Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 176 (2007) 199–213
www.elsevier.com/locate/entcs

On Modelling Sensor Networks in Maude

Dilia E. Rodríguez[1]

Information Directorate
Air Force Research Laboratory
Rome, NY, USA 13441

Abstract

Two problems in modelling sensor networks are: how to scale up specification and analysis techniques to larger models, and how to flexibly monitor their behavior. Guided by three obvious principles, and exploiting features of Maude, a high-level, modular approach is used to construct a framework for the specification of sensor networks that structures computations and states so as to flexibly monitor a model of the network, reduce the size of the state, and very significantly reduce the execution times of various analysis methods.

Keywords: Sensor network, Maude

1 Introduction

Formal methods can be useful in ensuring the correctness of complex systems. Yet a continuing research problem is how to scale up the applicability of these methods to larger models. Further spurring this quest is the need for research on sensor networks, which are very large-scale. Formal methods that would support this research must integrate specification and analysis of correctness with flexible monitoring of relevant metrics, as well as ameliorate the scalability problem. Maude [1][2]is an executable, formal specification language that implements rewriting logic[6], a powerful logic of concurrent change, and supports a range of formal methods. The approach taken here to accommodate the demands of modelling a sensor network is to significantly and jointly improve the efficiency and flexibility of the formal specification of the network. It exploits features of Maude, and guided by some obvious principles, structures states and computations so as to flexibly monitor a model of a system, effectively reduce the size of the state, and very significantly reduce the execution times of various analysis methods.

A sensor network consists of hundreds or thousands of battery-operated devices that communicate wirelessly, and are capable of limited computation and sensing.

[1] Approved for Public Release; distribution unlimited.

1571-0661 Published by Elsevier B.V.
doi:10.1016/j.entcs.2007.06.016

Being very different from a wired network, and a small wireless network, it makes possible new commercial, military and scientific applications, but much research is needed for the notional sensor network to be realized. In the network community most research depends on simulations, in which some of the possible behaviors of the system are explored. Maude is an executable specification language that supports a range of increasingly stronger formal methods: specification; simulation, by execution of the specification; searching of the state space, for reachability analysis; and model checking. The search and model-checking methods may examine all behaviors of models that are not too large, or otherwise a smaller subset limited by some constraints on the behaviors. These methods can be used individually and in combination to illuminate the behaviors of complex systems. This paper is concerned with scaling up the applicability of these methods. Both methods involve searching the state space of the model. The usual approach to address the scalability problem is to attack the state-space explosion problem. By contrast, the treatment presented here, while constraining the size of the state space, concentrates on significantly improving the efficiency of the model, which carries over to the efficiency of these formal methods.

The efficient model is constructed adhering to the essential distinction Maude makes in expressing determinism and nondeterminism, and exploiting some features of Maude guided by some obvious principles. A Maude specification corresponds to a rewrite theory in rewriting logic, which expresses the concurrent transitions of the system by a set of rewrite rules R, and the deterministic computations by a set of equations E. The rewrite rules R are rewritten *modulo* the equations E, which means that only the rules contribute to the size of the state space. The first obvious principle then is *If a feature is deterministic, do not treat it as nondeterministic.* For each kind of rewrite — with rules, and with equations — Maude offers a mechanism to control which part of the state is subject to rewrites. These mechanisms make it possible to abide by this other principle *If a part of a state cannot be rewritten, do not try.* These two principles guide the most fundamental decisions in the design of the model. It is a high-level approach that leads to efficiency.

The model of a sensor network can be described by the following epitome:

$$\begin{aligned} &\mathit{sensor\ network\ model} \\ &= \mathit{message\ transmission}(\mathit{nondeterministic}) \\ &\quad + \mathit{performance\ monitoring}(\mathit{deterministic}) \end{aligned}$$

Due to the strict limitations of the nodes constituting a sensor network, its model must monitor relevant performance metrics of the network, since they will determine the viability of the design or protocol of the network. A good model of the network provides flexible monitoring, as well as efficient execution. The principles introduced above guide the construction of such a model.

$$\begin{aligned} &\mathit{optimized\ sensor\ network\ model} \\ &= \mathsf{n}\rangle\!\rangle \mathit{all\ concurrent\ message\ transmissions} \langle\!\langle \mathsf{n} \\ &\quad \triangleright\ \mathsf{d}\rangle\!\rangle \mathit{performance\ monitoring} \langle\!\langle \mathsf{d} \end{aligned}$$

Its computations allow the observation of the state after each concurrent step, which

is a maximal subcomputation of concurrent message transmissions. This pattern not only affords flexibility in monitoring, but provides an opportunity for improved efficiency. The size of the state is effectively reduced at each concurrent step to fit the concurrent transitions, and after each concurrent step to optimize the deterministic observation of the state.

