Specifying a Reliable Broadcasting Protocol in Maude*

Grit Denker†, J.J. Garcia-Luna-Aceves‡, José Meseguer†, Peter Csaba Ölveczky†, Jyoti Raju‡, Brad Smith‡, Carolyn L. TalcottII

† Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA

{denker,meseguer,peter}@csl.sri.com

‡ Computer Engineering Department, University of California, Santa Cruz, CA 95064, USA

{jj,jyoti,brad}@cse.ucsc.edu

II Computer Science Department, Stanford University, Palo Alto, CA 94305, USA

clt@sail.stanford.edu

March 8, 1999

1 Introduction

This paper presents a formal and executable specification of a new protocol for reliable broadcasting of information in networks with dynamic topology. Reliable broadcasting is not trivial when the topology of the network can change due to failure and mobility. The aim is to ensure that all nodes that satisfy certain connectedness criteria receive the information within finite time, and that the source is notified about it. The protocol should furthermore incur as low latency and as few messages as possible.

Little work exists on reliable broadcast protocols in dynamic networks. Most broadcast protocols are based on the PI and PIF protocols [17, 16], and all the broadcasting protocols for dynamic topologies extending these are based on the routing protocol by Merlin and Segall [8] which incurs too much communication to be attractive for, say, a wireless network [6]. It is the goal of our specification effort to come up with a more general approach (in terms of mobility of nodes) and more efficient protocols for reliable broadcasting in dynamic networks than

*Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by DARPA and NASA through Contract NAS2-98073, by Office of Naval Research Contract N00014-96-C-0114, and by National Science Foundation Grants CCR-9505960 and CCR-9633363.

exists so far. The starting point of our effort was an informal draft specification of the protocol given by José Garcia-Luna [6]. We will refer to this description throughout the text as the original pseudocode.

The protocol is specified in *rewriting logic* [10, 11, 12, 13]. Rewriting logic is a logic which extends algebraic specification techniques to concurrent and reactive systems. Among its possible advantages over other executable specification formalisms are its being based upon a natural and well-known formalism, its natural integration of static and dynamic system aspects, the abstract modeling of communication, and its possibility to define execution strategies in the logic itself [1, 3, 2, 4]. Rewriting logic seems particularly suitable for specifying security and communication protocols. Such protocols are complex enough to warrant prototyping and their operational nature fits very well with rewriting logic. The use of rewriting logic for the specification and analysis of security protocols is shown in [5].

Apart from presenting the final network protocol, this report also aims at giving some snapshots of the specification effort of the group illustrating how the rewriting logic language *Maude* [3, 11, 15] can be used in the different stages of the specification and validation process. Due to the complexity of the protocol, we first focussed on specifying the protocol for the subcase of static networks, i.e., networks where no channels or nodes are lost or created. The protocol for the dynamic network setting was then developed, based of the final specification for the static setting.

Section 2 of this report briefly introduces rewriting logic and Maude. Section 3 describes the given (informal) protocol and our specification efforts for the static network case. Section 4 describes the protocol for dynamic networks. The appendices give the specifications and the strategy language used and show the results of executing the specifications.

2 Distributed Concurrent Objects in Rewriting Logic

We model a distributed system configuration as a soup (multi-set) of concurrent objects and messages that behave according to a set of rewrite rules describing the behavior of individual objects. The target application, broadcasting protocols in distributed systems, only use asynchronous communication. Thus, we specialize the rewriting logic object model [7, 9, 14] to the case of asynchronous message passing. We call these *distributed objects*.

We first explain how concurrent object systems are axiomatized in rewriting logic. We identify the subclass of object rewrite theories axiomatizing asynchronous behavior, called *distributed objects*.

In general, a rewrite theory is a pair $\mathcal{R} = ((\Omega, \Gamma), R)$, with (Ω, Γ) an equational specification with signature of operations Ω and set of equational axioms Γ; and with R a collection of labelled rewrite rules. The equational specification describes the *static* structure of the system's state space as an algebraic data type. The *dynamics* of the system are described by the rules in R, that specify local concurrent *transitions* that can occur in the system axiomatized by $\mathcal{R}$, and that can be applied *modulo* the equations Γ.

Let us then begin by explaining how the state space of a concurrent object system can be axiomatized as the initial algebra of an equational theory (Ω, Γ). That is, we explain the key state-building operations in Ω and the equations Γ that they satisfy. The concurrent state of an object-oriented system, often called a *configuration*, has typically the structure of a *multiset* made up of objects and messages. Therefore, we can view configurations as built up by a binary multiset union operator, which we can represent with empty syntax (i.e., juxtaposition) as $\_\ \_ : \mathbf{Conf} \times \mathbf{Conf} \longrightarrow \mathbf{Conf}$. (Following the conventions of mix-fix notation, we use underbars "_" to indicate argument positions.) The multiset union operator $\_\ \_$ is declared to satisfy the structural laws of associativity and commutativity and to have identity $\emptyset$. Objects and messages are singleton multiset configurations, and belong to subsorts $\mathbf{Object}, \mathbf{Msg} < \mathbf{Conf}$, so that more complex configurations are generated out of them by multiset union.

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. This finishes the description of some of the sorts, operators, and equations in the theory (Ω, Γ) axiomatizing the states of a concurrent object system. Particular systems are axiomatized by providing additional operations and equations, specifying, for example, the data operations on attribute values and the structure of messages. But the top level structure of the concurrent object system is always given by the multiset union operator.

The associativity and commutativity of a configuration's multiset structure make it very fluid. We can think of it as "soup" in which objects and messages float, so that any objects and messages can at any time come together and participate in a concurrent transition corresponding to a communication event of some kind. In general, the rewrite rules in R describing the dynamics of an object-oriented system can have the form

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

where r is the rule's label, $\bar{x}$ is a list of the variables occurring in the rule, the Ms 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. If two or more objects appear in the lefthand side, we call the rule *synchronous*, because it forces those objects to jointly participate in the transition. If there is only one object in the lefthand side, we call the rule *asynchronous*. A concurrent object rewrite theory is called a *distributed object theory* if the rules are asynchronous, with exactly one object and at most one message on the lefthand side. A distributed object configuration is then a multi-set of objects and messages in the context of a distributed object rewrite theory.

Rewriting logic then gives an inference system [10] to deduce, for a system axiomatized by a rewrite theory $\mathcal{R}$, which are all the finitary concurrent computations possible in such a system. Such computations are identified with *proofs* of the general form $\alpha : C \longrightarrow C'$ in the logic, where C and C' are terms representing system configurations.

2.1 Maude

We briefly summarize the syntax of Maude. There are three types of modules:

- *functional* modules, that are equational theories used to specify algebraic data types; they are declared with the syntax `fmod ... endfm`,
- *system* modules, that are rewrite theories specifying concurrent systems; they are declared with the syntax `mod ... endm`, and
- *object-oriented* modules, that provide special syntax to specify concurrent object-oriented systems but are entirely reducible to system modules; they are declared with the syntax `omod ... endom`.

All the above modules can be *parameterized*, and have an *initial model semantics*. That is, they specify the initial models of their corresponding theories, or, in the case of parameterized modules, the free extension of a model of the parameter theory to a model of the body.

Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts*—that is, types—and *subsorts*—that is, subtype inclusions. In object-oriented modules one can declare *classes* and *subclasses*. Each class is declared with the syntax

$$\texttt{class}\ C\ \texttt{|}\ a_1 : S_1\ \texttt{,}\ \ldots\ \texttt{,}\ a_n : S_n$$

where C is the name of the class, and for each $a_i : S_i$, a_i is an attribute identifier, and S_i is the sort inside which the values of such an attribute identifier must range.

One can also declare *operators*, introduced with the `op` keyword. They can have user-definable syntax, with underbars '_' marking the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc` and `comm`, stating, for example, that the operator is associative and commutative. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. In object-oriented modules, message-constructing operators are introduced with the keyword `msg`.

There are three kinds of logical axioms, namely, *equations*—introduced with the keywords `eq`, or, for conditional equations, `ceq`—*memberships*—declaring that a term has a certain sort and introduce with the keywords `mb` and `cmb`—and *rewrite rules*—introduced with the keywords `rl` and `crl`. Functional modules can only have equations and memberships. System and object-oriented modules can have any of the three kinds of axioms. The mathematical variables in such axioms are declared with the keywords `var` and `vars`.

3 RBP for Static Network Topologies

In this section we describe how Maude was used to arrive at a final formal specification of the protocol, the *reliable broadcast protocol (RBP)*, for the static topology only, from a given informal specification. In particular, Section 3.1 presents the informal first specification for the static case. Section 3.2 outlines the specification process used to to arrive at the final protocol, and Section 3.3 present the protocol for the static case.

3.1 The informal specification

The following is a somewhat abbreviated version of the informal specification given in [6], which describes the proposed protocol for the static topology. A static network is modeled as an undirected connected graph where each node knows its neighbors but does not know anything more about the topology. The objective of the protocol is to broadcast the message from a source node to all the nodes in the network and to give feedback to the source. We only consider one round of the protocol – only one message is to be broadcasted from some source j to all nodes reachable from j. Handling multiple messages from each source can be easily accommodated by means of bit vectors or counters.

Initially, for every node i in the network, $s_i = nil$ (this value denotes the "parent", i.e., the propagator, of node i) and $SN_i = nil$ (SN_i denotes the set of neighbors from which i has received the message). The cycle starts with the source node sending the message to all its neighbors, and terminates when the source node has received an acknowledgment from all its neighbors.

Every node i other than the source node reacts in the following way to input events (reception of message or acknowledgment):

1. If i receives a message for the first time, say from node p, it sets $s_i = p$ (p is the parent of i), adds the node p to the set SN_i (the set of nodes from which it has received the message), and propagates the message to every node in the set $N_i - p$.
2. If node i receives a message from a node q when it has already has received the message from some other node, then it simply adds q to the set SN_i.
3. When node i has received an acknowledgment from every node in the set $N_i - SN_i$, it sends an acknowledgment to every node in the set $SN_i - s_i$.
4. When node i receives acknowledgment from every neighbor in the set $SN_i - s_i$, it sends an acknowledgment to its parent node s_i.

3.2 The specification process

3.2.1 Formalization

There are many fuzzy requirements and much implicit knowledge in most informal specifications, such as in the starting point of our specification effort, a specification written in natural language and pseudo-code. The *formalization* part of the specification process aims at clarifying fuzzy requirements and make essential implicit knowledge explicit. During the formalization process, we realized that it was not clear whether one should take into account scenarios where an acknowledgment from node i to j may be received before before the message sent from i to j. In other words, are messages/acknowledgments sent from i to j received in the same order as they are sent? This important aspect was not mentioned in the informal specification and provides an example of how the formalization effort helps to make implicit assumptions explicit. In cooperation with the network experts from Santa Cruz it indeed turned out that the order of messages is preserved along a channel. There are two types of messages sent around in the system, the message to be broadcast, called `msg`, and the acknowledgements. The issue of message ordering is treated by a simplified version of sequence numbering, where an acknowledgment sent after the sending of the message `msg` along a channel is denoted `ack`, and where an acknowledgment sent without being preceded along the same channel by the message is denoted `ackParent`. Order-preservation of channels is then modeled by not reading an `ack` before reading the message from the same sender.

Regarding the informal specification, point 4 seems somewhat unclear, as it is not stated *when* the node should acknowledge its parent. Whenever it has received acknowledgments from every node in "current" $SN_i - s_i$? Only after the set SN_i could no longer be extended? If so, how can one know when SN_i is non-extendible?

In our understanding of the protocol, a node should acknowledge its parent when it has received acknowledgments from every neighbor except the parent.

3.2.2 Validation

A major advantage of rewriting logic specifications is that they can be *validated* immediately by executing test cases to provide quick feedback on the specification. This prototyping possibility comes for free, and we used this feature every time the specification was modified, and often encountered counterexamples immediately on quite simple test examples (such as a network of three nodes). In this validation effort, we executed the test cases using Maude's default interpreter which simulated some arbitrary run of the protocol for some given networks. The validation effort helped eliminate errors of syntax and of thought; furthermore, the built-in Maude facilities for tracing an execution were useful for discovering where the error occurred. This validation and correction cycle lead to a final extraction and clear formalization of the basic ideas of the starting informal protocol. In particular, it was agreed upon that the following follows from the informal specification:

1. A node should acknowledge all its "siblings" when it has received something (message or acknowledgment) from all its neighbors.
2. A node should acknowledge its "parent" node when it has received acknowledgments from all its neighbors except the parent.

Our final version of the protocol based on these ideas was validated using Maude's default interpreter. The extensive testing always returned the expected (and hoped-for) result.

3.2.3 Exploratory analysis

To increase our confidence in the specification even more or to invalidate it before the daunting tasks of proving it correct and/or extending it to the dynamic topology setting, the specification can be subject to close analysis using the meta-programming features of Maude. Since the specification should be terminating, one could apply a strategy that explores all possible rewrite paths from some given initial term. In particular, we wrote a strategy for finding every non-rewritable state reachable from the initial state. For non-terminating systems, this setting can be modified to give e. g., every state which is reachable in less than 50 one-step rewrites from some initial state.

Executing the specification using this strategy on a clique with three nodes did *not* produced the hoped-for result, namely a singleton set of irreducible states. Instead, the set of irreducible configurations reachable from the clique of three nodes included a term which indicated a deadlock before the expected end of one round of the protocol. A simple analysis of the output explained the error. If a node `a` is both "grandparent" and "sibling" to a node `b`, then, according to the ideas underlying the protocol, there is a deadlock as follows: Node `c` cannot acknowledge its parent `a` before it has received the `ackParent` message from its child `b`. Node `b` cannot acknowledge its parent `c` before it has received an acknowledgment from its sibling `a`. And `a` cannot acknowledge `c` before it has received something (which in this case only can be an `ackParent` message) from `c`!

Using Maude's meta-programming features, the user may himself define the rewrite strategies, thereby analyzing the specification in various ways. Although we only needed the quite straight-forward exploratory analysis to invalidate our first version, this capability is very useful for analyzing various executions and extracting the relevant information from these automatically.

Using the information from the exploratory analysis, the group came up with a new protocol for the static network setting, which is presented next.

3.3 Maude Specification

In this section we outline the Maude specification of the modified protocol for the static topology.

For each run of the protocol there is a single source node, and there is one message that needs to be sent to all the other nodes in the network. (Handling multiple messages from each source can be accommodated easily by means of bit vectors or counters.) The protocol proceeds roughly as follows:

1. The source node sends the message to all its neighbors.

2. When a node n receives the message for the first time, it remembers the sender (the *parent* of n), and sends the message to all its neighbors except its parent.

3. When a node n receives a message it has already seen, the node sends an acknowledgment to the sender of this latest message (its *sibling*).

4. When a node has received an acknowledgment from all its neighbors except its parent, it sends an acknowledgment to the parent and its cycle is finished.

5. The protocol terminates when the source has received acknowledgments from all its neighbors.

The protocol is similar to the well-known PIF protocol for static networks, but incurs a few more messages. In comparison with PIF-based protocols, the innovation of this protocol is the extra set of acknowledgment messages between "sibling" nodes. This adds to the complexity of the protocol, but the redundancy will result in an improved protocol for dynamic networks, better than the existing extensions of the PIF protocol.

We need to provide some application-specific data types such as sets of object identifiers and multiple copies of a certain message. First, we define object identifiers with two special object values `noOid` and `source`:

```
sorts   DefOid SrcOid DefSrcOid .
subsort Oid < DefOid .              *** Oid with default value
subsort Oid < SrcOid .              *** Oid with value source
subsort SrcOid < DefSrcOid .
subsort DefOid < DefSrcOid .
ops a b c d e f : -> Oid .          *** Six object names used.
op noOid : -> DefOid .
op source : -> SrcOid .
```

Sets of `Oid`s are multisets with an additional idempotency axiom:

```
sort OidSet .
subsort Oid < OidSet .
op nil : -> OidSet .
op _ _ : OidSet OidSet -> OidSet [assoc comm id: nil] .
var A : Oid .
eq A A = A .
```

together with the usual functions on sets. The class `Node` is declared

```
class Node | nbs: OidSet, parent: DefSrcOid,
             recdMsg: OidSet, recdAck: OidSet .
```

where `nbs` denotes the set (of the names) of neighbor objects, `parent` the node's parent (or the value `noOid` if it has no parent and `source` if the node is the source node for the cycle), and `recdMsg` and `recdAck` the sets of nodes from which the objects has received messages and acknowledgments respectively.

