Specifying and Checking Fault-tolerant Agent-based Protocols using Maude

Jeffrey Van Baalen[1], James L. Caldwell[1], and Shivakant Mishra[2]

[1] Department of Computer Science, University of Wyoming, Laramie, WY 82071
[2] Department of Computer Science, University of Colorado, Boulder, CO 80309

1 Introduction

Fault tolerance is an important issue in mobile, agent-based computing systems. However, most research in this area has focused on security and mobility issues. The DaAgent (Dependable Agent) system [8] is similar to several other agent-based computing systems including Agent-Tcl [6], Messengers [5], and Ajanta [9]. However, unlike these systems DaAgent is being designed to address fault tolerance issues. Within the DaAgent system several fault tolerant protocols are being investigated. These protocols have been specified in natural language, English prose, and a Java implementation within the DaAgent system is being tested. This approach has proved to be extremely time-consuming and inflexible, for example, it is difficult to rapidly change test conditions and fault injection is extremely primitive involving physically halting or reseting a machine, pulling network connections, or sending a kill message to the agent process. Testing the implementation of course serves as a weak form of evidence of correctness, but offers little real assurance of the correctness of the system.

In order to address these problems we are taking the approach of formalizing the protocol specifications and debugging them at the specification level. We are using the Maude executable specification language [2] for this purpose. We are taking this approach knowing that a majority of defects in systems can be traced back to specification errors. Also, much data supports the fact that errors identified early in the development process are dramatically less expensive to correct.

However, while simply specifying systems formally often identifies errors and ambiguities early, formal specifications can also contain errors. We have chosen to use an executable specification language so that we can debug our specifications without necessarily resorting to formal proofs of correctness or massive testing efforts. We have found that using Maude to specify our protocols has a number of benefits:

1. The Maude system has clean extendable syntax so that it is straightforward to create domain specific languages with appropriate abstractions that have a clearly defined semantics. This is an often undervalued feature that is very useful in a specification language, making it more understandable for the intended user.

2. The resulting specifications are executable and, in fact, the Maude engine is very fast (up to 1.3 million rewrites per second on a Pentium II). This enables us to construct a variety of initial states and explore very extensively the execution of the protocol from those states.
3. Maude is reflective. This enables us to separate a specification from the execution of that specification. This has proved useful in three ways. First, it has enabled different explorations from different initial states without changing our protocol specifications. Second, it has enabled a more sophisticated model-checking analysis in which all behaviors from some initial state are explored (up to some depth) via one of many possible search strategies (e.g., breadth first). Third, it has enabled us to specify fault models for our protocols separately from the protocol specifications. This has enabled the testing of the protocols, while injecting different types of faults.

The major contribution of this work is the separate specification of fault models at the meta-level enabling testing of protocols under conditions where faults are injected. Also, the fault injection technique can be combined with the search strategies, enabling the exhaustive testing of a protocol, injecting faults in all possible ways.

2 Rewriting Logic

The Maude language has a declarative semantics based on rewriting logic [2, 7]. Rewriting logic extends algebraic specification techniques to concurrent and reactive systems by providing a natural model of concurrency in which local concurrent transitions are specified as rewrite rules. The model is very flexible allowing both synchronous and asynchronous models of concurrency as well as a wide range of concurrent object systems. The flexibility of rewriting logic has enabled Maude's use for formalizing many different kinds of systems (see [4] for descriptions of a few).

In rewriting logic the state space of a concurrent system is formally specified as an algebraic data type by means of an equational specification consisting of a signature and a set of conditional equations. The equations in the specification of the system state define equivalence classes of terms over the signature. Concurrent transitions are specified as rewrite rules of the form $t \Rightarrow t'$, where t and t' are terms in the signature (normally) containing variables. Rules describe concurrent transitions because they specify how a system state matching t can change to a system state where the term matching t is replaced by t' (appropriately instantiated).