Two reasons justify the use of time in constructing this optimized model of a sensor network. First, time is a significant parameter in a sensor network. Since nodes have limited power, network lifetime is an important criterion in evaluating different network designs, as are various latency measures. Second, time serves to identify concurrent transitions. Each transition occurs at some given time. So transitions are concurrent if they occur at the same time. The optimized model of a sensor network described above is therefore most appropriately specified in an extension of Maude that supports the specification and analysis of real-time systems, Real-Time Maude [11].

This model, which is quite general, can be optimized further for some applications by observing a third principle: *If a smaller size of the state suffices, have it not larger.* This guidance can be applied in two circumstances. First, the wireless nature of communication means that many protocols induce propagation of messages. At different times the transmissions comprising a propagation are localized in different areas. It may be appropriate to represent a node outside the localized area of activity with a smaller state than one that is active. This is an opportunity to temporarily actually reduce the size of the network state. Second, in some protocols the participation of a node at some point comes to an end. It may be possible from then on to represent the network without that node. An example will illustrate these cases. Experiments show the effects of these further optimizations, with the fullest optimizations improving execution times up to two orders of magnitude.

2 Maude

Maude [1][2] is an executable language based on rewriting logic [6], a logic of concurrent change. In rewriting logic, a concurrent system is specified by a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, where (Σ, E) is a membership equational theory, with the signature Σ specifying kinds, sorts, and operations; E is a set of equations on Σ-terms; R is a set of labelled conditional rewrite rules,

$$l : t \longrightarrow t' \text{ if } cond$$

and ϕ is a function that specifies for each operation the set of argument positions on which rewriting with rules is precluded.

A rewrite theory corresponds to a system module in Maude. For system modules that satisfy some admissibility requirements [2], rewriting with rules is performed modulo the equations of the module. Thus, only rules contribute to the size of the state space.

This Core Maude is extended, in the language Full Maude, to support a module algebra and a declarative form of object orientation. An object-oriented system is represented by a term of sort `Configuration`, which has subsorts `Object` and `Msg`.

A configuration is a multiset of objects and messages, which is constructed with the juxtaposition operator. It is represented by a term of the following form:

$$M_1 \dots M_m \;\; \langle O_1 : C_1 | atts_1 \rangle \dots \langle O_n : C_n | atts_n \rangle.$$

The Ms are messages, and the other terms give the states of objects named O_i of class C_i, with the values of their attributes given in $atts_i$, for $i = 1, \dots, n$.

The transitions a configuration may take are specified by rules of the following general form:

$$\begin{aligned} r: \quad & M_1 \dots M_m \;\; \langle O_1 : C_1 | atts_1 \rangle \dots \langle O_n : C_n | atts_n \rangle \\ \rightarrow \quad & \langle O_{i_1} : C'_{i_1} | atts'_{i_1} \rangle \dots \langle O_{i_n} : C'_{i_n} | atts'_{i_n} \rangle \\ & \langle Q_1 : D_1 | atts''_1 \rangle \dots \langle Q_p : D_p | atts''_p \rangle \;\; M'_1 \dots M'_q \\ & \text{if } cond \qquad . \end{aligned}$$

If the condition *cond* holds, the messages M are consumed; some of the original objects O persist with new states and possibly new classes C', and new objects Q and messages M' emerge. Full Maude supports the convention that a rule or equation need not mention attributes of an object that are irrelevant.

Another extension of Maude supports the specification of real-time and hybrid systems. Real-Time Maude extends Full Maude with sorts and rules to model time. A real-time specification includes a specification of a sort `Time` for the time domain, and models the elapse of time with *tick rules* [11], which have the form:

$$\text{crl } \{t\} \longrightarrow \{t'\} \text{ in time } u \text{ if } cond \; .$$

This rule represents global transitions in which, if the condition *cond* is satisfied, the advance of time by u, of sort `Time`, transforms state t — the state of the entire system, which is signified by the curly brackets — into state t'.