The set of messages is declared

```
msg to_broadcast_ : Oid MessageContent -> Msg .
msg msg_from_to_ : MessageContent Oid Oid -> Msg .
msg ackParent_from_to_ : MessageContent Oid Oid -> Msg .
msg ack_from_to_ : MessageContent Oid Oid -> Msg .
```

where `(to A broadcast m)` indicates that node `A` is the source of the broadcasting cycle of the message with content `m`[1]. The message `(msg m from A to B)` represents a message with content `m` sent from `A` to `B`, a message `(ackParent m from A to B)` represents an acknowledgement from `A` to its parent `B`, and `ack m From A to B` represents an acknowledgment from `A` to its sibling `B`[2].

A node needs to send a message to all its neighbors except the parent at certain stages of the protocol. At first one might not recognize such a simple fact, and model the broadcast to the neighbors as a separate message sent to each. Upon closer inspection, it is natural to consider this a single event. Transforming a set of events into a single, more abstract one, can be quite tedious in process algebras. In rewriting logic it turns out to be quite straight-forward to add a new feature of "multimessages" to the specification. A message

```
(multimsg m from A To N)
```

for `N` a set `B1, B2, ..., Bn` of objects can simply be defined as denoting the construct

```
(msg m from A to B1) (msg m from A to B2) ... (msg m from A to Bn)
```

Formally:

```
vars A B : Oid .   var N : OidSet .   var M : MessageContent .
eq (multimsg M from A to B N) = (msg M from A to B)
                                (multimsg M from A to N-B) .
eq (multimsg M from A to nil) = none .
```

Given the variable declarations

[1] The message content will always be `m` in the protocol, since only one round of it is specified.

[2] As mentioned earlier, the division into `ack` and `ackParent` messages can be seen as an abstraction from sequence numbers in a channel or other means of enforcing the necessary message-order.

```
vars A B C : Oid .  vars N N' N'' : OidSet .
var M : Message .   var D : DefOid .
var S : SrcOid .    var O : DefSrcOid .
```

the transitions in the system can be defined as follows:

```
rl [Start] : (to A broadcast m) <A:Node|nbs: N, parent: noOid> =>
            <A:Node|parent: source> (multimsg m from A to N) .
```

This rule starts the cycle by letting the source node send a message to all its neighbors.

```
rl [RecMsg1] : (msg m From A To B) <B:Node|nbs: N, parent: noOid> =>
              <B:Node|parent: A> (multimsg m from B to N-A) .
```

This rule only applies to objects whose parent attribute has the value `noOid`, i.e., this would be the first time object `B` received the message, which it then propagates to all its neighbors except the parent.

```
rl [RecMsg2] :
   (msg m from A to B) <B:Node|nbs: N, parent: S, recdMsg: N'> =>
   <B:Node|recdMsg: N' A> (ack m from B to A) .
```

Due to the way the sorts are defined, `S` cannot represent the value `noOid`. Consequently, `B` has a parent and therefore has already seen the message – `B` should not propagate the message further, but should instead send an acknowledgement to the sender.

```
rl [RecAckSibling] :
   (ack m from A to B) <B:Node| recdMsg: A N, recdAck: N'> =>
   <B:Node|recdAck: N' A> .
```

An acknowledgment from a sibling is not consumed unless the node `B` already has received a message from the sibling. This restriction on the consumption of `ack`-messages is justified by the assumption that channels preserve the order of messages.

```
rl [ackParent] :
   <A:Node|nbs: B N, parent: B, recdMsg: N', recdAck: N> =>
   <A:Node|parent: noOid, recdMsg: nil, recdAck: nil>
   (ackParent m from A to B) .
```

Node `A` has received an acknowledgment from all neighbors except the parent and can acknowledge the parent and “reset” its attributes for a next cycle.

```
rl [RecAckParent] :
   (ackParent m from A to B) <B:Node|recdAck: N> =>
   <B:Node|recdAck: N A> .
```

The node just records the reception of an acknowledgment message.

```
rl [ResetSource] :
   <A:Node|nbs: N, parent: source, recdMsg: N', recdAck: N> =>
   <A:Node|parent: noOid, recdMsg: nil, recdAck: nil> .
```

The source has received an acknowledgment from all neighbors, and a cycle of the protocol terminates, while the source resets its attributes.

The initial state is defined as a graph, i.e., a set of objects, each identifying its neighbors, together with a `broadcast` message which triggers one round of the protocol. An initial configuration `test4` with four nodes `a`, `b`, `c`, and `d`, channels `(a,b)`, `(a,c)`, `(b,c)`, and `(c,d)`, and source `a` can be defined:

```
eq test4 =
     (to a broadcast m)
     <a:Node|nbs: b c, parent: noOid, recdMsg: nil, recdAck: nil>
     <b:Node|nbs: a c, parent: noOid, recdMsg: nil, recdAck: nil>
     <c:Node|nbs: a b d, parent: noOid, recdMsg: nil, recdAck: nil>
     <d:Node|nbs: c, parent: noOid, recdMsg: nil, recdAck: nil> .
```

Executing the specification with `test4` as an initial term should end with an irreducible state which equals the initial state minus the broadcast message. Appendix A shows a transcript of a validation session using Maude's default interpreter on various test configurations.

Exploratory analysis of all execution scenarios in the system has given the hoped-for results for the networks with a few nodes. Given the exponential explosion of possible runs with increases in network sizes, the system gets mired even for fairly small networks when called upon to perform an exhaustive exploration of possible runs. On our platform it failed to terminate within a reasonable time for a network of 6 nodes. Apart from the exponential number of runs, this fact could also be attributed to non-optimal coding of the strategy expression for finding all rewrites, which emphasized readability over efficiency. Appendix B shows the strategy language used to derive all irreducible forms reachable from a given initial state, as well as a transcript of applying that strategy to test configurations of the protocol.

4 RBP for Dynamic Network Topologies

4.1 Maude Specification

In dynamic networks links can fail and come up. The desired correctness criterion is that as along as a node has a physical path to the source of the message, it is gurantanteed to get messages from the source. The dynamic case requires the handling of connecting and disconnecting network sections. If a new link (i, b) is established and i and b reside in different, unconnected parts of the network, then messages have been sent to i must be forwarded to b and vice a versa. For example, if i has heard the latest message from source j and b has not yet heard since it had no physical path to j, then i forwards this message. Node i itself may be active or passive. In the former case i forwards the message as it does for any other neighbor. Thus, i extends the original diffusion computation to b. If node i already terminated the original diffusion computation, i.e., i is passive, then i starts a local diffusion computation in which it appears to be the source of the newly send message. The implementation of the dynamic case of RBP assumes Integer in order to label messages. This allows to decided which parts of a network have older messages than other parts.

In the dynamic case of RBP a richer data structure is necessary than in the static case. There can be more than one message in a network. For example, different partitions of the network can have different messages which are in transmittal and those messages might be forwarded to another partition when a new link comes up. Thus, each node has to store the latest number of a message with respect to a specific source node. Similar, all other information like the state of a node or its parent are parametric in the source node. Thus, most of the node attributes have the source node as parameter. In particular, the following attributes form the data structure of a node: (1) a set of direct neighbors and several attributes which store information with respect to a given source node j. These are (2) a flag which specifies whether the node is active or passive for j (i.e., a node is active if it has sent the latest message of j to its neighbors and has not yet received all acknowledgements), (3) a parent for messages from j, that is, the node from which it got first the latest message of source j, (4) the latest message number of messages from j, and (5) the state of a neighbor with respect to j, i.e., a flag which specifies whether the node is active or passive for a neighbor with respect to a message from j. A node i being active for a neighbor k with respect to a message from a source node j means that i has forwarded j's message to neighbor k but still waits for an acknowledgement from k.

We define the following class in Maude:

```
class Node | nbs : OidSet, states : IdStatusPFun,
             parents : IdIdPFun, seqNos : IdIntPFun,
             nbsStates : IdIdStatusPFun .
```

The neigbors set is a set of object identifiers. The other attributes are partial functions, that is, for a given source node identity, they deliver the state (`states : IdStatusPfun`), the identity of the parent (`parents : IdIdPFun`), the latest sequence number (`seqNos : IdIntPFun`), or, the state of a neighbor (`nbsStates : IdIdStatusPFun`). We modelled these data types in Maude by defining pairs and tuples which have as the first parameter always the identity of the source node. Partial functions are sets of pairs or triples.

```
*** sorts for pairs/triple
*** first parameter is the source node identity
sort Status .
sort IdStatusPair .
sort IdIdPair .
sort IdIntPair .
sort IdIdStatusTriple .

*** sorts for partial functions
sort IdStatusPFun .
subsort IdStatusPair < IdStatusPFun .
sort IdIdPFun .
subsort IdIdPair < IdIdPFun .
sort IdIntPFun .
subsort IdIntPair < IdIntPFun .
sort IdIdStatusPFun .
subsort IdIdStatusTriple < IdIdStatusPFun .
```

A variety of constants and operators is defined for these sorts. We only list a few here to give the flavor of the specification. The full specification is given in Appendix C.

```
*** operators for pairs/triplets
op active : -> Status .
op passive : -> Status .
op state : Oid Status -> IdStatusPair .
op parent : Oid Oid -> IdIdPair .
op seqNo : Oid MachineInt -> IdIntPair .
op nbState : Oid Oid Status -> IdIdStatusTriple .

*** operators for partial functions
op mtstates : -> IdStatusPFun .
op states : IdStatusPFun IdStatusPFun
            ->  IdStatusPFun [assoc comm id: mtstates] .
```

Assume the following node instance:

```
< i : Node | nbs : set(k,l), states : state(j,active),
             parents : parent(j,k), seqNos : seqNo(j,5),
             nbsStates : nbState(j,l,active) >
```

Node i has two neighbors, k and l. Currently, i is in active in a diffusion computation for source j. The latest message it got from j has the number 5, and i already forwarded this message to its neighbor l without having yet received an acknowledgement. Thus, i is in an active neighbor state for l with respect to source j.

The main messages in the system are (1) sending and acknowledging a message (number) from a source node between network nodes, and (2) establishing and deleting links between nodes.

```
msg msg_To_From_Src_ : MachineInt Oid Oid Oid -> Msg .
msg ack_To_From_Src_ : MachineInt Oid Oid Oid -> Msg .
msg newlink : Oid Oid -> Msg .
msg failure : Oid Oid -> Msg .
```

Messages are sent out to all neighbors in a broadcasting style. Thus, we define multisets of messages. `multimsg_To_From_Src_` takes a message number, a set of addressee nodes, a sender node, and a source node, and generates a send message for each addressee in the set.

```
*** multimessage-declarations:
msg multimsg_To_From_Src_ : MachineInt OidSet Oid Oid
                                -> Configuration .

*** multimessage definition:
eq multimsg I To A From B Src C = msg I To A From B Src C .
eq multimsg I To mtnbs From B Src C = none .
eq multimsg I To nbs(A,OIDSET) From B Src C
   = (msg I To A From B Src C) (multimsg I To OIDSET From B Src C) .
```

In the following we describe some of the rules for the dynamic RBP. The full specification is given in Appendix C.

Starting a diffusion computation

A source node sends the latest message number to all its neighbors (`multimsgToFromSrc(N + 1,OIDSET,A,A)`). The source node has to be passive to start a new diffusion computation.

```
crl [Send] :
     < A : Node | nbs : OIDSET,
                  states : states(state(A,passive),IDSTATUSPFUN),
                  parents : IDIDPFUN,
                  seqNos : seqNos(seqNo(A,N),IDINTPFUN),
                  nbsStates : IDIDSTATUSPFUN >
   =>
     < A : Node | states : assignIdStatusPFun(IDSTATUSPFUN, A, active),
                  parents : assignIdIdPFun(IDIDPFUN, A, A),
                  seqNos : assignIdIntPFun(IDINTPFUN, A, N + 1),
                  nbsStates : assignAllIdIdStatusPFun(
                                      IDIDSTATUSPFUN, A, OIDSET, active) >
     (multimsg (N + 1) To OIDSET From A Src A)
   if OIDSET =/= mtnbs .
```

As a result the source node sets its states to be active with respect to itself being the source (`states : assignIdStatusPFun(IDSTATUSPFUN, A, active)`), it becomes its own parent and it increases the latest sequence number. Since `A` is the source of the message, all neighbors of `A` will be set to active with respect to `A`. That means, `A` is waiting for acknowledgments from its neighbors.

Receiving a message

The reaction of node i upon receipt of message m from p depends on the message number. In the following we give a pseudocode representation of the current Maude implementation. In accord to the pseudocode suggested by García-Lunes, we use the following notation to refer to attributes: N^i denotes the neigbors of node i (`nbs`), and for source node j ST^i_j denotes the state of node i (`state(j,_)`), s^i_j denotes the parent of i (parent(j,_)), SQ^i_j denotes the last sequence number of i (`seqNo(j,_)`), and ST^i_{jk} denotes the state of i with respect to neighbor k (`nbsState(j,k,_)`).

if $SQ^i_j = m$
  send ack_m to p for source j
if ($SQ^i_j < m$ or SQ^i_j is not defined)
  $s^i_j := p$
  $SQ^i_j := m$
  $ST^i_{jp} := \mathit{passive}$
  if i has at least one more neighbor other than p
    $ST^i_j := \mathit{active}$
    $ST^i_{jk} := \mathit{active}$ for all $k \in N^i - p$
    send message m for source j to all $k \in N^i - p$
  if $N^i = p$
    $ST^i_j := \mathit{passive}$

send ack_m for source j to p
if $ST^i_j = \textit{active}$ and $SQ^i_j > m$
ST^i_{jp} := active
send message SQ^i_j for source j to p
if $ST^i_j = \textit{passive}$ and $SQ^i_j > m$
$ST^i_j := \textit{active}$
$s^i_j := i$
$ST^i_{jp} := \textit{active}$
send message SQ^i_j for source j to p

There are four main cases for receiving a message. If a node repeatedly receives the current message (i.e., $SQ^i_j = m$), then it only replies with an acknowledgement to the sender. This corresponds to the Maude rule

```
rl [RepeatRecCurrentMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  states : IDSTATUSPFUN,
                  parents : IDIDPFUN,
                  seqNos : seqNos(seqNo(C,M),IDINTPFUN),
                  nbsStates : IDIDSTATUSPFUN >
     (msg M To A From B Src C)
   =>
     < A : Node | nbs : nbs(B,OIDSET) >
     (ack M To B From A Src C) .
```