More formally (following [2]), a theory in rewriting logic is a pair $\mathcal{R} = ((\Omega, \Gamma), R)$, where (Ω, Γ) is an equational specification with signature Ω and equational axioms Γ. The equations of Γ define the algebraic structure of the theory, i.e., they define equivalence classes on terms with signature Ω. R is a collection of labelled rewrite rules that specify concurrent transitions that can occur in the system axiomatized by $\mathcal{R}$. The rules in R are applied *modulo* the equations in Γ.

The state space of a concurrent object system can be specified as an algebra by means of an equational theory (Ω, Γ). The concurrent state of such a system, often called a *configuration*, usually has the structure of a *multi-set* (an unordered collection allowing duplicates) containing objects and messages. Therefore, we can view configurations as built up by a binary multi-set union operator, which can be represented with empty syntax (i.e., juxtaposition) as $\_\ \_ : \mathbf{Conf} \times \mathbf{Conf} \Rightarrow \mathbf{Conf}$. (Following the conventions of mix-fix notation, under-bars indicate argument positions.) The multi-set union operator $\_\ \_$ is declared to satisfy the laws of associativity and commutativity and to have identity $\emptyset$. Objects and messages can then be specified as elements of sorts **Object** and **Msg** respectively and these sorts can be given as subsorts of **Conf** so that a single object or message is a singleton configuration.

An *object* 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, C is its class, the a_i's are the objects *attribute identifiers*, and the v_i's are the corresponding attribute *values*. The set of an object's attribute-values pairs is formed by application of the binary operator $\_,\_$ which is associative and commutative, so that the order of the attributes of an object is immaterial.

A message is also represented as a term. So, for example, given an operator **from_to_hello** : **Oid** × **Oid** → **Msg** and two Oids (object identifiers) **A** and **B**, the term **(from A to B hello)** is a message.

Rules specify local transitions by describing how one configuration can be transformed into another. In a concurrent object system, a rule might transform a configuration containing an object and a message for that object into a configuration in which the message has been removed and the object has been updated to reflect receipt of the message. Such a rule would have roughly the form:

```
rl[name] :
  ((from A to D some-message) ⟨ D : Agent | some-attributes ⟩ Conf)
  ⇒ (⟨ D : Agent | updated-attributes ⟩ Conf) .
```

where A and D are variables of sort f **Oid** and **Conf** is a variable of sort Configuration. The left hand side of this rule matches any configuration containing an object named D and a message to that object. The variable Conf is bound to the other objects and messages in the matched configuration. The right hand side of the rule specifies the transformed configuration which differs from the original because the message has been removed and agent D's attributes have been updated to reflect the receipt of the message. The rest of the configuration (Conf) is unchanged by this rule.

More generally, a rule can specify any combination of messages and objects in its left and right hand sides and can specify a condition that the instantiated left hand side must meet for the rule to be applicable [4].

$$
\begin{aligned}
r(\boldsymbol{x}) \ : \ & M_1, \ldots, M_n, \langle O_1 : F_1 \mid atts_1 \rangle, \ldots, \langle O_m : F_m \mid atts_m \rangle \\
& \Rightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots, \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\
& \quad \langle Q_1 : D_1 \mid atts''_1 \rangle, \ldots, \langle Q_p : D_p \mid atts''_p \rangle
\end{aligned}
$$

$$M'_1, \ldots, M'_q$$
$$if\ C$$

where r is the rule's label, $\boldsymbol{x}$ 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.

If two or more objects appear in the left hand side of a rule, the rule is *synchronous*. If there is only one object appearing on the left hand side, the rule is *asynchronous*. A concurrent object rewrite theory is called a *distributed object theory* if all of its rules are asynchronous [4].

Given a distributed object theory $\mathcal{R}$, rewriting logic provides an inference system [7] to deduce the finitary concurrent computations possible in $\mathcal{R}$. These computations are identified with proofs of the form $\alpha : C \longrightarrow C'$, where C and C' are terms representing system configurations.