In [11][9], Ölveczky and Meseguer introduce a scheme to model time for complex systems. It describes tick rules in terms of two functions:

$$\text{crl } \{t\} \longrightarrow \{\texttt{delta}(t, u)\} \text{ in time } u \text{ if } u \leq \texttt{mte}(t) \text{ and } cond' \; .$$

Function `mte` defines the maximum time elapse that will ensure time constraints; and function `delta` defines the effect of the elapse of time on the state of the system.

```
op mte : Configuration -> TimeInf [frozen (1)] .
op delta : Configuration Time -> Configuration [frozen (1)] .
```

As long as the value of `mte` is 0, time may not advance. On the other hand, when there is no constraint on the elapse of time the value of `mte` is infinity, `INF`, which is included in the sort `TimeInf` that extends `Time` with `INF`.

Each language in this family — Core Maude, Full Maude, and Real-Time Maude — supports a spectrum of techniques for the analysis of specifications: symbolic simulation, search of the state space, and model checking. In Real-Time Maude, these techniques exploit time to limit the size of the reachable state space.

3 Example

The sample protocol introduced here is used to illustrate a few ideas on modelling sensor networks, and to subject it to experimentation with the proposed techniques. It is intended to be simple, yet appropriate and relevant, *for the purposes of this work*. This choice is flooding. It is used to show how to model wireless communication, and support the collection of metrics. The proposed techniques do not depend on the particulars of how this is done. What is exploited is that a single transmission affects a small subset of nodes, and what is accommodated is the opportunity to observe the state at well defined and convenient points in the computation. Thus, as presented here, the flooding example addresses two considerations that are common to any sensor network protocol.

The techniques being proposed to improve the efficiency of the network model are quite opportunistic. Flooding, though a most simple protocol, is among the protocols least amenable to these techniques. A sensor network is a large collection of nodes covering a sensing field, with a few observers to receive data triggered by some events, or initiate queries. Consider a sensor network of n nodes, and suppose an observer is interested in some event in the northeast quadrant — comprised, say of $n/4$ nodes. A protocol that would deliver the required data would propagate messages mainly through that quadrant. Much of the rest of the network would be entirely uninvolved in this process. The proposed techniques take this opportunity to treat the network more like one of $n/4$ nodes than of n nodes. By contrast, flooding seeks to propagate a message to all n nodes. If the opportunities found in flooding make a difference, the techniques should be useful for other protocols.

Flooding underlies many sophisticated protocols for sensor networks [4]. For example, it is used for exploration in directed diffusion [5], for issuing "sleep" and "wake up" commands, and for multihop time synchronization [3]. Thus, for this work, the example of flooding seems appropriate.

A wireless network constrains the communication to be local: a message transmitted by a node `T` reaches all nodes within its communication range — its neighbors — and no other. This is specified as follows:

```
class WirelessNode .
crl [transmit] : T neNs  =>   T ~~> neNs
  if T ready-to-transmit and neNs the-neighbors-of T .
crl [transmit-to-none] : T  =>  T ~~> none
  if T ready-to-transmit and none the-neighbors-of T .
```

This specification reflects one basic decision: there are only objects in the configuration. This is to restrain the size of the configuration. The messages exchanged in the application are represented within the model of the transmitter, and the model of a neighbor. At this point, no restriction has been imposed on what constitutes a message; here it is entirely hidden and general. Different subclasses of `WirelessNode` can define a message in different ways. Different corresponding definitions of the operation `~~>` would define the transmission of a message by specifying the changes it effects in the state of the transmitter and the states of its neighbors. This is a general specification that captures the wireless nature of the communication between nodes.

In pure flooding each node retransmits any message it receives. A class `FloodingNode` specifies nodes of a sensor network that are engaged in pure flooding.

```
    class FloodingNode |
        neighbors : NzNatSet,     clock : Time,
        messages : MessageQueue,  heard : Nat .
***     others to support collection of metrics
    subclass FloodingNode < WirelessNode .
```

This example assumes no mobility, a usual case for sensor networks. Objects modelling the nodes have indexed identifiers, so the attribute `neighbors` is a set of indices. A wireless node receives a message when it hears a single message. It hears a message when any neighbor transmits. If a node hears more than one message at the same time, the messages interfere or collide, and the node receives no message. The attribute `heard` tracks the number of messages the node hears at the current time, which is the value of the attribute `clock`. A single queue, the attribute `messages`, holds messages received, and messages to be transmitted. As a message is received, and enqueued, it is assigned a processing time. When this has elapsed, the node transmits the message. To determine whether a node has received a message, all concurrent transmissions must be taken into account. Section 4 describes how to ensure this, while Section 5 introduces the remaining attributes of `FloodingNode`, which are needed to support the collection of metrics.

