbor timeout occur.

In the `Hnet33` case the following experiments were performed:

| | sendT | holdT | nGas | nTime | nRounds | NTO? | Know? |
|---|---|---|---|---|---|---|---|
| 1. | 10 | 15 | 2 | 2 | 15 | no | yes |
| 2. | 10 | 15 | 6 | 3 | 15 | no | yes |
| 3. | 10 | 15 | 1 | 3 | 15 | yes | no |

In the first two runs, every Hello-node comes to know its neighbors (two for the nodes connected to one lan and 4 for the node connected to two lans) and no neighbor timeouts occur. In the last run (with 1 unit of gas per round, corresponding to a slow network) neighbors become known then timeout due to the slow transmission and processing of packets. Nodes do not come to know all their neighbors at once.

In the `Hnet6` case the following experiments were performed:

| | sendT | holdT | nGas | nTime | nRounds | NTO? | Know? |
|---|---|---|---|---|---|---|---|
| 1. | 10 | 20 | 5 | 5 | 10 | no | no |
| 2. | 10 | 15 | 2 | 2 | 10 | no | no |
| 3. | 10 | 15 | 2 | 2 | 20 | no | no |
| 4. | 10 | 15 | 1 | 2 | 20 | no | no |
| 5. | 10 | 15 | 6 | 2 | 20 | no | yes |
| 6. | 10 | 12 | 6 | 4 | 10 | yes | no |

This is perhaps the most interesting case. Giving the timing granularity, six nodes on a lan causes congestion and it requires faster processing for the Hello protocol to stabilize. In the first three cases there are no neighbor timeouts and only one node gets to know all five of its neighbors at once. The rest come to know at most four at once. In the fourth case, again there are no neighbor timeouts, but nodes come to know fewer neighbors at once. In the fifth case all nodes come to know all their neighbors and there are no neighbor timeouts. In the sixth case, there are neighbor timesouts, but there are also times when all nodes know all their neighbors.

The above experiments illustrate how our simulation primitives can be used to study the relation between timer settings, network/processor speed, and network load. Much work work remains, to refine the primitives and ensure that the abstractions provide an adequate model of the network properties of interest. For example, refining the way gas is assigned would allow us to control the lan, interface, and node speeds independently. However, the initial results are encouraging, in that starting with the basic simulation primitives it is fairly easy to modify them and to carry out the simulations. The 20 round simulations of the 6 node network were the most complex and these took at most a few minutes to run.

The Maude code for the timed network and strategy modules will be made available on the Maude website (`http://maude.csl.sri.com`) soon, along with the HELLO protocol, example runs and additional PIM-DM protocol modules as they are developed.

## 6 Future Work

The work presented in this paper is the starting point for a number of research directions. The first task is to finish the specification of the PIM-DM protocol, and carry out a variety of analyses. This was the original motivating example. There are two interesting challenges here. One is to make clear, in some systematic way, the connection between the Maude specification and both the pseudo-code specification [21] that was our starting point, and the IETF RFC for PIM-DM, which was the starting point of the pseudo-code specification. This is important both as a means of validating the Maude specification, and as means of making the Maude specification understandable to protocol designers and implementors that are more familiar with these informal specifications. It is also important because the informal specification in the form of RFCs captures intuitions and requirements that are not so easy to formalize, but are essential to the correct understanding. A second challenge is to factor the full PIM-DM specification into sub-protocols, each dealing with a particular sub-task (the HELLO protocol being an easy example, but there are others for pruning the multicast tree and for grafting new subtrees, for example). The idea being to be able to analyze and test each sub-protocol independently, as well as to compose and analyze the composite. One benefit of this approach is that sub-protocols can then be reused elsewhere or replaced by alternative with greater ease and assurance. Another potential benefit is to be able to use results from analysis of the sub-protocols to simplify analysis and testing of composites. A similar case study specifying an adaptive multicast congestion control protocol as the composition of reusable sub-protocols is currently in progress [14].

Going beyond PIM-DM we need to look at other protocols, dealing with different aspects of network communication – for example group communication, security and fault-tolerance.

Another direction of research is to develop a richer collection of simulation strategies and analyses. Going beyond the basic timed network model we need to consider how to deal with probablistic issues both from the point of view of specification of properties and for purposes of more comprehensive simulation strategies. A first step is to investigate the use of the Real-Time Maude tool [17], for model checking of simple temporal properties.

Protocols in a particular domain, for example security or routing, are often specified with a style or domain specific notation that is quite systematic. Millen has made good use of this observation in the case of cryptographic communication protocols and formalized the notation in the language CAPSL [8]. The formalization makes implicit assumptions explicit and allows for execution and formal analysis of protocols specified in CAPSL. An interesting question is whether notations can be found for other domains that are easy for protocol designers to use but at the same time can be given a formal executable semantics.

Finally, an important challenge is to abstract from specific case studies involving composition of protocols and develop generic composition mechanisms that can be expressed as module operations in Maude.

**Acknowledgements.**

The authors would like to thank the Maude group and especially José Meseguer for many fruitful discussions and Brad Smith for helping us understand the PIM-DM protocol and kindly providing his pseudo-code. We also thank the anonymous referees for their helpful suggestions for improvement. This research was partially supported by DARPA/NASA NAS2-98073. ARPA/SRI subcontract 17-000042, NSF CCR-9900326, and ONR N00012-99-C-0198.

## References

[1] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations, August 2000. To appear in: International Symposium on Software Testing and Analysis.

[2] K. Bhargavan, C. A. Gunter, and D. Obradovic. RIP in SPIN/HOL, August 2000. To appear in: Theorem Provers for Higher-Order Logics.

[3] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000. Superceeds USC tech report 99-702b.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic, 1998. URL: `http://maude.csl.sri.com`.

[5] A. J. H. David L. Dill, Andreas J. Drexler and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[6] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, A. Helmy, D. Meyer, and L. Wei. Protocol independent multicast version 2 dense mode specification, July 1999. Internet Draft: draft-ietf-pim-v2-dm-03.txt, url: `http://www.ietf.org/ietf/1id-abstracts.txt`.

[7] G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*. IEEE, 2000. to appear.

[8] G. Denker and J. Millen. CAPSL intermediate language. In N. Heintze and E. Clarke, editors, *Proc. of Workshop on Formal Methods and Security Protocols, July 1999, Trento, Italy*, 1999. URL:`www.cs.bell-labs.com/who/nch/fmsp99/program.html`.

[9] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Malaga, 1999.

[10] M. Gordon and T. e. Melham. *Introduction to HOL: a theorem-proving environment for higher-order logic*. Cambridge University Press, 1993.

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

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

[13] ISI. Ns website: `http://www.isi.edu/nsnam/ns/`, 2000.

[14] M. Keaton, J. Meseguer, P. Ölveczky, and C. Talcott. Specification and analysis of the tasc aer/nca protocol in maude. work in progress, 2000.

[15] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*. The MIT Press, 1993.

[16] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. Submitted for publication, 1999.

[17] P. C. Ölveczky and J. Meseguer. Real-time maude: A tool for simulating and analyzing real-time and hybrid systems. In *3rd International Workshop on Rewriting Logic and its Applications, WRLA'00*, 2000. to appear.

[18] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction–CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.

[19] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[20] A. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *IEEE Symposium on Foundations of Secure Systems*, 1995.

[21] B. Smith. Pseudo-code for pim-dm, Oct. 1999. personal communication.