In Maude, a rewriting logic theory $\mathcal{R} = ((\Omega, \Gamma), R)$ is specified as a module with the syntax:

```
mod module-name is
      including module-list
      sorts sort-list
      subsort subsort-specs
     signatures
     variable-declarations
     equations
     rules
endm
```

where `sort-list` is a list of sorts to be used in the module, `subsort-specs` defines subset relationships between the sorts, `signatures` is a set of operator signature definitions, `variable-declarations` defines a set of symbols to be used as variables in the equations and rules of the module, `equations` is the set of equations in Γ, `rules` is the set of rewrite rules in R, and `module-list` is a list of module names whose rewrite theories get textually included in the module being defined. A simple example of a Maude module is

```
mod ND-INT is
      including MACHINE-INT .
      sort NdInt .
      subsort MachineInt < NdInt .
      op _?_ : NdInt NdInt → NdInt [assoc comm] .
      var N : MachineInt .
      var ND : NdInt .
      eq N ? N = N .
      rl [choice]: N ? ND ⇒ N .
endm
```

The module ND-INT defines a sort `NdInt` (nondeterministic integer) as a supersort of `MachineInt` (a builtin sort). It defines a commutative associative

constructor of `NdInts` denoted by **?**. Its one equation serves to remove duplicate `MachineInts` from an `NdInt`. Because **?** is commutative and associative, this module's single rule chooses an arbitrary MachineInt from an NdInt.

An example use of `NdInt` is to rewrite an expression such as
`(1 ? 5 ? 2 ? 1 ? 5) + (3 ? 11 ? 7 ? 3 ? 11)`
which results in the sum of a `MachineInt` nondeterminisically chosen from the first `NdInt` and a `MachineInt` nondeterministically chosen from the second `NdInt`.

In addition to modules declared with the keyword `mod`, Maude supports other kinds of modules, including an object oriented module (**omod**) which we make use of in our protocol specification. **omod**s can declare *classes* and *subclasses*. Each class is declared with the syntax **class** $C \mid a_1 : S_1, \cdots, a_n : S_n$ where C is the class name and for each $a_i : S_i$, a_i is an attribute identifier and S_i is the sort of values for that attribute. Objects in a class are terms of sort **Object** and are written with the previously described syntax.

3 DaAgent in Maude

The DaAgent system runs on a network of processors. A DaAgent server runs on every node in the network that might be visited by an agent or from which an agent might be launched. A node in the DaAgent network consists of the composition of instances of three module types: an agent server, an agent consultant, and some number of agent watchdogs, one per mobile agent.

The DaAgent server on a node services requests from local clients on that node to launch their agents. In addition, it services an agent that migrates from some other node to that node. Every mobile agent is associated with an agent watchdog. Agent watchdogs oversee the functioning of their agents and control the migration of agents to other nodes.

Additionally, there is one agent consultant on every node. An agent consultant on a node determines if a new agent can be launched from that node or if an agent can migrate to that node. Consultants implement an admission control protocol to determine if new agents meet all of the security and computational requirements to run on their nodes.

One of the protocols we have formally specified is called the watchdog-controlled agent recovery protocol (WC-ARP). This protocol automates detection of node failure and the agent recovery process. The key idea is that it uses agent watchdogs on the nodes that an agent visited earlier to monitor the execution of the agent on the current node. The earlier agent watchdogs detect node failures and recover an agent in case of a failure.

In WC-ARP, each agent has an ordered set of the some fixed number (n) of the most recent earlier watchdogs an agent visited. This ordered set of earlier watchdogs is called the agent's *entourage*. An agent's current watchdog is (referred to as AW) and all members of the agents entourage ($AW, AW_1, \cdots, AW_n$) have associated weights ($W_0, W_1 \cdots, W_n$). These weights specify the priority of

that watchdog for ensuring that the agent continues to execute. Since AW contains the most recently computed state of the agent it is the preferred watchdog to ensure the continued execution of the agent. So, in a typical assignment of weights, $W_i > W_j, 0 \leq i \leq n, i < j \leq n$.