Pure flooding in a dense, wireless network results in a high rate of collisions, and too much redundancy [7]. Random delays in retransmissions allay the first problem. A limit on the number of retransmissions alleviates the second. A count-based scheme that incorporates these measures can be specified for `CB-FloodingNode`s.

```
    class CB-FloodingNode | received : Nat .
    subclass CB-FloodingNode < FloodingNode .
```

A node that receives, retransmits. When a `#received-limit` is reached, the node no longer will receive. So its participation in the protocol ends. Section 7 describes how to reduce the size of the state by removing these nodes.

4 Cycle of Rewrites

A model of a sensor network must not only specify the activity of the network, but also observe it. From this initial decision, two principles guide the construction of a specification that is efficient in execution and flexible in monitoring the performance of the network. The first turns an essential distinction in Maude into guidance: *If a feature is deterministic, do not treat it as nondeterministic.* Nondeterministic computation is described with rewrite rules, while deterministic computation is described with equations. Only rules contribute to the size of the state space. This principle ensures that the state space is no larger than it need be. The second principle optimizes the efficiency of execution: *If a part of a state cannot be rewritten, do not try.* This can be accomplished by exploiting mechanisms that Maude provides to control rewriting. These principles structure computation and state. They determine a cycle of rewrites that is the framework of the specification and orchestrates changes in the representation of the state to achieve the dual goal of efficient execution and flexible monitoring.

Activity changes state; observation, does not. Though observations can be made between any state changes, some structure provides reasonable flexibility in monitoring, and can be exploited to improve efficiency. The activity of the network is accomplished by the nodes through the exchange of messages. Instead of allowing observations after every message transmission, the opportunity to observe the state comes after all concurrent message transmissions have occurred. Observation is allowed only after such a "concurrent step".

This structure induces a cycle of rewrites. Various nodes may transmit messages concurrently; which, in general, makes the behavior of the network nondeterministic. While equations may add convenience, conciseness or efficiency to the specification of the concurrent step, the essential nondeterminism of this step should be described with rewrite rules. On the other hand, the observation of a state should determine a single view of the state. It should be a deterministic computation, which must be described exclusively by equations. The observation-after-concurrent-step pattern induces a cycle of rewrites, where rewrites with the rules and equations that effect the concurrent step are followed by rewrites with equations to observe the state.

This state has two parts. There is a part that models the activity of the network, in which each node is naturally represented by an object, and each object-node is able to transmit messages. There is also a part that holds the observations and the metrics derived from them. These are global properties of the network, so they should not be part of the state of any individual node-object. They too are represented by objects, metrics-objects, which do not send or receive messages.

The cycle of rewrites imposes some constraints on these parts during state transitions. During a concurrent step only the state of node-objects may change. In turn, during an observation only the state of the metrics-objects may change. The second principle advises that when a part of the state cannot change, there should not be even a failed attempt to change it.

Maude provides mechanisms to control rewrites with rules and rewrites with equations, which make this possible. (See [2] for details.) For each, the declaration of an operator may specify argument positions in which that kind of rewrite is precluded — with the attribute `frozen` to prevent rewrites with rules, and the attribute `strat` to prevent rewrites with equations. Any term with that operator at the top would have its subterms at those positions unchanged by rewrites of that kind. In the case the subterms have no redexes, the proscription has no effect on the result of rewriting the term, *but it improves the rewriting performance by reducing the search space explored.* Therefore, so that there be no failed attempts to rewrite a part of the state that remains unchanged, the state should have structure. This is defined by an operator with two arguments, whose declaration precludes rewriting on the part of the state that should remain unchanged.

Since a part of the state that remains unchanged in one part of the cycle, may change in the other, the state must be represented with different constructors in each part of the cycle. Denote the part of the state with the metrics-objects by s_d; this is the part that is exclusively subject to deterministic computation. Denote the part with the node-objects by s_{nd}; this is the part that is subject to nondeterministic

computation. During the concurrent step the state is represented as $s_d :: s_{nd}$,

```
op _::_ : Configuration Configuration -> Configuration
        [ctor frozen (1) strat (2 0) ] .
```

where the operator _ :: _ precludes rewriting with rules in s_d. In the rest of the cycle, no part of the state may be rewritten with rewrite rules, but all of it may be rewritten with equations.

```
op _:::_ : Configuration Configuration -> Configuration
        [ctor frozen (1 2)] .
```

In this part of the cycle the state is represented by $s_d ::: s_{nd}$.