If a node receives a newer message ($SQ^i_j < m$) or a message from a node for which it has no current message number (as it is the case if a new link comes up), then it stops an earlier diffusion source and starts a new one with the newer message. The node from which it got the latest message becomes the parent ($s^i_j := p$). It forwards the newer message to all neighbors other than p and waits for their acknowledgement. In case there are no more neighbors, it can acknowledge the receipt of the message. We reflect this case in two Maude rules: one for the leaf-case and one for the case where the receiving node is not a leaf.

```
*** NONLEAFCASE
crl [RecNewMsg] :
      < A : Node | nbs : nbs(B,OIDSET),
                   states : IDSTATUSPFUN,
                   parents : IDIDPFUN,
                   seqNos : IDINTPFUN,
                   nbsStates : IDIDSTATUSPFUN >
     (msg M To A From B Src C)
     =>
      < A : Node | states : assignIdStatusPFun(IDSTATUSPFUN, C, active),
                   parents : assignIdIdPFun(IDIDPFUN, C, B),
```

```
                  seqNos : assignIdIntPFun(IDINTPFUN, C, M),
                  nbsStates : assignAllIdIdStatusPFun(
                                    assignIdIdStatusPFun(
                                         IDIDSTATUSPFUN, C, B, passive),
                                    C, OIDSET, active) >
      (multimsg M To OIDSET From A Src C)
    if (OIDSET =/= mtnbs)
       and forSourceSmallerOrNoSeqnoInIdIntPFun(C,M,IDINTPFUN) .

*** LEAFCASE
crl [RecNewMsgLeaf] :
     < A : Node | nbs : B,
                  states : IDSTATUSPFUN,
                  parents : IDIDPFUN,
                  seqNos : IDINTPFUN,
                  nbsStates : IDIDSTATUSPFUN >
     (msg M To A From B Src C)
    =>
     < A : Node | states : assignIdStatusPFun(IDSTATUSPFUN, C, passive),
                  parents : assignIdIdPFun(IDIDPFUN, C, B),
                  seqNos : assignIdIntPFun(IDINTPFUN, C, M),
                  nbsStates : assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                   C, B, passive) >
     (ack M To B From A Src C)
   if forSourceSmallerOrNoSeqnoInIdIntPFun(C,M,IDINTPFUN) .
```

If a node receives an older message ($SQ_j^i > m$), then its reaction depends on its state. If the node is still active than it extends the ongoing diffusion computation by notifying the sender about the newest message. If the node is already passive, then it starts a local diffusion computation with itself as the source. Again, two Maude rules define this behavior:

```
*** Active Receiver
     < A : Node | nbs : nbs(B,OIDSET),
                  states :  states(state(C,active),IDSTATUSPFUN),
                  seqNos : seqNos(seqNo(C,I),IDINTPFUN),
                  nbsStates : IDIDSTATUSPFUN >
     (msg M To A From B Src C)
   =>
    < A : Node | nbsStates : assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                   C, B, active) >
     (msg I To B From A Src C)
    if I > M .

*** Passive Receiver
crl [PassiveRecOldMsg] :
```

```
 < A : Node | nbs : nbs(B,OIDSET),
              states : states(state(C,passive),IDSTATUSPFUN),
              parents : IDIDPFUN,
              seqNos : seqNos(seqNo(C,I),IDINTPFUN),
              nbsStates : IDIDSTATUSPFUN >
 (msg M To A From B Src C)
=>
 < A : Node |  states : assignIdStatusPFun(IDSTATUSPFUN, C, active),
              parents : assignIdIdPFun(IDIDPFUN, C, A),
              nbsStates : assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                               C, B, active) >
 (msg I To B From A Src C)
 if I > M .
```

Receiving an acknowledgement

The following pseudocode summarizes the behavior of a node i when it receives an acknowledgment for message m and source j from node q.

if SQ^i_j ¡ m
  ack$_m$ for source j to q
if $SQ^i_j = m$ and $ST^i_j = passive$
  receive/consume the ack$_m$, do nothing else
if $SQ^i_j = m$ and $ST^i_j = active$ and ST^i_{jq} is defined (either passive or active)
  if s^i_j is defined
    if $ST^i_{jk} = passive$ or not defined for all $k \in (N^i - (s^i_j + q))$
      $ST^i_j := passive$
      $ST^i_{jq} := passive$
      if $s^i_j \neq i$
        send ack$_m$ for source j to s^i_j
    if $ST^i_{jk} = active$ for at least one $k \in (N^i - (s^i_j + q))$
      $ST^i_{jq} := passive$
if $SQ^i_j > m$ and $ST^i_j = active$
  $ST^i_{jq} := active$
  send message SQ^i_j for source j to q
if $SQ^i_j > m$ and $ST^i_j = passive$
  $ST^i_j := active$
  $s^i_j := i$
  $ST^i_{jq} := active$
  send message SQ^i_j for source j to q

When a node receives an acknowledgment for a message which is newer than its current one, then it sends back an acknowledgement with its own sequence number. This will later cause the neighbor to resend the latest message.

```
crl [RecAckNewerMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  seqNos : seqNos(seqNo(C,I),IDINTPFUN) >
     (ack M To A From B Src C)
   =>
     < A : Node | nbs : nbs(B,OIDSET) >
     (ack I To B From A Src C)
    if M > I  .
```

When a passive node receives an acknowledgement for its current message number, then it does not do anything.

```
rl [PassiveRecAckCurrentMsg] :
    < A : Node | nbs : nbs(B,OIDSET),
                 states :  states(state(C,passive),IDSTATUSPFUN),
                 seqNos : seqNos(seqNo(C,M),IDINTPFUN) >
     (ack M To A From B Src C)
   =>
     < A : Node | nbs : nbs(B,OIDSET) > .
```

When an active node receives an acknowledgement for its current message number and the status of the neighbor q which has sent the acknowledgement is known, i.e., either active or passive, then the node can itself set to passive and depending on whether or not all neighbors are passive, it can acknowledge its parent.

```
rl [ActiveRecAckCurrentMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  states :  states(state(C,active),IDSTATUSPFUN),
                  parents : parents(parent(C,X),IDIDPFUN),
                  seqNos : seqNos(seqNo(C,M),IDINTPFUN),
                  nbsStates : nbsStates(nbState(C,B,STAT1),
                                        IDIDSTATUSPFUN) >
     (ack M To A From B Src C)
   => if allNbsPassiveWRTSrcUnlessParent(IDIDSTATUSPFUN, C, X) == true
      then if X =/= A
           then < A : Node | states : states(state(C,passive),
                                             IDSTATUSPFUN),
                             nbsStates : nbsStates(nbState(C,B,passive),
                                                   IDIDSTATUSPFUN) >
                  (ack M To X From A Src C)
           else < A : Node | states : states(state(C,passive),
                                             IDSTATUSPFUN),
                             nbsStates : nbsStates(nbState(C,B,passive),
                                                   IDIDSTATUSPFUN) >
```

```
          fi
     else < A : Node | nbsStates : nbsStates(nbState(C,B,passive),
                                             IDIDSTATUSPFUN) >
     fi .
```

If a node receives an acknowledgment for an older message, than it knows that the neighbor has not yet seen the latest message. Thus, the node forward the latest message, either as part of an existing diffusion computation (i.e., the node is still active) or as a newly started local diffusion computation in which the node takes the parent role.

```
*** Extending ongoing diffusion computation
crl [ActiveRecAckOldMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  states :  states(state(C,active),IDSTATUSPFUN),
                  seqNos : seqNos(seqNo(C,I),IDINTPFUN),
                  nbsStates : IDIDSTATUSPFUN >
     (ack M To A From B Src C)
   =>
     < A : Node | nbsStates : assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                   C, B, active) >
     (msg I To B From A Src C)
   if I > M .


*** Start of new local diffusion computation
crl [PassiveRecAckOldMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  states : states(state(C,passive),IDSTATUSPFUN),
                  parents : IDIDPFUN,
                  seqNos : seqNos(seqNo(C,I),IDINTPFUN),
                  nbsStates : IDIDSTATUSPFUN >
     (ack M To A From B Src C)
   =>
     < A : Node | states : assignIdStatusPFun(IDSTATUSPFUN, C, active),
                  parents : assignIdIdPFun(IDIDPFUN, C, A),
                  nbsStates : assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                   C, B, active) >
     (msg I To B From A Src C)
    if I > M .
```

Link addition and deletion

We treat upcoming or failing links between two nodes `A` and `B` by creating two new, directed messages.

```
  rl [Link] : newlink(A,B) => (linkUpFrom A To B) (linkUpFrom B To A) .
```

The following messages are defined to handle upcoming and failing links.

```
msg newlink  : Oid Oid -> Msg .
msg linkUpFrom_To_ : Oid Oid -> Msg .
msg forSrc_LinkUp__ : Oid Oid Oid -> Msg .
msg failure : Oid Oid -> Msg .
msg linkDownFrom_To_ : Oid Oid -> Msg .
msg forSrc_LinkDown__ : Oid Oid Oid -> Msg .
```

Since both nodes might have several current messages for different sources, an appropriate diffusion computation must be started for each source. Thus, `linkUpFrom A To B` is duplicated to several `(forSrc X LinkUp A B)` messages.

The pseudocode for establishing a link from i to b for a source node j for which i has a defined value SQ^i_j is as follows:

if $ST^i_j = \mathit{active}$ and SQ^i_j is defined
  $N^i := N^i + b$
  $ST^i_{jb} := \mathit{active}$
  send message SQ^i_j for source j to b
  if ST^i_b is not defined
    $ST^i_b := \mathit{passive}$ /* initialization
  if SQ^i_b is not defined
    $SQ^i_b := 0$ /* initialization
  if ST^i_{bk} not defined
    $ST^i_{bk} := \mathit{passive}$ for all $k \in N^i + b$
if $ST^i_j = \mathit{passive}$ and SQ^i_j is defined
  $N^i := N^i + b$
  $ST^i_j := \mathit{active}$
  $s^i_j := i$
  $ST^i_{jb} := \mathit{active}$
  send message SQ^i_j for source j to b
  if ST^i_b is not defined
    $ST^i_b := \mathit{passive}$ /* initialization
  if SQ^i_b is not defined
    $SQ^i_b := 0$ /* initialization
  if ST^i_{bk} not defined
    $ST^i_{bk} := \mathit{passive}$ for all $k \in N^i + b$ /* initialization

The new neighbor is included in the neighbor set and the state with respect to the new neigbhor is active since the latest message is forwarded. An active node does the forwarding in the context of its current computation, a passive node starts a new, local diffusion computation and plays the parent role in this computation. The attributes for the new neighbor b are initialized.

```
crl [ActiveLink] :
      (forSrc A LinkUp B C)
      < B : Node | nbs : OIDSET,
                   states : states(state(A,active),IDSTATUSPFUN),
                   seqNos : seqNos(seqNo(A,M),IDINTPFUN),
                   nbsStates : IDIDSTATUSPFUN >
   =>
      < B : Node | nbs : nbs(C,OIDSET),
                   states : assignIdStatusPFunNoOverriding(
                              states(state(A,active),IDSTATUSPFUN),
                              C, passive),
                   seqNos : assignIdIntPFunNoOverriding(
                              seqNos(seqNo(A,M),IDINTPFUN),
                              C, 0),
                   nbsStates : assignAllIdIdStatusPFunNoOverriding(
                                 assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                      A, C, active),
                                 C, nbs(C,OIDSET), passive) >
       (msg M To C From B Src A)
   if M > 0 .

rl [PassiveLink] :
     (forSrc A LinkUp B C)
     < B : Node | nbs : OIDSET,
                  states : states(state(A,passive),IDSTATUSPFUN),
                  parents : IDIDPFUN,
                  seqNos : seqNos(seqNo(A,M),IDINTPFUN),
                  nbsStates : IDIDSTATUSPFUN >
   =>
   if M > 0
   then < B : Node | nbs : nbs(C,OIDSET),
                     states : assignIdStatusPFunNoOverriding(
                                states(state(A,active),IDSTATUSPFUN),
                                C, passive),
                     parents : assignIdIdPFun(IDIDPFUN, A, B),
                     seqNos : assignIdIntPFunNoOverriding(
                                seqNos(seqNo(A,M),IDINTPFUN),
                                C, 0),
                     nbsStates : assignAllIdIdStatusPFunNoOverriding(
                                   assignIdIdStatusPFun(
                                        IDIDSTATUSPFUN, A, C, active),
                                   C, nbs(C,OIDSET), passive) >
          (msg M To C From B Src A)
```

```
else  < B : Node | nbs : nbs(C,OIDSET),
                   states : assignIdStatusPFunNoOverriding(
                               states(state(A,passive),IDSTATUSPFUN),
                               C, passive),
                   seqNos : assignIdIntPFunNoOverriding(
                               seqNos(seqNo(A,M),IDINTPFUN), C, 0),
                   nbsStates : assignAllIdIdStatusPFunNoOverriding(
                                  assignIdIdStatusPFun(
                                     IDIDSTATUSPFUN, A, C, passive),
                                  C, nbs(C,OIDSET), passive) >
fi .
```

Similarly to new links, link deletion is also bidirectional and is duplicated for each source node.

The pseudocode for failing link from i to b for all source nodes j for which i has a defined value SQ^i_j is as follows:

if $ST^i_j = \mathit{active}$ and s^i_j is defined
  if $ST^i_{jk} = \mathit{passive}$ or not defined for all $k \in (N^i - s^i_j), k \neq b$
    $N^i := N^i - b$
    $ST^i_j := \mathit{passive}$
    if $s^i_j = b$
      delete it from parent-set
    if ST^i_{jb} is defined
      delete if from neighbor-status-set
    if $s^i_j \neq i$
      ack_m for source j to s^i_j
  if $ST^i_{jk} = \mathit{active}$ for at least one $k \in (N^i - s^i_j)$, $k \neq b$
    $N^i := N^i - b$
    if $s^i_j = b$
      delete it from parent-set
    if ST^i_{jb} is defined
      delete if from neighbor-status-set
if $ST^i_j = \mathit{passive}$
  $N^i := N^i - b$
  if $s^i_j = b$
    delete it from parent-set
  if ST^i_{jb} is defined
    delete if from neighbor-status-set
if ST^i_j is not defined
  $N^i := N^i - b$
  if $s^i_j = b$
    delete it from parent-set

if ST^{i}_{jb} is defined
delete if from neighbor-status-set

When a link goes down the active node deletes corresponding entries from all its attributes. Thus, an active node no longer waits for an acknowledgement. If an active node has received ackowledgments from all other neigbors, then it acknowledges its parent.

```
crl [ActiveFailure] :
    (forSrc A LinkDown B C)
    < B : Node | nbs : OIDSET,
                 states :  states(state(A,active),IDSTATUSPFUN),
                 parents : parents(parent(A,X), IDIDPFUN),
                 seqNos : seqNos(seqNo(A,M),IDINTPFUN),
                 nbsStates : IDIDSTATUSPFUN >
   => if  allNbsPassiveWRTSrcUnlessParent(
            deleteIdIdStatusPFunAsNeighborForSource(IDIDSTATUSPFUN, C, A),
            A, X)
         == true
      then (if X =/= B
           then  < B : Node | nbs : deleteFromSet(C,OIDSET),
                              states : states(state(A,passive),
                                              IDSTATUSPFUN),
                              parents : deleteIdIdPFunAsParentForSource(
                                             IDIDPFUN, C, A),
                              nbsStates :
                                 deleteIdIdStatusPFunAsNeighborForSource(
                                                    IDIDSTATUSPFUN, C, A) >
                 (ack M To X From B Src A)
            else < B : Node | nbs : deleteFromSet(C,OIDSET),
                              states : states(state(A,passive),IDSTATUSPFUN),

                              parents : deleteIdIdPFunAsParentForSource(
                                               IDIDPFUN, C, A),
                              nbsStates :
                                 deleteIdIdStatusPFunAsNeighborForSource(
                                                    IDIDSTATUSPFUN, C, A) >
            fi)
      else  < B : Node | nbs : deleteFromSet(C,OIDSET),
                         parents : deleteIdIdPFunAsParentForSource(
                                                    IDIDPFUN, C, A),
                         nbsStates :
                             deleteIdIdStatusPFunAsNeighborForSource(
                                                IDIDSTATUSPFUN, C, A) >
      fi
   if M > 0 .
```

If a link from a passive node i to a node b fails, then i deletes all the information about b in its attributes. The case in which i's state with respect to the source is not defined has to be treated with an extra rule in Maude.

```
  rl [PassiveFailure] :
      (forSrc A LinkDown B C)
      < B : Node | nbs : OIDSET,
                   states :  states(state(A,passive),IDSTATUSPFUN),
                   parents : IDIDPFUN,
                   nbsStates : IDIDSTATUSPFUN >
     =>
       < B : Node | nbs : deleteFromSet(C,OIDSET),
                    parents : deleteIdIdPFunAsParentForSource(IDIDPFUN, C, A),
                    nbsStates :
                         deleteAllIdIdStatusPFunAsNeighborForSource(
                                                IDIDSTATUSPFUN, C, A) > .


  crl [FailureNoSrc] :
  crl [FailureNoSrc] :
      (forSrc A LinkDown B C)
      < B : Node | nbs : OIDSET,
                   parents : IDIDPFUN,
                   nbsStates : IDIDSTATUSPFUN >
     =>
       < B : Node | nbs : deleteFromSet(C,OIDSET),
                    parents : deleteIdIdPFunAsParentForSource(IDIDPFUN, C, A),

                    nbsStates : deleteAllIdIdStatusPFunAsNeighborForSource(IDID
STATUSPFUN, C, A) >
     if not(inIdStatusPFun(A,IDSTATUSPFUN)) .
```

4.2 Analysis of dynamic RBP

The effort of formally specifying RBP for dynamic networks using the executable specification language Maude brought several weaknesses of the original pseudocode [6] to the light. Nevertheless, the analysis is not yet finished. At the current state the following results were gained.