During normal execution, AW sends an 'agentAlive' message to all members of the entourage periodically (every t time units). A watchdog AW_j concludes that $AW, AW_1, \cdots, AW_i$ have failed if, in the last $j * sp$ time units, it has not received an 'agentAlive' message from AW, nor has it received a 'migration support request' (msr) message from any of watchdogs $AW_1, \cdots, AW_i$. Here $sp = t * K$, with $K > 0$ being a protocol parameter. Hence, an agent watchdog concludes the failure of AW if it misses K consecutive 'agentAlive' messages.

AW may migrate the agent upon request or a watchdog AW_j may recover an agent if it believes that $AW, AW_1, \cdots, AW_{j-1}$ have failed. However, due to the possibility of a communication partition, AW_j can never be certain that another watchdog has failed.

A threshold ($th \in \mathbb{N}$) is a system parameter used to determine when an agent watchdog is authorized to migrate or recover the agent. A so-called runaway agent is a replicated instance of an agent incorrectly launched by an agent watchdog (say A_j) based on incorrect information that all agents $A_k, k > j$ have failed. If $th \geq (W_0 + W_1 + \cdots + W_n)/2$, runaway agents will never be launched, but in this case the protocol tolerates fewer faults. If $th < (W_0 + W_1 + \cdots + W_n)/2$, runaway agents may be created, but fault tolerance may be increased.

To migrate an agent, AW_j sends a 'migration support request' to every member of the agent's entourage. Then AW_j waits for 'migration supported' (ms) messages in reply. On receiving a 'migration support request', AW_i replies with a 'migration supported' message if it has not yet sent a 'migration supported' message or a 'migration support request' to any other agent watchdog. AW migrates the agent when it has received 'migration supported' messages from entourage members such that the sum of their W_i is greater than th.

An agent watchdog AW_j sends 'migration support request' messages to $AW_{j+1}, \cdots, AW_n$ when

1. AW_j concludes that $AW, AW_1, \cdots, AW_{j-1}$ have failed, and
2. AW_j has yet to send a 'migration supported' message to anyone.

Upon receiving a 'migration support request' message, a watchdog $AW_k, (k > j)$ replies with a 'migration supported' message if

1. AW_k concludes that $AW, AW_1, \cdots, AW_{j-1}$ have failed, and
2. AW_k has not yet sent a 'migration supported' or a 'migration support request' message to anyone.

AW_j recovers the agent from its local checkpoint when it has received 'migration supported' messages from entourage members such that the sum of their W_i is greater than th.

We formalize the WC-ARP protocol as an **omod** in Maude that contains classes for agents and agent controllers. These are defined as

```
class Agent-Controller |
      agents-pending : AgentSet, agents-migrating : Spairs,
      agents-fwd : Wtuples .
class Agent-Home .
class Agent-WatchDog | agents-running : AgentSet,
      awtg-support : Mpairs .
subclasses Agent-Home Agent-WatchDog < Agent-Controller .
class Agent |
      dsts : Principals, livTime : MachineInt, XTime : MachineInt,
      ent : Entourage .
```

The `agent` class models a mobile agent. Instances of this class have attributes `dsts`, which is a list of names of watchdogs to which the agent should be migrated, `livTime`, which specifies how long the agent should live on each watchdog, `XTime`, which is used to count how long an agent has been running on its current watchdog, and `ent` which is the agent's entourage.