The second principle can be applied again, this time to add structure to s_{nd} to further improve the efficiency of executing the specification. In any state, s_{nd} can always be partitioned into a subconfiguration s_{en} in which every object or message is in some substate that enables a transition, and a subconfiguration s_{nen} in which no object or message is in such a substate. So s_{nd} can be represented by s_{nen} `#|` s_{en}, where the operator _ `#|` _ allows no rewrites on s_{nen}, and rewrites of both kinds on s_{en}. Such an optimization may be applied once during each cycle: to the state in which no enabled transition is concurrent with one already taken, that is, before the concurrent step.

With this optimization the cycle takes the following form:

- optimize $s_d ::: s_{nd}$
- take the concurrent step on $s_d :: s_{nd}$
- observe on $s_d ::: s_{nd}$

It defines a versatile representation of the state of the model of the sensor network that adapts efficient execution and monitoring to the current concurrent step.

Time is a significant parameter for a sensor network, since its scarce energy resource limits its lifetime, but it also serves as a device to define a concurrent step and implement the cycle of rewrites. In a timed specification each transition is taken at a given time. So a concurrent step is the execution of all transitions that are taken at the same time, and the cycle of rewrites becomes:

- optimize $s_d ::: s_{nd}$ at time t
- execute all transitions taken on $s_d :: s_{nd}$ at time t
- observe $s_d ::: s_{nd}$ at time t
- advance time

It is a framework for a timed specification of a sensor network.

This framework is defined in Real-Time Maude by specializing and exploiting the scheme that Ölveczky and Meseguer introduced to model time for complex systems [11][9] (see page 4). Only the last step of the cycle advances time. It is defined by specializing the scheme to constrain the size of the state space. The remaining, instantaneous steps of the cycle are defined by manipulating the definition of `mte` and the representation of the state.

The state that gets transformed by the advance of time is the final state of the third step of the cycle. So the tick rule is defined for a state as represented in that

deterministic step. The scheme of Ölveczky and Meseguer describes the advance of time in terms of function `mte`, which gives the maximum elapse of time possible that satisfies time constraints; and function `delta`, which describes the effect of that advance on the state of the system. To minimize the number of states, the maximum allowed for the elapse of time is prescribed to be also the minimum. This is one of the strategies Real-Time Maude offers for nondeterministic models of time. Here time advances in the same eager way, but in a deterministic model of time:

$$\text{crl } \{t\} \longrightarrow \{\texttt{delta}(t, \texttt{mte}(t))\} \text{ in time } \texttt{mte}(t) \text{ if } \texttt{mte}(t) :: \texttt{Time} \ .$$

Both `mte` and `delta` distribute over the structure of the state, as represented in deterministic steps of the cycle.

$$\texttt{mte}(s_d ::: s_{nd}) = \texttt{mte}(s_{nd})$$

$$\texttt{delta}(s_d ::: s_{nd}) = \texttt{delta}(s_d) ::: \texttt{delta}(s_{nd})$$

The tick rule for the cycle of rewrites is defined as follows:

$$\begin{aligned} \text{crl} \quad & \{s_d ::: s_{nd}\} \longrightarrow \{\texttt{delta}(s_d ::: s_{nd}, \texttt{mte}(s_d ::: s_{nd}))\} \\ & \text{in time } \texttt{mte}(s_d ::: s_{nd}) \\ & \text{if mte}(s_d ::: s_{nd}) :: \text{Time} \ . \end{aligned}$$

As long as $\texttt{mte}(s_d ::: s_{nd})$ is 0 time will not advance. Neither will it advance if $\texttt{mte}(s_d ::: s_{nd})$ is `INF`, which means that there is no constraint of the elapse of time. This rule advances time only if there is some constraint that requires the advance.

The cycle of rewrites can now be defined. The first step

$$\{s'_d ::: s'_{nd}\} = \{s_d :: \mathit{Optimize}(s_{nd})\} \text{ if } \texttt{mte}(s_{nd}) = 0$$

optimizes the state, and prepares it for the nondeterministic step of the cycle. The function `mte` is defined so that when all events constrained to take place at the current time have occurred, $\texttt{mte}(s_{nd})$ becomes positive. Thus, the nondeterministic, concurrent step of the cycle

$$\{s_d :: s_{nd}\} \rightarrow^* \{s'_d :: s'_{nd}\}$$

continues until $\texttt{mte}(s'_{nd})$ becomes positive. At this point time may not advance because, as represented here, the state cannot satisfy the condition of the tick rule. The observation step of the cycle is defined by the following equation:

$$\{s'_d :: s'_{nd}\} = \{\mathit{Observe}(s_d ::: s_{nd})\} \text{ if } \texttt{mte}(s_{nd}) > 0$$

which observes the state, and prepares it for the tick rule.