The pseudocode on which we based our first specification of the dynamic RBP was incomplete. In several places it was not clear what are the assumptions about node attributes. A description of initial state was missing, so was the termination condition for the protocol. Moreover, the original pseudocode was incomplete with respect to how attributes are updated. Some cases were left out in the original specification.

We propose the following initialization condition. Each network node i must satisfy the following initial state.

$ST^i_j :=$ *passive* for all $j \in (N^i + i)$
$ST^i_{jk} :=$ *passive* for all $k \in N^i$ and all $j \in (N^i + i)$
$SQ^i_j := 0$ for all $j \in (N^i + i)$

All other fields are undefined.

The protocol terminates for a source node j when j receives acknowledgements from all its neighbors $k \in N^j$.

The latest implementation of RBP extends and improves the original pseudocode with respect to the following aspects:

Receiving a message: Our implementation considers the case where node i receives a message from source j with number m, but there is no sequence number stored for j. This case is reflected in the pseudocode in the case "if ($SQ^i_j < m$ or SQ^i_j is not defined)". Whenever a node is linked to a new node or a partition of the network, it will get messages from source nodes which are unknown to the node.

Receiving an acknowledgement: In the following we describe the changes to the original pseudocode.

- The case "$SQ^i_j = m$ and $ST^i_j =$ *active*" has been extended by the further condition "and ST^i_{jq} is defined (either passive or active)". This condition makes explicit an implicit assumption, that is, only acknowledgements from neighbor nodes are accepted to which messages have been sent.
- In the case "$SQ^i_j = m$ and $ST^i_j =$ *active*" we explicitly distinguished between the cases where the parent is defined or not ("if s^i_j is defined" and "if s^i_j is undefined"). The original pseudocode compares s^i_j and i and implicitly assumes that, if s^i_j is not defined, then the test fails. We take into account that an implementation requires a more careful treatment of undefined values.
- The original case "if $ST^i_{jk} =$ *passive* for all $k \in (N^i - (s^i_j + q))$" has been extended by "or not defined". Again, undefined values must be treated accordingly in a fully typed language as Maude is (and as most programming languages are). Thus, any implementation would need to be explicit about the treatment of undefined values. Our Maude implementation is abstract in the sense that we can cover in the corresponding rule both cases ($ST^i_{jk} =$ *passive* or not defined) without defining in detail what an undefined value is. We define an operator `allNbsPassiveWRTSrcUnlessParent` which only considers thoses entries in the attribute `nbsStates` which are defined for source j.

Link addition: • The code for a new link between nodes i and b has to be applied for several source nodes. But the original pseudocode does not formally define how the source nodes are derived. Our Maude specification clarifies that a new link has to be understood in a bidirectional way, and, moreover, the given pseudocode has to be executed for every source node for which i or b have a defined value SQ^i_j, that is, for every source node of which i or b have the latest message. Our solution proposes multi-sets of messages and appropriate rewrite rules to duplicate the task of establishing the link between two nodes for every source node and we formally define how the source nodes in questions are extracted from the states of node i and b (see definitions for `multiLinkUp` and rule `[MultiLinkUp]`).

- The statement to add the new node to the neighbor set was missing in the original pseudocode: $N^i := N^i + b$.
- The attributes for a new neighbor have to be initialized in an appropriate way. There were no information about this in the original pseudocode. We proposed the following updates which correspond to the initial state as explained earlier.

  if ST^i_b is not defined
  $\quad ST^i_b := \mathit{passive}$ /* initialization
  if SQ^i_b is not defined
  $\quad SQ^i_b := 0$ /* initialization
  if ST^i_{bk} not defined
  $\quad ST^i_{bk} := \mathit{passive}$ for all $k \in N^i + b$

Link deletion: • Analogously to link addition, we implemented link deletion in a two-way style and defined how the source nodes for which a link has to be deleted can be determined from a current state (see (see definitions for `multiLinkDown` and rule `[MultiLinkDown]`).

- The case "all neighbors are passive" has been extended in order to deal with undefined values ("if ST^i_{jk} = passive or not defined for all $k \in N^i, k \neq b$"). This makes explicit the appropriate treatment of (partial) attributes.
- The originally proposed pseudocode does not handle the update of the attributes s^i_j (`parents`) and ST^i_{jk} `nbsStates`). Parts of the following code have been added in several places of the pseudocode which deals with link deletion:

  if $s^i_j = b$
  $\quad$ delete it from parent-set
  if ST^i_{jb} is defined
  $\quad$ delete if from neighbor-status-set

We have some further suggestions concerning other cases in the pseudocode which we have not implemented yet. We are currently discussing those suggestions.

Receiving an acknowledgement:
When an active node receives an acknowledgment of the current message number, but no parent is defined we suggest the following algorithm: The case of an undefined parent

if $SQ_j^i = m$ and $ST_j^i = active$ and ST_{jq}^i is defined (either passive or active)
  if s_j^i is undefined
    if $ST_{jk}^i = passive$ or not defined for all $k \in (N^i - q)$
      $ST_j^i := passive$
      $ST_{jq}^i := passive$
    if $ST_{jk}^i = active$ for at least one $k \in (N^i - q)$
      $ST_{jq}^i := passive$

Further investigations are necessary to decide whether this case can practically occur. If not, then it is an example of an interesting invariant property of the protocol. In either case a more detailed study is necessary. A similar argument applies for the link deletion.

Link deletion:
The original pseudocode does not handle the following case:

if $ST_j^i =$ active and s_j^i is not defined
  if $ST_{jk}^i = passive$ or not defined for all $k \in N^i, k \neq b$
    $N^i := N^i - b$
    $ST_j^i := passive$
    if ST_{jb}^i is defined
      delete if from neighbor-status-set
  if $ST_{jk}^i = active$ for at least one $k \in N^i, k \neq b$
    $N^i := N^i - b$
    if ST_{jb}^i is defined
      delete if from neighbor-status-set

Again, it is not clear whether a state is possible where an active node receives an acknowledgement for a source node for which it has no parent. As mentioned before, our proposed algorithm might be a solution, but further discussions are still necessary. If it turns out that such a state can never occure, then we should be able to proof this invariant formally.

The analysis is not finished yet. Our current implementation works fine for several test cases using a default strategy. But abnormal behavior appears when we run the protocol with an exhaustive search strategy which traverses all possible behaviors.

For example, given a test scenario with three nodes a, b, and c where b has the neighbors a and c, a is a source nodes which sends message number one and the link between a and b breaks down. Running the protocol with this initial configuration using an exhaustive search strategy delivers three different states of which one is a correct state, the second one reveals an undesired behavior and the third one showed an error in the original pseudocode which has been corrected in the current version.

The first resulting state is the one where the link between a and b goes down before a could send the mesage. The resulting situation is a network of three nodes a, b, and c, where only b and c are connected and no message has been sent. This is a correct final state.

In the second final state node a sends the message 1 to node b, but the link goes down before the message could be received (note that we assume asynchronous communication for distributed networks). Thus, the message stays in the configuration without ever being received by b. This is due to the fact, that the algorithm does not say anything about what needs to be done with messages which are floating in the network-channels when one of these channels breaks. The obvious solution is to delete all messages with the link. But we have to make sure that all actions are performed in an atomic way. Otherwise a new link may come up and remaining messages may interfer with this new link.

In third final state node a sends message 1 to b, b receives the message and forwards it to c. Then the link goes down. In the original pseudocode, c was not able to receive the forwarded message 1 because the comparison with the latest sequence number would always fail since c has no sequence number for a. We proposed a solution to this problem by extending the pseudocode such that a node can receive a message from a source node for which it does not have a latest sequence number. This case is handled analogously to the case where a newer message is received.

Further test cases in which link additions and deletions are combined with sending one or more messages are currently under investigation.

References

[1] M. Clavel. *Reflection in general logics, rewriting logic, and Maude.* PhD thesis, University of Navarre, 1998.

[2] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier, 1998.

[3] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic.* Computer Science Laboratory, SRI International, Menlo Park, 1999.

[4] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

[5] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998.

[6] J. J. Garcia-Luna. Reliable broadcasting in computer networks, 1998. Manuscript, UC Santa Cruz.

[7] Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, pages 309–339. DIMACS Series, Vol. 18, American Mathematical Society, 1994.

[8] P. M. Merlin and A. Segall. A failsafe distributed routing protocol. *IEEE Trans. Commun.*, 27(9):1280–1288, 1979.

[9] José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.

[10] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[11] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[12] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proc. Concur'96*, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[13] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997.* Springer-Verlag, 1998.

[14] J. Meseguer and C. Talcott. A Partial Order Event Model for Concurrent Objects, 1998. submitted for publication.

[15] J. Meseguer and T. Winkler. Parallel programming in Maude. In *Research Directions in High-Level Parallel Programming Languages*, number 574 in Lecture Notes in Computer Science, pages 253–293. Springer-Verlag, 1992.

[16] A. Segall. Distributed network protocols. *IEEE Trans. Info. Theory*, 29(1):25–35, 1983.

[17] A. Segall and B. Awerbuch. A reliable broadcast protocol. *IEEE Trans. Commun.*, 31(7):896–901, 1983.

A The complete final specification for static networks

This appendix presents the executable Full Maude [3] specification of the final version of the protocol for static networks, followed by the result of executing some test cases using Maude's default interpreter.

The specification is given as follows.

```
in full-maude.maude

loop init .

(omod RBP is
sorts  DefOid OidSet MessageContent SrcOid DefSrcOid .
subsort Oid < DefOid .             *** Oid with default value
subsort Oid < OidSet .
subsort Oid < SrcOid .             *** Oid with value source
subsort SrcOid < DefSrcOid .
subsort DefOid < DefSrcOid .

class  Node | nbs : OidSet, parent : DefSrcOid, recdMsg : OidSet,
              recdAck : OidSet .

*** messages:
msg to_broadcast_ : Oid MessageContent -> Msg .
msgs msg_from_to_ ackParent_from_to_ ack_from_to_ :
                    MessageContent Oid Oid -> Msg .

*** multimessage-declaration:
op multimsg_from_to_ : MessageContent Oid OidSet -> Configuration .

op m : -> MessageContent .         *** "message content"
ops a b c d e f : -> Oid .         *** Six objects in the first example.
op noOid : -> DefOid .
op source : -> SrcOid .

*** [multi]set constructors:
```

```
op nil : -> OidSet .
op __ : OidSet OidSet -> OidSet [assoc comm id: nil prec 15] .

*** Some set-operations:
op in : Oid OidSet -> Bool .
op subseteq : OidSet OidSet -> Bool .
op _-_ : OidSet Oid -> OidSet .       *** Set minus one Oid!
op minusParent : OidSet SrcOid -> OidSet . *** Set minus a parent.

*** The following are initial configurations for testing:
ops test3 test4 test6 : -> Configuration .

vars A B C : Oid . vars N N' N'' : OidSet .
var M : MessageContent . var D : DefOid .
var S : SrcOid . var O : DefSrcOid .


*** Functions on sets:
eq A A = A .     *** This equation makes the multiset into a set.
eq in(A, B N) = A == B or in(A,N) .
eq in(A,nil) = false .
eq subseteq(A N ,N') = in(A,N') and subseteq(N,N') .
eq subseteq(nil,N') = true .
eq (A N) - A = N - A .
ceq N - A = N if not in(A,N) .
eq minusParent(N,A) = N - A .
eq minusParent(N,source) = N .


*** multimessage definition:
eq multimsg M from A to (B N) =
             (msg M from A to B) (multimsg M from A to (N - B)) .

eq multimsg M from A to nil = none .

*** Some initial configurations for testing:       *** TEST
***                                                *** TEST
eq test3 =
   (to a broadcast m)
   < a : Node | nbs : b c, parent : noOid, recdMsg : nil, recdAck : nil >
   < b : Node | nbs : a c, parent : noOid, recdMsg : nil, recdAck : nil >
   < c : Node | nbs : a b, parent : noOid, recdMsg : nil, recdAck : nil > .

eq test4 =
  (to a broadcast m)
  < a : Node | nbs : b c, parent : noOid, recdMsg : nil, recdAck : nil >
```

```
  < b : Node | nbs : a c, parent : noOid, recdMsg : nil, recdAck : nil >
  < c : Node | nbs : b a d, parent : noOid, recdMsg : nil, recdAck : nil >
  < d : Node | nbs : c, parent : noOid, recdMsg : nil, recdAck : nil > .

eq test6 =
  (to a broadcast m)
  < a : Node | nbs : b c f, parent : noOid, recdMsg : nil, recdAck : nil >
  < b : Node | nbs : a e f, parent : noOid, recdMsg : nil, recdAck : nil >
  < c : Node | nbs : f a d, parent : noOid, recdMsg : nil, recdAck : nil >
  < d : Node | nbs : c e, parent : noOid, recdMsg : nil, recdAck : nil >
  < e : Node | nbs : b d, parent : noOid, recdMsg : nil, recdAck : nil >
  < f : Node | nbs : a b c, parent : noOid, recdMsg : nil, recdAck : nil > .

*** Start the broadcasting cycle:
rl [Start] :
   (to A broadcast m)
   < A : Node | nbs : N, parent : noOid, recdMsg : nil, recdAck : nil > =>
   < A : Node | parent : source > (multimsg m from A to N) .

*** Stays at it is: no siblings => noAck of siblings!
rl [RecMessage1] :
   (msg m from A to B)
   < B : Node | nbs : A N, parent : noOid, recdMsg : nil,
              recdAck : nil >
    => < B : Node | parent : A > (multimsg m from B to (N - A)) .

*** If we've already receive a message,
*** we acknowledge the sender directly:
rl [RecMessage2] :
   (msg m from A to B) < B : Node | parent : S, recdMsg : N > =>
   < B : Node | recdMsg : A N > (ack m from B to A) .

*** Acknowledge the parent when received ack/ackParent from
*** all other neighbors:
rl [AckParent] :
   < A : Node | nbs : B N, parent : B, recdAck : N > =>
   < A : Node | parent : noOid, recdMsg : nil, recdAck : nil >
   (ackParent m from A to B) .

rl [RecAckParent] :
   (ackParent m from A to B) < B : Node | parent : S, recdAck : N > =>
   < B : Node | recdAck : A N > .

*** Receiving acknowledgement from a sibling is straigh-forward:
rl [RecAckSibling] :
   (ack m from A to B)
```

```
   < B : Node | parent : S, recdMsg : A N, recdAck : N' > =>
   < B : Node | recdAck : A N' > .

*** Reset the source when cycle finished:
rl [ResetSource] :
   < A : Node | nbs : N, parent : source, recdAck : N > =>
   < A : Node | parent : noOid, recdMsg : nil, recdAck : nil > .

endom)
```

Assuming that the code above is located in a file `omodrbp.maude`, the following is a transcript of a Full Maude session which uses Maude's default interpreter to simulate some run from the test networks `test3`, `test4`, and `test6`.

```
Maude>  in omodrbp.maude
Reading in file: "full-maude.maude"
==========================================
fmod FULL-MAUDE
Done reading in file: "full-maude.maude"


Introduced module: RBP
Maude>  (rew test3 .)

Rewrite in RBP : test3 .
Result Configuration :
< a : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : b c >
< b : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : a c >
< c : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : a b >
Maude>  (rew test4 .)

Rewrite in RBP : test4 .
Result Configuration :
< a : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : b c >
< b : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : a c >
< c : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : a b d >
< d : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : c >
Maude>  (rew test6 .)

Rewrite in RBP : test6 .
Result Configuration :
< e : Node | recdAck : nil , recdMsg : nil , parent : noOid ,
             nbs : b d >
< a : Node | recdAck : nil , recdMsg : nil , parent : noOid ,
             nbs : b c f >
< b : Node | recdAck : nil , recdMsg : nil , parent : noOid ,
```

```
             nbs : e a f >
< c : Node | recdAck : nil , recdMsg : nil , parent : noOid ,
             nbs : a d f >
< d : Node | recdAck : nil , recdMsg : nil , parent : noOid ,
             nbs : e c >
< f : Node | recdAck : nil , recdMsg : nil , parent : noOid ,
             nbs : a b c >
Maude>
```

B The strategy language

In this appendix we present the strategy language which can be used to find all the irreducible terms reachable from a given term. In particular, we use the meta-level of Maude to define a function

```
op irreducibleforms : Module Term QidList -> TermSet
```

which takes (the meta-representation of) a rewrite module `M`, (the meta-representation of) a term `t` in `M`, and a list `L` of (the meta-representations of) names of rules in `M` and returns the set of (the meta-representations) of all terms reachable from `t` in `M` using the rules in `M` named in `L`, and which cannot be further be rewritten using the rules in `L`

We then show a transcript of a Full Maude session finding all irreducible states reachable from our test configurations with 3 and 4 nodes.