The sort of the `dsts` attribute is an ordered list of `Principals` (a subsort of `Oid`) defined as

```
subsort Principal < Oid .
subsort Principal < Principals .
op none : → Principals .
op _ _ : Principals Principals → Principals [ assoc id: none] .
```

An `entourage` is an ordered i-tuple, ($i \leq n$) of agent watchdog names (Principals) defined as

```
subsort Principal < Entourage .
op emptyEnt : → Entourage .
op _‘,_ : Entourage Entourage → Entourage [ assoc id: emptyEnt] .
op addEnt : Principal Entourage → Entourage .
```

The `addEnt` operator is used to add a new watchdog name to the front of an `Entourage`, ensuring that the entourage is a tuple of at most n principals. `AddEnt` is defines as

```
vars A B : Principal .
var Ent : Entourage .
eq addEnt(A,emptyEnt) = A .
eq  addEnt(A,(Ent,B)) =
  if (length((Ent,B)) == entSize) then (A,Ent) else (A,Ent,B) fi .
```

The agent controller class (and its subclasses) defines two kinds of agent controllers: agent watchdogs and agent homes. Agent homes have a set of pending agents (agents to be migrated), a set of agents that are in the process of migrating, and a set of agents that have been forwarded. Agent watchdogs have, in

addition, a set of running agents and a set of agents awaiting support for migration. Agent homes do not require these two additional attributes because agents only begin on agent homes. Therefore, there is no need for an `agent-running` attribute, nor is there need for an `awtg-support` attribute because this is where agents are placed when a watchdog is awaiting 'migration supported' messages from the agent's entourage. When an agent starts out at its home, it has no entourage.

`Agents-pending` and `agents-running` are `AgentSets` which are unordered collections of `Agent` objects defined as

```
subsort Agent < Agents .
op `{_`} : Agents → AgentSet .
op none : → Agents .
op _`,_ : Agents Agents → Agents [ assoc comm id: none ] .
```

The `agents-migrating` attribute is used to record, in an agent controller *AW*, the fact that a message has been sent to migrate an agent. Instances of the `agents-migrating` attribute are pairs of `Agents` and `MachineInts.` The second component of the pair is used to record the amount of time that has elapsed since the migration message was sent. If a 'migration accepted' message is not received after a fixed number of time units, *AW* concludes that the watchdog it attempted to migrate the agent to is unreachable.

We also define the different types of messages of the protocol

```
msgs from_to_migrate_ from_to_ma_
     : Principal Principal Object → Msg .
msgs  from_to_agentAlive_ from_to_msr_ from_to_ms_
     : Principal Principal Principal → Msg .
```

The 'agentAlive' message is the message that *AW* periodically sends to members of the agents entourage. To migrate an agent from controller A to watchdog B, A sends a 'migrate' message to B containing the agent. If B accepts the agent it sends an 'ma' message back to A. To obtain migration support for an agent `Ag`, a watchdog sends an 'msr' message to each member of `Ag`'s entourage. In response, entourage members send an 'ms' message to support an agent's migration.

The `awtg-support` attribute records information used in the protocol after *AW* sends 'msr' messages to an agent's entourage. The sort of values for this attribute is an unordered set of `Mpairs.` Each `Mpair` is an ordered pair consisting of an `Agent` and an ordered set of `Bpairs.` Each `Bpair` is a `Principal` paired with a boolean flag. When a watchdog sends 'msr' messages for an agent `Ag,` it places `Ag` along with a list of `Bpairs` for each watchdog in Ag's entourage in the value of the `awgt-support` attribute. The `Bpairs` are used to record whether or not an 'ms' message has been received from each member.

```
subsort Mpair < Mpairs .
subsort Bpair < Bpairs .
op _;_ : Principal Bool → Bpair .
op _;_ : Agent Bpairs → Mpair .
op none : → Bpairs . op none : → Mpairs .
op _‘,_ : Bpairs Bpairs → Bpairs [  assoc id: none ] .
op _‘,_ : Mpairs Mpairs → Mpairs [ assoc comm id: none ] .
```

The `agents-fwd` attribute of an agent controller is used to record the agents that the controller has forwarded. This is an unordered set of triples consisting of an agent, the number of hops the agent has taken since it was forwarded from this watchdog, and the number of time units since the last alive message was received from the forwarded agent.