Constructed following two obvious principles, which impose a high-level structure on computations and states, this is a framework for efficient and flexible specifications of sensor networks.

5 Observations

An object o of class `Observation` is part of s_d. It holds global information about s_{nd}, obtained by "observing" s_{nd} through the operation o `<-` s_{nd}. A subclass of

```
class FloodingNode | *** others
    received-message-at : TimeInf, sent-message : Bool . *** at this time
```

Fig. 1. `FloodingNode` attributes associated with metrics.

```
eq  (< M : Metrics | collisions : C,  transmissions : K,
        have-received : R,  latency : INF >)
 <- (< N : FloodingNode | clock : T,  heard : H,
        sent-message : B,  received-message-at : T  >)
 =  < M : Metrics | collisions : C + if H > 1 then H else 0 fi,
       transmissions : K + if B then 1 else 0 fi,  have-received : s R > .
ceq (< M : Metrics | collisions : C,  transmissions : K,  latency : iT >)
 <- (< N : FloodingNode | clock : T,  heard : H,
        sent-message : B,  received-message-at : iT' >)
 =  (< M : Metrics | collisions : C + if H > 1 then H else 0 fi,
        transmissions : K + if B then 1 else 0 fi >)
  if not( iT == INF and T == iT' ) .
```

Fig. 2. Observing a `FloodingNode`.

`Observation` is used to obtain metrics for the flooding example.

```
class Observation  .
op _<-_ : Object Configuration -> Object .
eq V <- (none).Configuration = V .
ceq V <- (C1 C2) = (V <- C1) <- C2  if C1 =/= none and C2 =/= none .
```

Note that `Observation V` contains some previously gathered data. The term `V <- C` denotes an `Observation` object that combines data gathered from `Configuration C` with that in `V`. The configuration `C` remains unchanged by the `<-` operation.

In Maude, an object-oriented system is represented as a multiset of objects and messages. The "observation" function `<-` should respect this structure, satisfying the following conditions.

```
eq V <- (C1 (C2 C3)) = V <- ((C1 C2) C3) [nonexec] .
eq V <- (C2 C1) = V <- (C1 C2)           [nonexec] .
```

Consider the flooding example. The goal of flooding is that a message sent by a node be received by all other nodes in the network. It is important to determine whether this goal is achieved, and furthermore, to obtain metrics of interest. For this protocol they include latency, or the time it takes to deliver the message to all nodes; the cumulative number of transmissions this takes; and the number of collisions that occur while achieving this goal

```
class Metrics |
    clock : Time,  done : Bool,  collisions : Nat,
    transmissions : Nat,  all-received : Bool,
    have-received : Nat,  latency : TimeInf .
subclass Metrics < Observation .
```

To collect these metrics each `FloodingNode` is observed, and then it is determined whether the message reached all. Figure 1 shows attributes of `FloodingNode` that support the collection of metrics. Figure 2 defines the observation function `<-` for a `FloodingNode`, and Figure 3 defines the operation `collect`, which is an example of *Observe* in the cycle of rewrites.

6 Lazy Configurations

Section 4 describes how s_{nd} is optimized for efficient execution by partitioning it into s_{nen} `#|` s_{en}. A lazy configuration is a further optimization in which s_{nen} is

```
op collected : Object -> Object .
eq collected(< M : Metrics | clock : T,  have-received : R,  latency : INF >)
 = < M : Metrics | done : true,  all-received : R == #nodes,
       latency : if R == #nodes then T else INF fi > .
eq collected(< M : Metrics | latency : T >) = < M : Metrics | done : true > .

op collect_ : Configuration -> Configuration .
eq collect( (< M : Metrics | done : false > Os) ::: Ns )
  = (collected( < M : Metrics | >  <-  Ns) Os) ::: Ns .
```

Fig. 3. Collecting metrics from `FloodingNodes`.

partitioned into two subconfigurations guided by a third principle: *If a smaller size of the state suffices, have it not larger.* The original optimization improves rewriting performance because the search space is confined to s_{en} instead of $s_{en}s_{nen}$. The size of the state is effectively reduced for the current concurrent step. In a lazy configuration the size of the states of the nodes in one of the subconfigurations of s_{nen} is actually reduced, at least for the current cycle. In later cycles those nodes may recover their full-size state and become part of s_{en}. What underlies this further optimization is the nature of propagation of messages in sensor networks. Though less general than the original optimization, it should be applicable in modelling many protocols for sensor networks.