The strategy language is defined as follows.

```
(fmod META-RBP is

protecting META-LEVEL[RBP] .

*** Now comes the strategy that checks all rewrites:
sorts StrategyExpression Strategy .
sorts TermSet SetOfTermLists .
subsort Term < TermSet .
subsort TermList < SetOfTermLists .

op apply : Qid -> Strategy .
op applySomewhere : Qid -> Strategy .
op applySomewhereSubst : Qid Substitution -> Strategy .

op first : ResultPair -> Term .
op use : Module Term Qid Substitution -> Term .
op uselist : Module TermList Qid Substitution -> TermList .
```

```
*** Somewhat trickier ones to find all possible rewrites
*** using one rule at the top of the term:
op emptyTermSet : -> TermSet .
op set : TermSet TermSet -> TermSet [assoc comm id: emptyTermSet] .
op applyAll : Module Term Qid -> TermSet .
op applyAllHelp : Module Term Qid MachineInt -> TermSet .


var M : Module . var T : Term . var RL : Qid . var SUBST : Substitution .
var TL : TermList . var F CONST SORT : Qid . var S S' : Strategy .
var N : MachineInt .

eq first({T,SUBST}) = T .

eq use(M,F,RL,SUBST) = first(meta-apply(M,F,RL,SUBST,0)) .
eq use(M,{CONST}SORT,RL,SUBST) = first(meta-apply(M,{CONST}SORT,RL,SUBST,0)) .
eq use(M,F[TL],RL,SUBST) =
          if first(meta-apply(M,F[TL],RL,SUBST,0)) == error*
          then (if uselist(M,(TL),RL,SUBST) == error* then error*
                else F[uselist(M,(TL),RL,SUBST)] fi)
          else first(meta-apply(M,F[TL],RL,SUBST,0)) fi .

eq uselist(M,T,RL,SUBST) = use(M,T,RL,SUBST) .
eq uselist(M,(T,TL),RL,SUBST) = if use(M,T,RL,SUBST) == error* then
                                  (if uselist(M,TL,RL,SUBST) == error*
                                   then error*
                                   else (T,uselist(M,TL,RL,SUBST)) fi)
                                else (use(M,T,RL,SUBST),TL) fi .

*** This finds all the rewrites at top position in current term and rule:
eq applyAll(M,T,RL) = applyAllHelp(M,T,RL,0) .
eq applyAllHelp(M,T,RL,N) =
           if first(meta-apply(M,T,RL,none,N)) == error*
           then emptyTermSet
           else set(first(meta-apply(M,T,RL,none,N)),
                    applyAllHelp(M,T,RL,N + 1)) fi .

eq set(T,T) = T .
*** We now define the list of terms that can be reached by one rewrite step
*** [using a particular rule] that can be reached from some term.

op allRewrites : Module Term Qid -> TermSet .    *** all one-step rewrites
                                                 *** from current term
                                                 *** using a rule.
op emptySTL : -> SetOfTermLists .
```

```
op setTL : SetOfTermLists SetOfTermLists -> SetOfTermLists [assoc comm id: emptySTL] .
op allRewritesList : Module TermList Qid -> SetOfTermLists .
op useOn : Qid SetOfTermLists -> TermSet .
op tosetTL : TermSet -> SetOfTermLists .  *** Conversion.
op appfirst : TermSet SetOfTermLists -> SetOfTermLists .

vars TS TS' : TermSet .
vars STL STL' : SetOfTermLists .


eq setTL(TL,TL) = TL .

eq allRewrites(M,F,RL) = applyAll(M,F,RL) .
eq allRewrites(M,{CONST}SORT,RL) = applyAll(M,{CONST}SORT,RL) .
eq allRewrites(M,F[TL],RL) = set(applyAll(M,F[TL],RL),
                                 useOn(F,allRewritesList(M,TL,RL))) .
eq useOn(F,emptySTL) = emptyTermSet .
eq useOn(F,TL) = F[TL] .
eq useOn(F,setTL((TL),(STL))) = set(F[TL],useOn(F,STL)) .
*** strictly speaking, there is the need to add a meta-reduce
*** to F[TL] in the right-handside of the last equation, but since
*** it is unnecessary for configurations, it is omitted for
*** efficiency purposes

eq allRewritesList(M,T,RL) =   tosetTL(allRewrites(M,T,RL)) .
ceq allRewritesList(M,(T,TL),RL) =
       setTL(appfirst(allRewrites(M,T,RL),(TL)),
             appfirst(T,allRewritesList(M,TL,RL))) if TL =/= nil .

*** This is rather the "product" between the set(list)
*** of first terms and the set (list) of termlists.
*** e. g., "[t;u;v] appfirst [(t1,t2);(t5,t6,t7)]" equals
*** [(t,t1,t2);(u,t1,t2);(v,t1,t2);(t,t5,t6,t7);(u,t5,t6,t7);(v,t5,t6,t7)]
eq appfirst(TS,emptySTL) = emptySTL .
eq appfirst(emptyTermSet,STL) = emptySTL .
ceq appfirst(T,setTL(TL,STL)) = setTL((T,TL), appfirst(T,STL))
                                if STL =/= emptySTL .
eq appfirst(T,TL) = (T,TL) .
ceq appfirst(set(T,TS),STL) = setTL(appfirst(T,STL), appfirst(TS,STL))
                              if TS =/= emptyTermSet .

*** Takes a TermSet into a SetOfTermLists (where each list is
*** a singleton).
eq tosetTL(emptyTermSet) = emptySTL .
eq tosetTL(T) = T .
eq tosetTL(set(T,TS)) = setTL(T,tosetTL(TS)) .
```

```
*** allRewrites is extended to sets of rules:
op allRewrites : Module Term QidSet -> TermSet .
var QS : QidSet .
eq allRewrites(M,T,none) = emptyTermSet .
ceq allRewrites(M,T,(RL ; QS)) = set(allRewrites(M,T,RL),
                                     allRewrites(M,T,QS)) if QS =/= none .

*** Find irreducible forms:
op irreducibleforms : Module Term QidSet -> TermSet .
*** This calls irf(Module,termsToCheckInThisRound,TermsCheckedInThisRound,QS)
op irf : Module TermSet TermSet QidSet -> TermSet .

*** Trigger:
eq irreducibleforms(M,T,QS) = irf(M,meta-reduce(M,T),emptyTermSet,QS) .

*** End of process:
eq irf(M,emptyTermSet,emptyTermSet,QS) = emptyTermSet .

*** Next round:
eq irf(M,emptyTermSet,set(T,TS),QS) = irf(M,set(T,TS),emptyTermSet,QS) .

*** Find "next-round-form" of current term, if any,
*** else add it to the set of irreducible forms:
ceq irf(M,set(T,TS),TS',QS) = set(T,irf(M,TS,TS',QS))
                              if allRewrites(M,T,QS) == emptyTermSet .
ceq irf(M,set(T,TS),TS',QS) = irf(M,TS,set(allRewrites(M,T,QS),TS'),QS)
                              if allRewrites(M,T,QS) =/= emptyTermSet .

***
*** Specific constants for this example:
***
op allstaticrules : -> QidSet .   *** all rules in the static spec
eq allstaticrules =  ('Start ; 'RecMessage1 ; 'RecMessage2 ; 'AckParent ;
                      'RecAckParent ; 'RecAckSibling ; 'ResetSource ) .

op alldynamicrules : -> QidSet .  *** all rules in dynamic spec
eq alldynamicrules =  ('Send ; 'RepeatRecCurrentMsg ; 'RecNewMsg ;
                        'RecNewMsgLeaf ; 'ActiveRecOldMsg ;
                        'PassiveRecOldMsg ; 'RecAckNewerMsg ;
                        'PassiveRecAckCurrentMsg ; 'ActiveRecAckCurrentMsg ;
                        'ActiveRecAckOldMsg ; 'PassiveRecAckOldMsg ;
                        'Link ; 'MultiLinkUp ; 'ActiveLink ; 'PassiveLink ;
                        'Failure ; 'MultiLinkDown ; 'ActiveFailure ;
                        'PassiveFailure ; 'FailureNoSrc ) .
```

```
endfm)
```

Assuming that the strategy language above is in a file `meta-rbp.maude`, the following is a transcript of a Full Maude session finding all the irreducble terms reachable from initial states `test3` and `test4`.

```
Maude>  in omodrbp.maude  *** read main module and initialize Full Maude
Reading in file: "full-maude.maude"
==========================================
mod FULL-MAUDE
Done reading in file: "full-maude.maude"


Introduced module: RBP
Maude>  in meta-rbp.maude  *** read in strategy language

Introduced module: META-RBP
Maude>  (red irreducibleforms(RBP, {'test3}'Configuration,
allstaticrules) .)

Reduce in META-RBP : irreducibleforms ( RBP , { 'test3 }
'Configuration , allstaticrules ) .
Result Term :
'__ [ '<_:_|_> [ { 'a } 'Oid , { 'Node } 'Node ,
'_`,_ [ 'recdAck`:_ [ { 'nil } 'OidSet ] ,
'recdMsg`:_ [ { 'nil } 'OidSet ] , 'parent`:_ [ { 'noOid }
'DefOid ] , 'nbs`:_ [ '__ [ { 'b } 'Oid , { 'c } 'Oid ] ] ] ] ,
'<_:_|_> [ { 'b } 'Oid , { 'Node } 'Node , '_`,_ [
'recdAck`:_ [ { 'nil } 'OidSet ] , 'recdMsg`:_ [ { 'nil }
'OidSet ] , 'parent`:_ [ { 'noOid } 'DefOid ] , 'nbs`:_ [ '__
[ { 'a } 'Oid , { 'c } 'Oid ] ] ] ] , '<_:_|_> [ { 'c } 'Oid ,
{ 'Node } 'Node , '_`,_ [ 'recdAck`:_ [ { 'nil } 'OidSet ] ,
'recdMsg`:_ [ { 'nil } 'OidSet ] , 'parent`:_ [ { 'noOid }
'DefOid ] , 'nbs`:_ [ '__ [ { 'a } 'Oid , { 'b } 'Oid ] ] ] ] ]
Maude>   (down RBP : red irreducibleforms(RBP,{'test3}'Configuration,
allstaticrules) .)   *** Since only one Term is returned, it can be pretty printed

Result Configuration :
< a : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : b c >
< b : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : a c >
< c : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : a b >
Maude>  (down RBP : red irreducibleforms(RBP, {'test4}'Configuration, all-
staticrules) .)

Result Configuration :
```

```
< a : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : b c >
< b : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : a c >
< c : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : a b d >
< d : Node | recdAck : nil , recdMsg : nil , parent : noOid , nbs : c >
Maude>
```

C The complete specification of the protocol for dynamic topologies

This appendix contains the executable Full Maude specification of the protocol for the dynamic topology, followed by a transcript of the results of executing the specification using Maude's default interpreter and then the strategy which finds all irreducible terms reachable from a given initial state.