The remainder of the WC-ARP **omod** is a collection of rewrite rules that specify the protocol as manipulations of the `Configuration`. Space does not allow the inclusion of the whole specification, but here are some examples. The following rule initiates the run of an agent. It matches an agent home that contains an agent in its `agents-pending` field. It places a migrate message into the `Configuration` and modifies the agent home, moving the agent from the `agents-pending` field to the `agents-migrating` field. It modifies the agent's entourage, initializes the migration timeout counter data structure, and increments the timeout counters (`incAll` of any other agents in the `agents-migrating` field

```
vars A B D : Principal .
var Dsts : Principals .
var N : MachineInt .
var Sp : Spairs .
var Agts : Agents .
rl [BeginRun] :
   (< A : Agent-Home | agents-pending :
                         { < B : Agent | dsts : (D Dsts),
                                         livTime : N,
                                         ent : Ent >, Agts },
                       agents-migrating : Sp >
     Conf)
   ⇒ ((from A to D migrate
             < B : Agent | dsts : Dsts, XTime : 0,
                           livTime : N, ent : addEnt(A, Ent) >)
        < A : Agent-Home | agents-pending :  { Agts },
                           agents-migrating :
                             (< B : Agent | dsts : Dsts,
                                            XTime : 0,
                                            livTime : N,
                                            ent : Ent > ; 0),
                           incAll(Sp) >
        Conf) .
```

When the `XTime = livTime` for an agent, the next rule sends 'migration support messages' to the agent's entourage (the operator `sendMsrs` constructs these), setting up the data structure `(mkBpairs)` to record 'migration supported' replies.

```
crl [requestMigrate] :
    (< A : Agent-WatchDog | agents-running :
                              { < Ag : Agent | XTime : N,
                                               livTime : N,
                                               dsts : Dsts,
                                               ent : Ent >, Agts},
                            agents-fwd : Wp,
                            awtg-support : Mp >
     Conf)
    => (sendMsrs(A,Ag,Ent)
       < A : Agent-WatchDog | agents-running : { Agts },
                              agents-fwd : incAll(Wp),
                              awtg-support :
                                ((< Ag : Agent | XTime : N,
                                                 livTime : N,
                                                 dsts : Dsts,
                                                 ent : Ent > ;
                                    mkBpairs(Ent)), Mp) >
        Conf)
   if (Dsts =/= none) .
```

The formalization of WC-ARP as a theory in rewriting logic uncovered a number of inconsistencies and unspecified conditions in the English specification. For example, the English specification did not address the issue of what to do if a migration message is never replied to. Also it did not address the migration condition early in an agent's tour before its entourage contains n watchdogs.

Most importantly, the formal Maude specification is executable: the result of running the WC-ARP protocol on an initial configuration `initConf` is simulated by rewriting `initConf` in the WC-ARP theory. The resulting configuration (assuming termination) is the term of sort `Configuration` that results. We have simulated the protocol from a number of different initial situations and these simulations have exposed additional errors and omissions in the protocol specification.

We have also formalized, as additional rewrite rules, properties that we would like the WC-ARP protocol to have, for example, no duplicate agents are ever created if the threshold and weights have appropriate values. These rules record state information and halt the simulation if the properties are ever violated. Notice that these simulations do not test that the protocol has desired properties when faults occur. The real benefit of using Maude is realized when its reflective capabilities are exploited.

4 Using Reflection in Maude

Informally, a reflective logic is one in which aspects of its meta-theory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant meta-theoretic aspects. In other words, a reflective logic is a logic which can be faithfully interpreted in itself [2].