The limited range of communication of wireless nodes means that a message will get to a distant node by being passed from neighbor to neighbor; a sensor network is a multihop network. In many protocols, a message sent by one node induces a propagation of messages. At a given moment different nodes may have different levels of engagement in a propagation. Some may be actively involved. Others may temporarily be bystanders, but soon will be participating again. Still others may be far removed from the propagation, entirely uninvolved.

The running example illustrates this. A node that is actively involved is in s_{en}, which means that either it has a nonempty queue of messages with a message ready to be sent at the current time, or is a neighbor of such a node. There are differences among the remaining nodes. Some have nonempty message queues, but no message ready to transmit. They are waiting to transmit later. Others have empty queues. These are the ones that are at least temporarily entirely uninvolved.

It may be appropriate to model nodes so that a node fully engaged in the activity of the network has a larger state than that of a node far removed from the current activity. The state of one of the latter nodes for the running example illustrates this point.

```
< node(28) : CB-FloodingNode |
    clock : 22,
    neighbors : 8 U 29,
    received-message-at : INF,
    received : 0,
    messages : emptyMessageQueue,
    heard : 0,
    sent-message : false >
```

The last three attributes are associated with activity. The empty message queue indicates that this node is not scheduled to send messages neither now nor soon. Attribute `heard` keeps track of messages heard. Only neighbors of transmitting nodes have a positive value for this attribute. Attribute `sent-message` indicates whether the node sent a message at the current time. For all nodes far removed

from a propagation these three attributes have the same values. These are attributes whose values are more transient than the previous three. Following the advice of the third principle, the state of this node is represented by this smaller state:

```
< node(28) : DormantCB-FloodingNode |
    neighbors : 8 U 29,
    received-message-at : INF,
    received : 0 >
```

Time also becomes superfluous when the node is far removed from any activity. Should a propagation reach it, the larger state is restored, with the value of time that the rest of the active nodes currently have.

A lazy configuration may represent s_{nd} whenever the model of the protocol distinguishes levels of inactivity. It consists of two inactive configurations — a "waiting" and an "active" one — and s_{en}, the "active" configuration. The constructor for the lazy configuration

```
op _#|_|#_ : Configuration Configuration Configuration
          -> Configuration [ctor strat (2 0) frozen (1 3)] .
```

allows rewrites only in its second argument, the active configuration.

A lazy configuration must be reestablished at each cycle. The transformation begin with the tick rule.

```
eq mte(Cw #| Ca |# Cd) = mte(Cw Ca) .
eq delta(Cw #| Ca |# Cd, T)
  = none #| delta(deactivate(Cw Ca), T) |# Cd  .
```

The `deactivate` operation identifies the part of the configuration that would be likely to become dormant. For the rest of the configuration, `deactivate` acts as the identity function. In the flooding example, `deactivate` identifies nodes with empty `messages` queues as the candidates. When the tick rule is taken, the waiting times of the messages in the queues are updated, and a new set of enabled transitions is determined by nodes with messages ready to be transmitted. At this point, in the first step of a new cycle, the configuration is repartitioned. Whether a node remains dormant, or is "awakened" depends on whether one of its neighbors is ready to transmit. More generally, the *Optimize* function completes the repartition according to the interpretation of waiting, active and dormant.

7 Shrinkable Configurations

In some protocols, or phases of protocols, at some point the participation of a node ends. Guided by the third principle — *If a smaller size of the state suffices, have it not larger.* — those nodes might be removed from the representation of the state. For example, this might be reasonable when a node fails. In the flooding example it becomes possible when `CB-FloodingNode`s (page 6) have retransmitted the maximum allowed number of times.

At each cycle there may be new nodes to be removed. This transformation of the state begins with the tick rule. A `vanish` function identifies the part of the state to be removed. Then the advance of time is defined as follows:

```
eq  delta( (OCw #| OCa |# OCd), T )
  = none #| delta(deactivate( vanish(OCw OCa)), T) |# OCd .
eq  mte(vanish(O)) = INF .
```

```
eq  delta(deactivate(vanish(0)), T) = vanish(0) .
```

8 Experiments

The specification of flooding for `CB-FloodingNodes` (page 6) has as parameters the number of nodes in the network, `#nodes`; and the maximum number of times a node may retransmit, `#received-limit`. A few configurations were studied to reveal whether the message would reach all nodes, what value of `#received-limit` this would require, and what the associated metrics were. For the smallest configurations the three different representations for s_{nd} were used: ordinary configurations, lazy configurations and shrinkable configurations. For larger ones, only the shrinkable configurations were used. To help determine the efficacy of the techniques, when lazy or shrinkable configurations were used, the `Metrics` objects tracked how many nodes were active at each tick. A `Metrics` object was generated at each tick, reporting the metrics until then. This was the most informative part of the state.