The specification is given as follows:

```
*** RBP for dynamic topologies
*** Each node in the system can send a message -> concurrency
*** The correctness criterion is that the latest message from a source
*** node should arrive at all nodes which currently have a path to
*** this message

loop init .

(omod RBP is
  protecting MACHINE-INT .

  sort MachineIntSet .
  subsort MachineInt < MachineIntSet .
  op mtIntSet : -> MachineIntSet .
  op intSet : MachineIntSet MachineIntSet -> MachineIntSet
                                             [assoc comm id: mtIntSet] .

  sort OidSet .
  subsort Oid < OidSet .

  *** sorts for pairs/triple
  *** first parameter is the source node identity
  sort Status .
  sort IdStatusPair .
  sort IdIdPair .
  sort IdIntPair .
  sort IdIdStatusTriple .
```

```
*** sorts for partial functions
sort IdStatusPFun .
subsort IdStatusPair < IdStatusPFun .
sort IdIdPFun .
subsort IdIdPair < IdIdPFun .
sort IdIntPFun .
subsort IdIntPair < IdIntPFun .
sort IdIdStatusPFun .
subsort IdIdStatusTriple < IdIdStatusPFun .

*** operations for class attributes
op active : -> Status .
op passive : -> Status .
op state : Oid Status -> IdStatusPair .
op parent : Oid Oid -> IdIdPair .
op seqNo : Oid MachineInt -> IdIntPair .
op nbState : Oid Oid Status -> IdIdStatusTriple .

*** operations for partial functions
op mtstates : -> IdStatusPFun .
op inIdStatusPFun : Oid IdStatusPFun -> Bool .
op states : IdStatusPFun IdStatusPFun ->  IdStatusPFun
                                          [assoc comm id: mtstates] .
op assignIdStatusPFun : IdStatusPFun  Oid Status ->  IdStatusPFun .
op assignIdStatusPFunNoOverriding : IdStatusPFun  Oid Status
                                    ->  IdStatusPFun .

op mtparents : -> IdIdPFun .
op inIdIdPFun : Oid IdIdPFun -> Bool .
op inIdIdPFunAsParentForSource : Oid Oid IdIdPFun -> Bool .
op parents : IdIdPFun IdIdPFun ->  IdIdPFun
                                   [assoc comm id: mtparents] .
op assignIdIdPFun : IdIdPFun  Oid Oid ->  IdIdPFun .
op deleteIdIdPFunAsParentForSource : IdIdPFun  Oid Oid  ->  IdIdPFun .

op mtseqNos : -> IdIntPFun .
op inIdIntPFun : Oid IdIntPFun -> Bool .
op forSourceSmallerOrNoSeqnoInIdIntPFun : Oid MachineInt IdIntPFun
                                          -> Bool .
op seqNos : IdIntPFun IdIntPFun ->  IdIntPFun
                                    [assoc comm id: mtseqNos] .
op assignIdIntPFun : IdIntPFun  Oid MachineInt ->  IdIntPFun .
op assignIdIntPFunNoOverriding : IdIntPFun  Oid MachineInt ->  IdIntPFun .
op deleteIdIntPFun : IdIntPFun  Oid  ->  IdIntPFun .
op dom : IdIntPFun -> OidSet .
```

```
op mtnbsStates : -> IdIdStatusPFun .
op inIdIdStatusPFun : Oid Oid IdIdStatusPFun -> Bool .
op inIdIdStatusPFunAsNeighborForSource : Oid Oid IdIdStatusPFun -> Bool .
op nbsStates : IdIdStatusPFun IdIdStatusPFun
               ->  IdIdStatusPFun [assoc comm id: mtnbsStates] .
op assignIdIdStatusPFun : IdIdStatusPFun Oid Oid Status ->
                          IdIdStatusPFun .
op assignIdIdStatusPFunNoOverriding : IdIdStatusPFun Oid Oid Status
                                      -> IdIdStatusPFun .
op deleteIdIdStatusPFunAsNeighborForSource : IdIdStatusPFun  Oid Oid
                                             ->  IdIdStatusPFun .
op assignAllIdIdStatusPFun : IdIdStatusPFun Oid OidSet Status
                             -> IdIdStatusPFun .
op assignAllIdIdStatusPFunNoOverriding : IdIdStatusPFun Oid OidSet Status
                                         -> IdIdStatusPFun .
*** this operation takes an Oid and deletes all entries in the partial
*** function where Oid is either in the domain or codomain
op deleteAllIdIdStatusPFunAsNeighborForSource :  IdIdStatusPFun Oid Oid
                                                 -> IdIdStatusPFun .
*** this operation looks up whether all neighbors (except parent node)
*** are passive wrt to the given source node (second parameter)
op allNbsPassiveWRTSrcUnlessParent : IdIdStatusPFun Oid Oid -> Bool .



***
***  NETWORK NODES
***
class Node | nbs : OidSet, states : IdStatusPFun, parents : IdIdPFun,
             seqNos : IdIntPFun, nbsStates : IdIdStatusPFun .


*** messages:
msg msg_To_From_Src_ : MachineInt Oid Oid Oid -> Msg .
msg ack_To_From_Src_ : MachineInt Oid Oid Oid -> Msg .
msg newlink  : Oid Oid -> Msg .
msg linkUpFrom_To_ : Oid Oid -> Msg .
msg forSrc_LinkUp__ : Oid Oid Oid -> Msg .
msg failure : Oid Oid -> Msg .
msg linkDownFrom_To_ : Oid Oid -> Msg .
msg forSrc_LinkDown__ : Oid Oid Oid -> Msg .

*** multimessage-declarations:
msg multimsg_To_From_Src_ : MachineInt OidSet Oid Oid -> Configuration .
msg multiLinkUp_From_To_ : OidSet Oid Oid -> Configuration .
```

```
msg multiLinkDown_From_To_ : OidSet Oid Oid -> Configuration .


ops a b c d e f : -> Oid .       *** Topology with up to six objects


*** Initial configurations for testing
ops test1 test2 test3 : -> Configuration .

*** Set constructors:
op mtnbs : -> OidSet .
op nbs : OidSet OidSet -> OidSet [assoc comm id: mtnbs] .
op inSet : Oid OidSet  -> Bool .
op deleteFromSet : Oid OidSet -> OidSet .

*** Variable declarations:
var STAT1 STAT2 : Status .
var IDSTATUSPFUN : IdStatusPFun .
var IDIDPFUN : IdIdPFun .
var IDINTPFUN : IdIntPFun .
var IDIDSTATUSPFUN : IdIdStatusPFun .
var I J M : MachineInt .
vars A B C D X : Oid .
var OIDSET : OidSet .
var BOOL : Bool .
var CONF : Configuration .

***
*** EQUATIONS FOR DATATYPES (partial functions etc)
***

*** Equations  for IdStatusPFun:
eq inIdStatusPFun(A, states(state(B,STAT1), IDSTATUSPFUN))
  = if A == B
    then true
    else inIdStatusPFun(A, IDSTATUSPFUN)
    fi .

eq inIdStatusPFun(A, mtstates) = false .

eq assignIdStatusPFun(states(state(A,STAT1), IDSTATUSPFUN), A, STAT2)
  = states(state(A,STAT2), IDSTATUSPFUN) .

ceq assignIdStatusPFun(IDSTATUSPFUN, A, STAT1)
  = states(state(A,STAT1), IDSTATUSPFUN)
  if not(inIdStatusPFun(A,IDSTATUSPFUN)) .
```

```
eq assignIdStatusPFunNoOverriding(states(state(A,STAT1), IDSTATUSPFUN),
                                  A, STAT2)
  = states(state(A,STAT1), IDSTATUSPFUN) .

ceq assignIdStatusPFunNoOverriding(IDSTATUSPFUN, A, STAT1)
  = states(state(A,STAT1), IDSTATUSPFUN)
  if not(inIdStatusPFun(A,IDSTATUSPFUN)) .


*** Equations for IdIdPFun:
eq inIdIdPFun(A, parents(parent(B,C), IDIDPFUN))
  = if A == B
    then true
    else inIdIdPFun(A, IDIDPFUN)
    fi .

eq inIdIdPFun(A, mtparents) = false .

eq inIdIdPFunAsParentForSource(A, D, parents(parent(B,C), IDIDPFUN))
  = if (A == C) and (D == B)
    then true
    else inIdIdPFunAsParentForSource(A, D, IDIDPFUN)
    fi .

eq inIdIdPFunAsParentForSource(A, D, mtparents) = false .

eq assignIdIdPFun(parents(parent(A,B), IDIDPFUN), A, C)
  = parents(parent(A,C), IDIDPFUN) .

ceq assignIdIdPFun(IDIDPFUN, A, B)
  = parents(parent(A,B), IDIDPFUN)
  if not(inIdIdPFun(A,IDIDPFUN)) .

eq deleteIdIdPFunAsParentForSource(parents(parent(A,B), IDIDPFUN), B, A)
  = IDIDPFUN .

ceq deleteIdIdPFunAsParentForSource(IDIDPFUN, A, B)
  = IDIDPFUN
  if not(inIdIdPFunAsParentForSource(A,B,IDIDPFUN)) .


*** Equations for IdIntPFun:
eq dom(mtseqNos) = mtnbs .

eq dom(seqNos(seqNo(A,I), IDINTPFUN)) = nbs(A, dom(IDINTPFUN)) .
```

```
eq inIdIntPFun(A, seqNos(seqNo(B,I), IDINTPFUN))
  = if A == B
    then true
    else inIdIntPFun(A, IDINTPFUN)
    fi .

eq inIdIntPFun(A, mtseqNos) = false .

ceq forSourceSmallerOrNoSeqnoInIdIntPFun(A, J, IDINTPFUN) = true
   if not(inIdIntPFun(A, IDINTPFUN)) .

eq forSourceSmallerOrNoSeqnoInIdIntPFun(A, J, seqNos(seqNo(B,I),
                                        IDINTPFUN))
  = if A == B
    then if J > I
         then true
         else false
         fi
    else forSourceSmallerOrNoSeqnoInIdIntPFun(A, J, IDINTPFUN)
    fi .

eq forSourceSmallerOrNoSeqnoInIdIntPFun(A, J, mtseqNos) = true .

eq assignIdIntPFun(seqNos(seqNo(A,I), IDINTPFUN), A, J)
  = seqNos(seqNo(A,J), IDINTPFUN) .

ceq assignIdIntPFun(IDINTPFUN, A, I)
  = seqNos(seqNo(A,I), IDINTPFUN)
  if not(inIdIntPFun(A,IDINTPFUN)) .

eq assignIdIntPFunNoOverriding(seqNos(seqNo(A,I), IDINTPFUN),
                               A, J)
  = seqNos(seqNo(A,I), IDINTPFUN) .

ceq assignIdIntPFunNoOverriding(IDINTPFUN, A, I)
  = seqNos(seqNo(A,I), IDINTPFUN)
  if not(inIdIntPFun(A,IDINTPFUN)) .

eq deleteIdIntPFun(seqNos(seqNo(A,I), IDINTPFUN), A)
  = IDINTPFUN .

ceq deleteIdIntPFun(IDINTPFUN, A)
  = IDINTPFUN
  if not(inIdIntPFun(A,IDINTPFUN)) .
```

```
*** Equations for IdIdStatusPFun:
eq inIdIdStatusPFun(A, B, nbsStates(nbState(C,D,STAT1), IDIDSTATUSPFUN))
  = if (A == C) and (B == D)
    then true
    else inIdIdStatusPFun(A, B, IDIDSTATUSPFUN)
    fi .

eq inIdIdStatusPFun(A, B, mtnbsStates) = false .

eq inIdIdStatusPFunAsNeighborForSource(B,A, nbsStates(nbState(C,D,STAT1),
                                       IDIDSTATUSPFUN))
  = if (A == C) and (B == D)
    then true
    else inIdIdStatusPFunAsNeighborForSource(B, A, IDIDSTATUSPFUN)
    fi .

eq inIdIdStatusPFunAsNeighborForSource(B, A, mtnbsStates) = false .

eq assignIdIdStatusPFun(nbsStates(nbState(A,B,STAT1), IDIDSTATUSPFUN),
                        A, B, STAT2)
  = nbsStates(nbState(A,B,STAT2), IDIDSTATUSPFUN) .

ceq assignIdIdStatusPFun(IDIDSTATUSPFUN, A, B, STAT1)
  = nbsStates(nbState(A,B,STAT1), IDIDSTATUSPFUN)
  if not(inIdIdStatusPFun(A,B,IDIDSTATUSPFUN)) .


eq assignIdIdStatusPFunNoOverriding(nbsStates(nbState(A,B,STAT1),
                                              IDIDSTATUSPFUN),
                                    A, B, STAT2)
  = nbsStates(nbState(A,B,STAT1), IDIDSTATUSPFUN) .

ceq assignIdIdStatusPFunNoOverriding(IDIDSTATUSPFUN, A, B, STAT1)
  = nbsStates(nbState(A,B,STAT1), IDIDSTATUSPFUN)
  if not(inIdIdStatusPFun(A,B,IDIDSTATUSPFUN)) .

eq deleteIdIdStatusPFunAsNeighborForSource(nbsStates(nbState(A,B,STAT1),
                                           IDIDSTATUSPFUN), B, A)
  = IDIDSTATUSPFUN .

ceq deleteIdIdStatusPFunAsNeighborForSource(IDIDSTATUSPFUN, B, A)
  = IDIDSTATUSPFUN
  if not(inIdIdStatusPFun(B,A,IDIDSTATUSPFUN)) .
```

```
*** assign the same status to a set of neighbors (for a given source node)
eq assignAllIdIdStatusPFun(IDIDSTATUSPFUN, A, nbs(B,OIDSET), STAT1)
  = assignAllIdIdStatusPFun(
          assignIdIdStatusPFun(IDIDSTATUSPFUN, A, B, STAT1),
          A, OIDSET, STAT1) .

eq assignAllIdIdStatusPFun(IDIDSTATUSPFUN, A, mtnbs, STAT1)
  = IDIDSTATUSPFUN .

*** assign the same status to a set of neighbors (for a given source
*** node) but only if there is no entry for this source yet
eq assignAllIdIdStatusPFunNoOverriding(IDIDSTATUSPFUN, A,
                                       nbs(B,OIDSET), STAT1)
  = assignAllIdIdStatusPFun(
          assignIdIdStatusPFunNoOverriding(IDIDSTATUSPFUN, A, B, STAT1),
          A, OIDSET, STAT1) .

eq assignAllIdIdStatusPFunNoOverriding(IDIDSTATUSPFUN, A, mtnbs, STAT1)
  = IDIDSTATUSPFUN .

eq deleteAllIdIdStatusPFunAsNeighborForSource(
      nbsStates(nbState(A, B, STAT1),IDIDSTATUSPFUN),
      B, A)
  = deleteAllIdIdStatusPFunAsNeighborForSource(IDIDSTATUSPFUN, B, A) .

ceq deleteAllIdIdStatusPFunAsNeighborForSource(IDIDSTATUSPFUN, B, A)
  = IDIDSTATUSPFUN
  if not(inIdIdStatusPFun(A, B, IDIDSTATUSPFUN)) .

eq allNbsPassiveWRTSrcUnlessParent(nbsStates(nbState(B,C,STAT1),
                                   IDIDSTATUSPFUN), A, X)
  = if A == B
    then if C == X
         then allNbsPassiveWRTSrcUnlessParent(IDIDSTATUSPFUN, A, X)
         else if STAT1  == passive
              then allNbsPassiveWRTSrcUnlessParent(IDIDSTATUSPFUN, A, X)
              else false
              fi
         fi
    else allNbsPassiveWRTSrcUnlessParent(IDIDSTATUSPFUN, A, X)
    fi .

eq allNbsPassiveWRTSrcUnlessParent(mtnbsStates, A, X)  = true .


*** Equations for sets:
```

```
eq nbs(A,A) = A .
eq inSet(A, nbs(B,OIDSET))
  = if A == B
    then true
    else inSet(A, OIDSET)
    fi .

eq inSet(A, mtnbs) = false .

eq deleteFromSet(A, nbs(B,OIDSET))
  = if A == B
    then OIDSET
    else nbs(B,deleteFromSet(A, OIDSET))
    fi .

eq deleteFromSet(A, mtnbs)
  = mtnbs .


*** multimessage definition:
eq multimsg I To A From B Src C = msg I To A From B Src C .
eq multimsg I To mtnbs From B Src C = none .
eq multimsg I To nbs(A,OIDSET) From B Src C
   = (msg I To A From B Src C) (multimsg I To OIDSET From B Src C) .

eq multiLinkUp X From A To B  = forSrc X LinkUp A B .
eq multiLinkUp mtnbs From A To B = none .
eq multiLinkUp nbs(X,OIDSET) From A To B
   = (forSrc X LinkUp A B) (multiLinkUp OIDSET From A To B) .

eq multiLinkDown X From A To B  = forSrc X LinkDown A B .
eq multiLinkDown mtnbs From A To B = none .
eq multiLinkDown nbs(X,OIDSET) From A To B
   = (forSrc X LinkDown A B) (multiLinkDown OIDSET From A To B) .


*** Some initial configurations for testing:
eq test1 = < a : Node | nbs : nbs(b,c),
                        states : states(state(a,passive),
                                        state(b,passive),
                                        state(c,passive)),
                        parents : mtparents,
                        seqNos : seqNos(seqNo(a,0),
                                        seqNo(b,0),
                                        seqNo(c,0)),
                        nbsStates : nbsStates(nbState(a,b,passive),
```

```
                                    nbState(a,c,passive)) >
           < b : Node | nbs : nbs(a,c),
                        states : states(state(a,passive),
                                        state(b,passive),
                                        state(c,passive)),
                        parents : mtparents,
                        seqNos : seqNos(seqNo(a,0),
                                        seqNo(b,0),
                                        seqNo(c,0)),
                        nbsStates : nbsStates(nbState(a,a,passive),
                                              nbState(a,c,passive)) >
           < c : Node | nbs : nbs(b,a),
                        states : states(state(a,passive),
                                        state(b,passive),
                                        state(c,passive)),
                        parents : mtparents,
                        seqNos : seqNos(seqNo(a,0),
                                        seqNo(b,0),
                                        seqNo(c,0)),
                        nbsStates : nbsStates(nbState(a,a,passive),
                                              nbState(a,b,passive)) > .

*** Test for New Link
eq test2 = < b : Node | nbs : a,
                        states : states(state(a,passive),
                                        state(b,passive)),
                        parents : mtparents,
                        seqNos : seqNos(seqNo(a,0),
                                        seqNo(b,0)),
                        nbsStates : nbState(a,a,passive) >
           < c : Node | nbs : a,
                        states : states(state(a,passive),
                                        state(c,passive)),
                        parents : mtparents,
                        seqNos : seqNos(seqNo(a,0),
                                        seqNo(c,0)),
                        nbsStates : nbState(a,a,passive) >
           newlink(b,c) .

*** Test for Failure of Link
eq test3 = < b : Node | nbs : nbs(a,c),
                        states : states(state(a,passive),
                                        state(b,passive),
                                        state(c,passive)),
                        parents : mtparents,
                        seqNos : seqNos(seqNo(a,0),
```

```
                                    seqNo(b,0),
                                    seqNo(c,0)),
                    nbsStates : nbsStates(nbState(b,a,passive),
                                          nbState(b,c,passive),
                                          nbState(a,a,passive),
                                          nbState(a,c,passive),
                                          nbState(c,a,passive),
                                          nbState(c,c,passive)) >
           < a : Node | nbs : b,
                    states : states(state(a,passive),
                                    state(b,passive)),
                    parents : mtparents,
                    seqNos : seqNos(seqNo(a,0),
                                    seqNo(b,0)),
                    nbsStates : nbsStates(nbState(a,b,passive),
                                          nbState(b,b,passive)) >

           < c : Node | nbs : b,
                    states : states(state(b,passive),
                                    state(c,passive)),
                    parents : mtparents,
                    seqNos : seqNos(seqNo(b,0),
                                    seqNo(c,0)),
                    nbsStates : nbsStates(nbState(c,b,passive),
                                          nbState(b,b,passive)) >
           failure(b,a) .


*** Start broadcasting a message:
*** RESTRICTED VERSION FOR MODELCHECKING:
***           only node 'a' sends exactly one message (a,1)
crl [Send] :
    < a : Node | nbs : OIDSET,
                 states : states(state(a,passive),IDSTATUSPFUN),
                 parents : IDIDPFUN,
                 seqNos : seqNos(seqNo(a,0),IDINTPFUN),
                 nbsStates : IDIDSTATUSPFUN >
   =>
     < a : Node | states : assignIdStatusPFun(IDSTATUSPFUN, a, active),
                  parents : assignIdIdPFun(IDIDPFUN, a, a),
                  seqNos : assignIdIntPFun(IDINTPFUN, a, 1),
                  nbsStates : assignAllIdIdStatusPFun(
                                   IDIDSTATUSPFUN, a, OIDSET, active) >
     (multimsg 1 To OIDSET From a Src a)
   if OIDSET =/= mtnbs .
```

```
*** Receiver (active or passive) receives current message -> ack
rl [RepeatRecCurrentMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  states : IDSTATUSPFUN,
                  parents : IDIDPFUN,
                  seqNos : seqNos(seqNo(C,M),IDINTPFUN),
                  nbsStates : IDIDSTATUSPFUN >
     (msg M To A From B Src C)
   =>
     < A : Node | nbs : nbs(B,OIDSET) >
     (ack M To B From A Src C) .


*** NONLEAFCASE
*** Receiver  (active or passive) receives newer message than his
*** current one (wrt to a source) -> forget old message, forward newest
*** message to all neighbors
crl [RecNewMsg] :
      < A : Node | nbs : nbs(B,OIDSET),
                   states : IDSTATUSPFUN,
                   parents : IDIDPFUN,
                   seqNos : IDINTPFUN,
                   nbsStates : IDIDSTATUSPFUN >
     (msg M To A From B Src C)
     =>
      < A : Node | states : assignIdStatusPFun(IDSTATUSPFUN, C, active),
                   parents : assignIdIdPFun(IDIDPFUN, C, B),
                   seqNos : assignIdIntPFun(IDINTPFUN, C, M),
                   nbsStates : assignAllIdIdStatusPFun(
                                 assignIdIdStatusPFun(
                                      IDIDSTATUSPFUN, C, B, passive),
                                 C, OIDSET, active) >
      (multimsg M To OIDSET From A Src C)
    if (OIDSET =/= mtnbs)
       and forSourceSmallerOrNoSeqnoInIdIntPFun(C,M,IDINTPFUN) .

*** LEAFCASE
*** Receiver  (active or passive) receives newer message than his
*** current one (wrt to a source) -> forget old message,
*** Receiver is leaf node, i.e., besides the object from which it got
*** the message, there are no more neighbors -> acknowledge sender
crl [RecNewMsgLeaf] :
     < A : Node | nbs : B,
                  states : IDSTATUSPFUN,
                  parents : IDIDPFUN,
```

```
                 seqNos : IDINTPFUN,
                 nbsStates : IDIDSTATUSPFUN >
     (msg M To A From B Src C)
    =>
     < A : Node | states : assignIdStatusPFun(IDSTATUSPFUN, C, passive),
                 parents : assignIdIdPFun(IDIDPFUN, C, B),
                 seqNos : assignIdIntPFun(IDINTPFUN, C, M),
                 nbsStates : assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                  C, B, passive) >
     (ack M To B From A Src C)
   if forSourceSmallerOrNoSeqnoInIdIntPFun(C,M,IDINTPFUN) .



*** Receiver  (active) receives older message than his
*** current one (wrt to a source) -> he sends the newest message to
*** the sender and sets the sender active
crl [ActiveRecOldMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                 states :  states(state(C,active),IDSTATUSPFUN),
                 seqNos : seqNos(seqNo(C,I),IDINTPFUN),
                 nbsStates : IDIDSTATUSPFUN >
     (msg M To A From B Src C)
   =>
    < A : Node | nbsStates : assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                  C, B, active) >
     (msg I To B From A Src C)
    if I > M .

*** Receiver  (passive) receives older message than his
*** current one (wrt to a source) -> he sends the newest message to
*** the sender and sets the sender active, and sets himself to be
*** active and to be the source of this LOCAL DIFFUSION COMPETITION
crl [PassiveRecOldMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                 states : states(state(C,passive),IDSTATUSPFUN),
                 parents : IDIDPFUN,
                 seqNos : seqNos(seqNo(C,I),IDINTPFUN),
                 nbsStates : IDIDSTATUSPFUN >
     (msg M To A From B Src C)
   =>
    < A : Node |  states : assignIdStatusPFun(IDSTATUSPFUN, C, active),
                 parents : assignIdIdPFun(IDIDPFUN, C, A),
                 nbsStates : assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                  C, B, active) >
     (msg I To B From A Src C)
```

```
    if I > M .

*** for newer message  -> ack current/old message
crl [RecAckNewerMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  seqNos : seqNos(seqNo(C,I),IDINTPFUN) >
     (ack M To A From B Src C)
   =>
     < A : Node | nbs : nbs(B,OIDSET) >
     (ack I To B From A Src C)
    if M > I  .


*** Receiver (passive) receives acknowledgment for current message
*** -> do nothing, consume message
rl [PassiveRecAckCurrentMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  states :  states(state(C,passive),IDSTATUSPFUN),
                  seqNos : seqNos(seqNo(C,M),IDINTPFUN) >
     (ack M To A From B Src C)
   =>
     < A : Node | nbs : nbs(B,OIDSET) > .


*** Receiver (active) receives acknowledgment for current message
*** -> set the sender to passive (wrt to the source), if all neighbors
*** already acknowledged this message, i.e., they are set to passive
*** wrt to the source, then the receiver sets itself to passive and
*** if the receiver is not the source, then also acknowledge the
*** source
rl [ActiveRecAckCurrentMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  states :  states(state(C,active),IDSTATUSPFUN),
                  parents : parents(parent(C,X),IDIDPFUN),
                  seqNos : seqNos(seqNo(C,M),IDINTPFUN),
                  nbsStates : nbsStates(nbState(C,B,STAT1),
                                        IDIDSTATUSPFUN) >
     (ack M To A From B Src C)
   => if allNbsPassiveWRTSrcUnlessParent(IDIDSTATUSPFUN, C, X) == true
      then if X =/= A
           then < A : Node | states : states(state(C,passive),
                                             IDSTATUSPFUN),
                             nbsStates : nbsStates(nbState(C,B,passive),
                                                   IDIDSTATUSPFUN) >
                (ack M To X From A Src C)
           else < A : Node | states : states(state(C,passive),
```

```
                                         IDSTATUSPFUN),
                          nbsStates : nbsStates(nbState(C,B,passive),
                                               IDIDSTATUSPFUN) >
           fi
      else < A : Node | nbsStates : nbsStates(nbState(C,B,passive),
                                         IDIDSTATUSPFUN) >
      fi .


*** Receiver (active) receives acknowledgment for older
*** message -> send newer/current message to the sender and sets the
*** sender to be active wrt to the source.
crl [ActiveRecAckOldMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  states :  states(state(C,active),IDSTATUSPFUN),
                  seqNos : seqNos(seqNo(C,I),IDINTPFUN),
                  nbsStates : IDIDSTATUSPFUN >
     (ack M To A From B Src C)
   =>
     < A : Node | nbsStates : assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                   C, B, active) >
     (msg I To B From A Src C)
   if I > M .


*** Receiver  (passive) receives acknowledgement for older message
*** than his current messages (wrt to a source) -> he sends the newest
*** message to the sender and sets the sender active, sets himself to be
*** active and to be the source of this LOCAL DIFFUSION COMPETITION
crl [PassiveRecAckOldMsg] :
     < A : Node | nbs : nbs(B,OIDSET),
                  states : states(state(C,passive),IDSTATUSPFUN),
                  parents : IDIDPFUN,
                  seqNos : seqNos(seqNo(C,I),IDINTPFUN),
                  nbsStates : IDIDSTATUSPFUN >
     (ack M To A From B Src C)
   =>
     < A : Node | states : assignIdStatusPFun(IDSTATUSPFUN, C, active),
                  parents : assignIdIdPFun(IDIDPFUN, C, A),
                  nbsStates : assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                   C, B, active) >
     (msg I To B From A Src C)
    if I > M .


*** NEW LINK
```

```
rl [Link] :
   newlink(A,B)
   => (linkUpFrom A To B) (linkUpFrom B To A) .

rl [MultiLinkUp] :
     (linkUpFrom A To B)
     < A : Node | seqNos : IDINTPFUN >
   =>
     (multiLinkUp dom(IDINTPFUN) From A To B)
     < A : Node | seqNos : IDINTPFUN > .


*** new link from B -> C for source A; B is active for A. Thus, send
*** the latest message to C and set C to be active
crl [ActiveLink] :
      (forSrc A LinkUp B C)
      < B : Node | nbs : OIDSET,
                   states : states(state(A,active),IDSTATUSPFUN),
                   seqNos : seqNos(seqNo(A,M),IDINTPFUN),
                   nbsStates : IDIDSTATUSPFUN >
   =>
      < B : Node | nbs : nbs(C,OIDSET),
                   states : assignIdStatusPFunNoOverriding(
                              states(state(A,active),IDSTATUSPFUN),
                              C, passive),
                   seqNos : assignIdIntPFunNoOverriding(
                              seqNos(seqNo(A,M),IDINTPFUN),
                              C, 0),
                   nbsStates : assignAllIdIdStatusPFunNoOverriding(
                                 assignIdIdStatusPFun(IDIDSTATUSPFUN,
                                                      A, C, active),
                                 C, nbs(C,OIDSET), passive) >
       (msg M To C From B Src A)
   if M > 0 .


*** PassiveLink is similar to ActiveLink, but
*** LOCAL DIFFUSION COMPETITION (PassiveLink)
*** vs FORWARD (ActiveLink)
*** Therefore, additionally B sets
*** itself to active and to the parent of this LOCAL DIFFUSION
*** COMPETITION (if there is a message to be sent)
rl [PassiveLink] :
    (forSrc A LinkUp B C)
    < B : Node | nbs : OIDSET,
                 states : states(state(A,passive),IDSTATUSPFUN),
```

```
                 parents : IDIDPFUN,
                 seqNos : seqNos(seqNo(A,M),IDINTPFUN),
                 nbsStates : IDIDSTATUSPFUN >
   =>
   if M > 0
   then < B : Node | nbs : nbs(C,OIDSET),
                     states : assignIdStatusPFunNoOverriding(
                                  states(state(A,active),IDSTATUSPFUN),
                                  C, passive),
                     parents : assignIdIdPFun(IDIDPFUN, A, B),
                     seqNos : assignIdIntPFunNoOverriding(
                                  seqNos(seqNo(A,M),IDINTPFUN),
                                  C, 0),
                     nbsStates : assignAllIdIdStatusPFunNoOverriding(
                                     assignIdIdStatusPFun(
                                          IDIDSTATUSPFUN, A, C, active),
                                     C, nbs(C,OIDSET), passive) >
          (msg M To C From B Src A)
   else  < B : Node | nbs : nbs(C,OIDSET),
                      states : assignIdStatusPFunNoOverriding(
                                  states(state(A,passive),IDSTATUSPFUN),
                                  C, passive),
                      seqNos : assignIdIntPFunNoOverriding(
                                  seqNos(seqNo(A,M),IDINTPFUN), C, 0),
                      nbsStates : assignAllIdIdStatusPFunNoOverriding(
                                     assignIdIdStatusPFun(
                                        IDIDSTATUSPFUN, A, C, passive),
                                     C, nbs(C,OIDSET), passive) >
   fi .


*** FAILURE OF A LINK
rl [Failure] :
   failure(A,B)
   =>
   (linkDownFrom A To B) (linkDownFrom B To A) .

rl [MultiLinkDown] :
    (linkDownFrom A To B)
    < A : Node | seqNos : IDINTPFUN >
   =>
    (multiLinkDown dom(IDINTPFUN) From A To B)
    < A : Node | seqNos : IDINTPFUN  > .


*** link down from B -> C for source A; B is active for A
```

```
*** delete neighbor C from neighbor set and from neighborstate set
*** moreover, if all other neighbors acknowledged the latest message, then
*** if B is not the parent for source A, B will acknowledge the
*** message and set himself to passive wrt to A
*** otherwise B only sets himself to passive wrt  to A (no ack needed)
crl [ActiveFailure] :
    (forSrc A LinkDown B C)
    < B : Node | nbs : OIDSET,
                 states :  states(state(A,active),IDSTATUSPFUN),
                 parents : parents(parent(A,X), IDIDPFUN),
                 seqNos : seqNos(seqNo(A,M),IDINTPFUN),
                 nbsStates : IDIDSTATUSPFUN >
   => if  allNbsPassiveWRTSrcUnlessParent(
              deleteIdIdStatusPFunAsNeighborForSource(IDIDSTATUSPFUN, C, A),
              A, X)
           == true
      then (if X =/= B
           then  < B : Node | nbs : deleteFromSet(C,OIDSET),
                              states : states(state(A,passive),
                                               IDSTATUSPFUN),
                              parents : deleteIdIdPFunAsParentForSource(
                                             IDIDPFUN, C, A),
                              nbsStates :
                                 deleteIdIdStatusPFunAsNeighborForSource(
                                                     IDIDSTATUSPFUN, C, A) >
                 (ack M To X From B Src A)
            else < B : Node | nbs : deleteFromSet(C,OIDSET),
                              states : states(state(A,passive),IDSTATUSPFUN),
                              parents : deleteIdIdPFunAsParentForSource(
                                               IDIDPFUN, C, A),
                              nbsStates :
                                 deleteIdIdStatusPFunAsNeighborForSource(
                                                     IDIDSTATUSPFUN, C, A) >
            fi)
      else  < B : Node | nbs : deleteFromSet(C,OIDSET),
                         parents : deleteIdIdPFunAsParentForSource(
                                                     IDIDPFUN, C, A),
                         nbsStates :
                             deleteIdIdStatusPFunAsNeighborForSource(
                                                 IDIDSTATUSPFUN, C, A) >
      fi
   if M > 0 .


*** FAILURE OF Link
*** node is passive for the corresponding source
```

```
  rl [PassiveFailure] :
      (forSrc A LinkDown B C)
      < B : Node | nbs : OIDSET,
                   states :  states(state(A,passive),IDSTATUSPFUN),
                   parents : IDIDPFUN,
                   nbsStates : IDIDSTATUSPFUN >
     =>
       < B : Node | nbs : deleteFromSet(C,OIDSET),
                    parents : deleteIdIdPFunAsParentForSource(IDIDPFUN, C, A),
                    nbsStates :
                         deleteAllIdIdStatusPFunAsNeighborForSource(
                                                IDIDSTATUSPFUN, C, A) > .


  *** node has no entry for the corresponding source, i.e., no
  *** information whether it is active or passive for this source
  crl [FailureNoSrc] :
      (forSrc A LinkDown B C)
      < B : Node | nbs : OIDSET,
                   parents : IDIDPFUN,
                   nbsStates : IDIDSTATUSPFUN >
     =>
       < B : Node | nbs : deleteFromSet(C,OIDSET),
                    parents : deleteIdIdPFunAsParentForSource(IDIDPFUN, C, A),
                    nbsStates : deleteAllIdIdStatusPFunAsNeighborForSource(IDIDSTATUSPFUN,
C, A) >
     if not(inIdStatusPFun(A,IDSTATUSPFUN)) .

  endom)
```