In Maude's formalization of reflection, terms are "quoted" to allow them to be manipulated at the meta-level (one step up the reflective hierarchy).[1] Maude provides operators to quote and unquote terms so that they can be moved up and down the reflective hierarchy. Modules are terms which can be manipulated just as all others terms. The operator **up :** `ModuleName` $\rightarrow$ `Term`, produces the meta-level representation of the module whose name is given as argument and the operator **up :** `ModuleName` $\times$ `Term` $\rightarrow$ `Term`, produces the meta-level representation of its second argument in the module whose name is given as its first argument. The meta-level representation of the `NdInt` module given in section 2 is:

```
up(NdInt) =
mod 'ND-INT is
    including 'MACHINE-INT .
    sorts 'NdInt .
    subsort 'MachineInt < 'NdInt .
    op '_?_ : 'NdInt 'NdInt → 'NdInt [ assoc comm ] .
    var 'N : 'MachineInt .
    var 'ND : 'NdInt .
    eq '_?_ [ 'N , 'N ] = 'N .
    rl [ 'choice ] : '_?_ [ 'N , 'ND ] ⇒ 'N .
endm .
```

Maude also enables meta-computation via the builtin operator **meta-rewrite** which takes three arguments: the meta-level representation of a module M, the meta-level representation of a term t, and an integer n. Rewriting a meta-rewrite expression of the form **meta-rewrite(**M,t,n**)** produces the meta-level representation of the term that results from performing n rewrites of the term **down(**M,t**)** in the module **down(**M**)**. In the case where n is 0, an unbounded number of rewrites are performed, halting when no rewrite rule applies, or looping forever if some rule is always enabled. For example, rewriting the expression:

```
meta-rewrite(
    up(NdInt),
    up(NdInt,(1 ? 5 ? 2 ? 1 ? 5) + (3 ? 11 ? 7 ? 3 ? 11)),
   0)
```

results in `({'4} 'MachineInt) = up(4)`.

[1] For details of the quoting process, see [2]

A common use of meta-computation, discussed extensively in [3, 1, 2], is to explicitly specify a rewriting strategy. Maude rewrite theories are required to be neither terminating nor Church-Rosser. Strategies are used to control evaluation in such rewrite theories. It is possible to define a very wide variety of rewriting strategies using rewrite rules at the meta-level and then to execute a specification with one or more of these strategies.

In [4], this strategy mechanism is used to explore (in a breadth-first manner) all the possible rewrite sequences in a theory beginning with some initial state. Using this approach they were able to validate protocol specifications. We have adopted a similar meta-level rewrite strategy to explore the possible execution sequences of the WC-ARP protocol, also in a breadth-first manner.

5 Fault Models

We use the reflective facilities in Maude to inject faults into WC-ARP simulations without changing the WC-ARP theory. This has enabled us to validate the WC-ARP theory under different fault models. The fault modeling technique can be used in conjunction with the breadth-first search strategy to inject faults into a simulation at any desired point of the computation.

A basic strategy for injecting faults is to randomly interrupt a rewrite sequence in WC-ARP and modify the configuration being operated on to emulate the fault. For example, to inject a crash of a watchdog AW into a WC-ARP simulation, we interrupt the simulation at some random point and remove from the configuration AW and all messages to AW.

Randomly interrupting and modifying a rewrite sequence is straightforward to do at the meta-level. As a simple example, here is the partial specification of a meta-level module that runs a WC-ARP simulation for a random number of steps, removes a watchdog from the configuration at that point, and then continues the simulation to completion.

```
(omod RUN-WC-ARP is
       including META-LEVEL[WC-ARP] .
       op oneCrash : Term Qid → Term .
       op remWD : Term Qid → Term .
       op rand : MachineInt → MachineInt .
       var T : Term .
       var W : Qid .
       eq oneCrash(T,W) =
          meta-rewrite(WC-ARP,
                       remWD(meta-rewrite(WC-ARP,T,rand(seed)),W),
                       0) .
endom)
```

The operator `oneCrash` meta-rewrites a random number of times the meta-level representation of a configuration (`T`), removes the watchdog named `W` from the

resulting configuration (`remWD`), and meta-rewrites the modified configuration until termination.