The timed rewrite (`trew`) and untimed model checking commands of Real-Time Maude were used [8]. The timed rewriting of some initial configuration may have one of three results: the `Metrics` objects reveal that the message reached all nodes, what the associated metrics (and any meta-metrics) were; the `Metrics` objects reveal that the message has not reached all, but an examination of the node configuration shows it may still be possible; and, lastly, `Metrics` objects and node configuration show the message will not reach all. In the second case the computation is resumed; the final state of the previous try becomes the configuration to be rewritten for some specified time. In the last case the limit of retransmissions is incremented.

If the timed rewriting showed the message reached all, the latency l, and associated metrics — transmissions t and collisions c — the following untimed model checking command was used to check that all computations with initial state `init` satisfy the linear temporal logic formula `<> all-received-by` l `w` t `transmissions-&` c `collisions` [10] [8].

```
(mc init
    |=u <> all-received-by l w t transmissions-& c collisions .)
```

where

```
op  all-received-by_w_transmissions-&_collisions
    : Time Nat Nat -> Prop [ctor] .
eq  {(OC < mets : Metrics | done : true, clock : T',latency : T',
                transmissions : M', collisions : N' > )  ::: C }
    |= all-received-by T w M transmissions-& N collisions
  = T' <= T and-then (M' <= M and-then N' <= N) .
```

Counterexamples provide new triples to try.

The following tables show some results. The table on the left gives the results of four simulations: the latency l, and number of transmissions t and collisions c, for configurations with n nodes, and retransmission limit rL. The table on the right shows the effectiveness of the techniques: the cpu time for the simulation, and

average number of active nodes.

k	n	rL	l	t	c
c_1	20	2	32	26	10
c_2	30	2	43	45	14
c_3	30	3	43	61	22
c_4	40	3	53	97	32

k	type	cpu time	% active
c_1	ordinary	1751 sec	100
	lazy	134 sec	47
	shrinkable	62 sec	43
c_2	shrinkable	127 sec	28

The table below gives some model-checking results.

k	l	t	c	result	type	cpu time
c_2	32	26	10	true	ordinary	97,416 sec
					lazy	4,996 sec
					shrinkable	1,810 sec
c_3	43	70	56	counterex	shrinkable	798,076 sec

9 Conclusion

A high-level, modular approach that exploited features of Maude and was guided by three obvious principles led to the cycle of rewrites, which is a fairly general framework for the specification of sensor networks. It can serve more generally as a framework for the specification of real-time systems. The structure the cycle imposes — on computations and states — is what jointly achieves efficient execution and flexible monitoring.

Lazy configurations and shrinkable configurations, which are justified by the behavior of some protocols, introduce further structure and reduce the size of the state, yielding further speedups in execution.

The framework presented here only constrains the size of the state space. Future research will apply the approach that very significantlhy improved the efficiency of execution to actually reduce the size of the state space.

Acknowledgement

My appreciation and thanks to the anonymous referees and Program Committee Chair Carolyn Talcott for their helpful suggestions.

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science,* 285(2):187–243, 2002.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual (Version 2.2). December 2005, http://maude.cs.uiuc.edu http://maude.cs.uiuc.edu.

[3] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[4] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. Complex Behavior at Scale: An Experimental Study of Low Power Wireless Sensor Networks. *Technical Report UCLA/CSD-TR 02-0013, Computer Science Department, UCLA*, July 2002.

[5] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of ACM International Conference on Mobile Computing and Networking (MOBICOM'00)*, pp. 56–67, 2000.

[6] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[7] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. *The broadcast storm problem in a mobile ad hoc network.* In Proc. of the 5th annual ACM/IEEE Int. Conference on Mobile Computing and Networking, ACM Press, pp. 151–162, 1999.

[8] P. C. Ölveczky. Real-Time Maude 2.1 Manual. October 2005, http://maude.cs.uiuc.edu.

[9] P. C. Ölveczky, and J. Meseguer. Real-Time Maude 2.1. In *Fifth International Workshop on Rewriting Logic and its Applications, 2004*. Electronic Notes in Theoretical Computer Science, 117:285–314, 2004.

[10] P. C. Ölveczky, and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation,* to appear.

[11] P. C. Ölveczky, and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science,* 285:359–405, 2002.