Assuming that this file is in `dynamicrbp.maude`, a Full Maude session may go as follows:

```
Maude> in dynamicrbp.maude


Introduced module: RBP
Maude> (rew test1 .)

Rewrite in RBP : test1 .
Result Configuration :
< a : Node | nbs : nbs ( b , c ) ,
             nbsStates : nbsStates ( nbState ( a , b , passive ) ,
                                     nbState ( a , c , passive ) ) ,
             seqNos : seqNos ( seqNo ( a , 1 ) ,
                               seqNo ( b , 0 ) ,
```

```
                        seqNo ( c , 0 ) ) ,
          parents : parent ( a , a ) ,
          states : states ( state ( a , passive ) ,
                            state ( b , passive ) ,
                            state ( c , passive ) ) >
< b : Node | nbs : nbs ( a , c ) ,
             nbsStates : nbsStates ( nbState ( a , a , passive ) ,
                                     nbState ( a , c , passive ) ) ,
             seqNos : seqNos ( seqNo ( a , 1 ) ,
                               seqNo ( b , 0 ) ,
                               seqNo ( c , 0 ) ) ,
             parents : parent ( a , a ) ,
             states : states ( state ( a , passive ) ,
                               state ( b , passive ) ,
                               state ( c , passive ) ) >
< c : Node | nbs : nbs ( a , b ) ,
             nbsStates : nbsStates ( nbState ( a , a , passive ) ,
                                     nbState ( a , b , passive ) ) ,
             seqNos : seqNos ( seqNo ( a , 1 ) ,
                               seqNo ( b , 0 ) ,
                               seqNo ( c , 0 ) ) ,
             parents : parent ( a , a ) ,
             states : states ( state ( a , passive ) ,
                               state ( b , passive ) ,
                               state ( c , passive ) ) >
Maude> (rew test2 .)

Rewrite in RBP : test2 .
Result Configuration :
< b : Node | nbs : nbs ( a , c ) ,
             nbsStates : nbsStates ( nbState ( a , a , passive ) ,
                                     nbState ( a , c , passive ) ,
                                     nbState ( b , c , passive ) ,
                                     nbState ( c , a , passive ) ,
                                     nbState ( c , c , passive ) ) ,
             seqNos : seqNos ( seqNo ( a , 0 ) ,
                               seqNo ( b , 0 ) ,
                               seqNo ( c , 0 ) ) ,
             parents : mtparents ,
             states : states ( state ( a , passive ) ,
                               state ( b , passive ) ,
                               state ( c , passive ) ) >
< c : Node | nbs : nbs ( a , b ) ,
             nbsStates : nbsStates ( nbState ( a , a , passive ) ,
                                     nbState ( a , b , passive ) ,
                                     nbState ( b , a , passive ) ,
```

```
                                 nbState ( b , b , passive ) ,
                                 nbState ( c , b , passive ) ) ,
             seqNos : seqNos ( seqNo ( a , 0 ) ,
                               seqNo ( b , 0 ) ,
                               seqNo ( c , 0 ) ) ,
             parents : mtparents ,
             states : states ( state ( a , passive ) ,
                               state ( b , passive ) ,
                               state ( c , passive ) ) >
Maude> (rew test3 .)

Rewrite in RBP : test3 .
Result Configuration :
< a : Node | nbs : mtnbs ,
             nbsStates : mtnbsStates ,
             seqNos : seqNos ( seqNo ( a , 0 ) ,
                               seqNo ( b , 0 ) ) ,
             parents : mtparents ,
             states : states ( state ( a , passive ) ,
                               state ( b , passive ) ) >
< b : Node | nbs : c ,
             nbsStates : nbsStates ( nbState ( a , c , passive ) ,
                                     nbState ( b , c , passive ) ,
                                     nbState ( c , c , passive ) ) ,
             seqNos : seqNos ( seqNo ( a , 0 ) ,
                               seqNo ( b , 0 ) ,
                               seqNo ( c , 0 ) ) ,
             parents : mtparents ,
             states : states ( state ( a , passive ) ,
                               state ( b , passive ) ,
                               state ( c , passive ) ) >
< c : Node | nbs : b ,
             nbsStates : nbsStates ( nbState ( b , b , passive ) ,
                                     nbState ( c , b , passive ) ) ,
             seqNos : seqNos ( seqNo ( b , 0 ) ,
                               seqNo ( c , 0 ) ) ,
             parents : mtparents ,
             states : states ( state ( b , passive ) ,
                               state ( c , passive ) ) >
Maude>
```