Clearly more general manipulations at the meta-level can be used to emulate many different classes of faults. For example, we can also simulate a watchdog crashing during a WC-ARP run and then recovering at some later point in the run by (1) meta-rewriting an initial configuration a random number of times, (2) removing the watchdog, (3) meta-rewriting the resulting configuration another random number of times, (4) adding the watchdog back into the configuration, (5) continuing meta-rewriting to completion.

Similarly, we simulate a communication partition by (1) meta-rewriting an initial configuration a random number of times, (2) removing any messages between watchdogs in separate partitions, (3) partitioning the watchdogs in the resulting configuration into two or more new configurations, (4) meta-rewriting these new configurations separately for a random number of steps, (5) then merging the configurations and meta-rewriting the merged configuration to termination.

So far we have modeled crashes, crashes with later recovery, and communication partitions as just described.

If, rather than generating a random number of meta-rewrite as described above, we parameterize the number steps, we can choose a set of values for these parameters and then apply the same fault scenario to all possible rewrite sequences (up to some depth) from an initial state. This enables model checking a simulation under a fixed fault scenario.

Note that the module implementing the breadth-first search executes at the meta-level with respect to the WC-ARP module, a module implementing a fault scenario must execute at the meta-meta-level with respect to the WC-ARP module. Also note that by adding one additional level to this reflective tower it is possible to vary the parameters controlling the number of steps in each part of a fault scenario. This enables the systematic checking of fault scenarios in any desired combination of rewrites and faults.

6 Conclusions and Future Work

We have proto-typed the fault-tolerant protocols used in the DaAgent system by formally specifying them as systems of rules in Maude's rewrite logic. Maude has been used to model check the protocol, including fault-tolerant aspects, by explicit state methods; this has been accomplished by using the powerful reflective capabilities provided by Maude. The use of reflection to separate the theory specifying the protocol from the execution and fault models is a powerful abstraction technique that offers what we believe is an unprecedented level of flexibility.

Currently, we are using Maude further explore the fault-tolerant protocols used in the DaAgent system.

In future work, we intend to extend our methods to include support for symbolic model checking. Maude also includes an inductive prover which we may apply to the problem of formally verifying certain properties of the protocols.

References

1. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, , and J. Meseguer. Metalevel computation in Maude. In C. Kirchner, , and H. Kirchner, editors, *2nd Intl. Workshop on Rewriting logic and its Applications*, volume 15. Elsevier, 1998.
2. M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Technical report, SRI International, Menlo Park, CA, Jan 1999.
3. M. Clavel and J. Meseguer. Reflection in rewriting logic and its applications in the Maude language. In *Proceedings of IMSA-97*, pages 128–139, Japan, 1997. Information-Technology Promotion Agency.
4. G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DARPA Information and Survivability Conference and Exposition (DISCEX'00)*, pages 251–265, Hilton Head, South Carolina, Jan 2000. IEEE Computer Society Press.
5. M. Dillencourt, L. F. Bic, and M. Fukuda. Distributed computing using autonomous agents. *IEEE Computer*, 28(8), Aug 1996.
6. R. S. Gray. Agent tcl: A transportable agent system. Technical report, Dartmouth College, November 1995.
7. J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, editor, *Proceedings WADT'97*, volume 1376 of *LNCS*, pages 18–61. Springer Verlag, 1998.
8. S. Mishra, Y. Huang, and H. Kuntur. Daagent: A dependable mobile agent system (fastabstract). In *Proceedings of the 29th International Symposium on Fault-tolerant Computing*, Madison, WI, June 1999. IEEE.
9. A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R Singh. Mobile agent programming in Ajanta. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, Austin, TX, 1999.