and can be continued by exploring all possible rewrites:

```
Maude> in meta-rbp.maude

Introduced module: META-RBP
```

```
Maude> (red irreducibleforms(RBP, {'test2}'Configuration, alldynamicrules) .)

Reduce in META-RBP : irreducibleforms ( RBP , { 'test2 } 'Configuration , alldynamicrules )
Result Term : '__ [
'<_:_|_>
 [ { 'b } 'Oid ,
   { 'Node } 'Node ,
   '_`,_ [ 'nbs`:_ [ 'nbs [ { 'a } 'Oid , { 'c } 'Oid ] ] ,
           'nbsStates`:_ [ 'nbsStates [
               'nbState [ { 'a } 'Oid , { 'a } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'a } 'Oid , { 'c } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'b } 'Oid , { 'c } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'c } 'Oid , { 'a } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'c } 'Oid , { 'c } 'Oid , { 'passive } 'Status ] ] ] ,
            'seqNos`:_ [ 'seqNos [
               'seqNo [ { 'a } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
            'parents`:_ [ { 'mtparents } 'IdIdPFun ] ,
            'states`:_ [ 'states [
               'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ] ] ,

'<_:_|_>
 [ { 'c } 'Oid ,
   { 'Node } 'Node ,
   '_`,_ [ 'nbs`:_ [ 'nbs [ { 'a } 'Oid , { 'b } 'Oid ] ] ,
           'nbsStates`:_ [ 'nbsStates [
               'nbState [ { 'a } 'Oid , { 'a } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'a } 'Oid , { 'b } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'b } 'Oid , { 'a } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'b } 'Oid , { 'b } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'c } 'Oid , { 'b } 'Oid , { 'passive } 'Status ] ] ] ,
            'seqNos`:_ [ 'seqNos [
               'seqNo [ { 'a } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
            'parents`:_ [ { 'mtparents } 'IdIdPFun ] ,
            'states`:_ [ 'states [
               'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ] ] ]

Maude>  (red irreducibleforms(RBP, {'test1}'Configuration, alldynamicrules) .)
```

```
Reduce in META-RBP : irreducibleforms ( RBP , { 'test1 } 'Configuration , alldynamicrules )
Result TermSet : set ( '__  [
'<_:_|_>
 [ { 'a } 'Oid ,
   { 'Node } 'Node ,
   '_`,_ [ 'nbs`:_ [ 'nbs [ { 'b } 'Oid , { 'c } 'Oid ] ] ,
           'nbsStates`:_ [ 'nbsStates [
               'nbState [ { 'a } 'Oid , { 'b } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'a } 'Oid , { 'c } 'Oid , { 'passive } 'Status ] ] ] ,
           'seqNos`:_ [ 'seqNos [
               'seqNo [ { 'a } 'Oid , { '1 } 'NzMachineInt ] ,
               'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
           'parents`:_ [ 'parent [ { 'a } 'Oid , { 'a } 'Oid ] ] ,
           'states`:_ [ 'states [
               'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ] ] ,
'<_:_|_>
 [ { 'b } 'Oid , { 'Node } 'Node ,
   '_`,_ [ 'nbs`:_ [ 'nbs [ { 'a } 'Oid , { 'c } 'Oid ] ] ,
           'nbsStates`:_ [ 'nbsStates [
               'nbState [ { 'a } 'Oid , { 'a } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'a } 'Oid , { 'c } 'Oid , { 'passive } 'Status ] ] ] ,
           'seqNos`:_ [ 'seqNos [
               'seqNo [ { 'a } 'Oid , { '1 } 'NzMachineInt ] ,
               'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
           'parents`:_ [ 'parent [ { 'a } 'Oid , { 'a } 'Oid ] ] ,
           'states`:_ [ 'states [
               'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ] ] ,
'<_:_|_>
 [ { 'c } 'Oid ,
   { 'Node } 'Node ,
   '_`,_ [ 'nbs`:_ [ 'nbs [ { 'a } 'Oid , { 'b } 'Oid ] ] ,
           'nbsStates`:_ [ 'nbsStates [
               'nbState [ { 'a } 'Oid , { 'a } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'a } 'Oid , { 'b } 'Oid , { 'passive } 'Status ] ] ] ,
           'seqNos`:_ [ 'seqNos [
               'seqNo [ { 'a } 'Oid , { '1 } 'NzMachineInt ] ,
               'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
           'parents`:_ [ 'parent [ { 'a } 'Oid , { 'a } 'Oid ] ] ,
           'states`:_ [ 'states [
```

```
               'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ]
               ] ] ,


'__ [
'<_:_|_>
 [ { 'a } 'Oid ,
   { 'Node } 'Node ,
   '_`,_ [ 'nbs`:_ [ 'nbs [ { 'b } 'Oid , { 'c } 'Oid ] ] ,
           'nbsStates`:_ [ 'nbsStates [
               'nbState [ { 'a } 'Oid , { 'b } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'a } 'Oid , { 'c } 'Oid , { 'passive } 'Status ] ] ] ,
           'seqNos`:_ [ 'seqNos [
               'seqNo [ { 'a } 'Oid , { '1 } 'NzMachineInt ] ,
               'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
           'parents`:_ [ 'parent [ { 'a } 'Oid , { 'a } 'Oid ] ] ,
           'states`:_ [ 'states [
               'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ] ] ,
'<_:_|_>
 [ { 'b } 'Oid ,
   { 'Node } 'Node ,
   '_`,_ [ 'nbs`:_ [ 'nbs [ { 'a } 'Oid , { 'c } 'Oid ] ] ,
           'nbsStates`:_ [ 'nbsStates [
               'nbState [ { 'a } 'Oid , { 'a } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'a } 'Oid , { 'c } 'Oid , { 'passive } 'Status ] ] ] ,
           'seqNos`:_ [ 'seqNos [
               'seqNo [ { 'a } 'Oid , { '1 } 'NzMachineInt ] ,
               'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
           'parents`:_ [ 'parent [ { 'a } 'Oid , { 'a } 'Oid ] ] ,
           'states`:_ [ 'states [
               'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ] ] ,
'<_:_|_>
  [ { 'c } 'Oid ,
    { 'Node } 'Node ,
    '_`,_ [ 'nbs`:_ [ 'nbs [ { 'a } 'Oid , { 'b } 'Oid ] ] ,
            'nbsStates`:_ [ 'nbsStates [
               'nbState [ { 'a } 'Oid , { 'a } 'Oid , { 'passive } 'Status
```

```
] ,
               'nbState [ { 'a } 'Oid , { 'b } 'Oid , { 'passive } 'Status ] ] ] ,
            'seqNos`:_ [ 'seqNos [
               'seqNo [ { 'a } 'Oid , { '1 } 'NzMachineInt ] ,
               'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
            'parents`:_ [ 'parent [ { 'a } 'Oid , { 'b } 'Oid ] ] ,
            'states`:_ [ 'states [
               'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ] ] ] ,
'__
[ '<_:_|_>
   [ { 'a } 'Oid ,
   { 'Node } 'Node ,
   '_`,_ [ 'nbs`:_ [ 'nbs [ { 'b } 'Oid , { 'c } 'Oid ] ] ,
            'nbsStates`:_ [ 'nbsStates [
               'nbState [ { 'a } 'Oid , { 'b } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'a } 'Oid , { 'c } 'Oid , { 'passive } 'Status ] ] ] ,
            'seqNos`:_ [ 'seqNos [
               'seqNo [ { 'a } 'Oid , { '1 } 'NzMachineInt ] ,
               'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
            'parents`:_ [ 'parent [ { 'a } 'Oid , { 'a } 'Oid ] ] ,
            'states`:_ [ 'states [
               'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ] ] ,

'<_:_|_>
 [ { 'b } 'Oid ,
   { 'Node } 'Node ,
   '_`,_ [ 'nbs`:_ [ 'nbs [ { 'a } 'Oid , { 'c } 'Oid ] ] ,
            'nbsStates`:_ [ 'nbsStates [
               'nbState [ { 'a } 'Oid , { 'a } 'Oid , { 'passive } 'Status ] ,
               'nbState [ { 'a } 'Oid , { 'c } 'Oid , { 'passive } 'Status ] ] ] ,
            'seqNos`:_ [ 'seqNos [
               'seqNo [ { 'a } 'Oid , { '1 } 'NzMachineInt ] ,
               'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
               'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
             'parents`:_ [ 'parent [ { 'a } 'Oid , { 'c } 'Oid ] ] ,
             'states`:_ [ 'states [
               'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
               'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ] ] ,
```

```
'<_:_|_>
 [ { 'c } 'Oid ,
   { 'Node } 'Node ,
   '_`,_ [ 'nbs`:_ [ 'nbs [ { 'a } 'Oid , { 'b } 'Oid ] ] ,
          'nbsStates`:_ [ 'nbsStates [
              'nbState [ { 'a } 'Oid , { 'a } 'Oid , { 'passive } 'Status ] ,
              'nbState [ { 'a } 'Oid , { 'b } 'Oid , { 'passive } 'Status ] ] ] ,
          'seqNos`:_ [ 'seqNos [
              'seqNo [ { 'a } 'Oid , { '1 } 'NzMachineInt ] ,
              'seqNo [ { 'b } 'Oid , { '0 } 'MachineInt ] ,
              'seqNo [ { 'c } 'Oid , { '0 } 'MachineInt ] ] ] ,
          'parents`:_ [ 'parent [ { 'a } 'Oid , { 'a } 'Oid ] ] ,
          'states`:_ [ 'states [
              'state [ { 'a } 'Oid , { 'passive } 'Status ] ,
              'state [ { 'b } 'Oid , { 'passive } 'Status ] ,
              'state [ { 'c } 'Oid , { 'passive } 'Status ] ] ] ] ] ] )
Maude>
```

Since `test2` only has one irreduble form, it is easy to pretty print it:

```
Maude> (down RBP : red irreducibleforms(RBP, {'test2}'Configuration, alldynamicrules) .)

Result Configuration :

< b : Node | nbs : nbs ( a , c ) ,
             nbsStates : nbsStates ( nbState ( a , a , passive ) ,
                                     nbState ( a , c , passive ) ,
                                     nbState ( b , c , passive ) ,
                                     nbState ( c , a , passive ) ,
                                     nbState ( c , c , passive ) ) ,
             seqNos : seqNos ( seqNo ( a , 0 ) ,
                               seqNo ( b , 0 ) ,
                               seqNo ( c , 0 ) ) ,
              parents : mtparents ,
             states : states ( state ( a , passive ) ,
                               state ( b , passive ) ,
                               state ( c , passive ) ) >

< c : Node | nbs : nbs ( a , b ) ,
             nbsStates : nbsStates ( nbState ( a , a , passive ) ,
                                     nbState ( a , b , passive ) ,
                                     nbState ( b , a , passive ) ,
                                     nbState ( b , b , passive ) ,
                                     nbState ( c , b , passive ) ) ,
             seqNos : seqNos ( seqNo ( a , 0 ) ,
                               seqNo ( b , 0 ) ,
```

```
                        seqNo ( c , 0 ) ) ,
          parents : mtparents ,
          states : states ( state ( a , passive ) ,
                            state ( b , passive ) ,
                            state ( c , passive ) ) >
Maude>
```