

Towards Formal Modeling and Analysis of Networks of Embedded Medical Devices in Real-Time Maude

Peter Csaba Ölveczky
Department of Informatics, University of Oslo, Norway

Abstract

With the increasing number of medical devices and of accidents resulting from them being used in isolation in a hectic operating room, there is a trend towards integrating such devices into networks. This paper describes the application of the Real-Time Maude tool to the formal modeling and analysis of a network integrating an x-ray machine, a ventilation machine, and a controller. This case study is motivated by an accidental death in an operating room. As part of the formal specification and analysis, the paper introduces novel techniques for: (i) modeling nondeterministic transmission delays while maintaining completeness and reasonable performance of the analysis; (ii) modeling clock drifts; and (iii) analyzing bounded response properties.

1 Introduction

Current medical devices are still mostly stand-alone systems (see, e.g., [12]). Medical workers therefore often need to manually monitor and operate different devices in a hectic operating room. This is error-prone, and may lead to tragic accidents such as the following, reported in [6]:

A 32-year-old woman had a laparoscopic cholecystectomy performed under general anesthesia. At the surgeon's request, a plane film x-ray was shot during a cholangiogram. The anesthesiologist stopped the ventilator for the film. The x-ray technician was unable to remove the film because of its position beneath the table. The anesthesiologist attempted to help her, but found it difficult because the gears on the table had jammed. Finally, the x-ray was removed, and the surgical procedure recommenced. At some point, the anesthesiologist glanced at the EKG and noticed severe bradycardia. He realized he had never restarted the ventilator. This patient ultimately expired [died].

That is, the ventilator that helped the patient breathe during the surgery was briefly turned off to allow an X-ray to be taken without blurring the picture. The X-ray machine

jammed, the anesthesiologist went to fix the X-ray but forgot to turn on the ventilator, leading to the patient's death.

To avoid such tragedies, next generation medical systems are envisioned to integrate medical devices into networks of such devices (see, e.g., [5, 3]). Given the criticality of such networks, there is a clear need for formally analyzing the networks before they are deployed. This paper describes the application of the Real-Time Maude tool [9] to the formal modeling and analysis of a small interlock protocol for such networks. The example was proposed by a domain expert and is motivated by the above tragedy, and describes an interlock protocol for a controller, an X-ray device, and a ventilation machine. Although the example is quite small, it poses interesting modeling and analysis questions.

Real-Time Maude is a high-performance tool that extends the rewriting logic-based Maude system [2] to support the formal specification and analysis of object-based real-time systems. The tool emphasizes ease and expressiveness of specification, and provides a spectrum of analysis methods, including simulation through timed rewriting, temporal logic model checking, and search for reachability analysis. Real-Time Maude has proved useful for analyzing advanced communication protocols [10], embedded car software, and state-of-the-art wireless sensor network [11, 4] and scheduling [7] algorithms.

The contributions of this paper are the following:

1. It presents a Real-Time Maude solution to an (admittedly small) example that might become a benchmark for networks of embedded devices.
2. It provides a new model for nondeterministic messaging delays that allows efficient model checking in the presence of such delays.
3. It shows how *clock drifts* can be modeled.
4. It presents a way of model checking *bounded response* properties.

2 Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* of the form (Σ, E, IR, TR) , where:

- (Σ, E) is a *membership equational logic* [2] theory with Σ a signature¹ and E a set of conditional equations. (Σ, E) specifies the system’s state space as an algebraic data type, and must contain a specification of a sort `Time` modeling the (discrete or dense) time domain.
- IR is a set of *labeled conditional instantaneous rewrite rules* specifying the system’s *instantaneous* (i.e., zero-time) local transitions, each of which is written `cr1 [l] : t => t' if cond`, where l is a *label*. Such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t' , *provided* the condition holds. The rules are applied *modulo* the equations E .² *Unconditional* rewrite rules are written with syntax `rl [l] : t => t'`.
- TR is a set of *tick (rewrite) rules*, written with syntax `cr1 [l] : {t} => {t'} in time τ if cond` that model time elapse. `{_}` is a built-in constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The initial states must be ground terms of sort `GlobalSystem` and must be reducible to terms of the form `{t}` using the equations in the specifications.

A *class declaration*

```
class C | att1 : s1, ... , attn : sn .
```

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n . An *object* of class C in a given state is represented as a term `< O : C | att1 : val1, ..., attn : valn >` where O is the object’s *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n . In a concurrent object-oriented system, the state, which is usually called a *configuration*, is a term of the built-in sort `Configuration`. It has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude. The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

```
rl [1] :
  m(O, w)
  < O : C | a1 : x, a2 : O', a3 : z > =>
  < O : C | a1 : x + w, a2 : O', a3 : z >
  m'(O') .
```

¹i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols* (or *operators*)

² E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A . Operationally, a term is reduced to its E' -normal form modulo A before any rewrite rule is applied.

defines a family of transitions in which a message m , with parameters O and w , is read and consumed by an object O of class C . The transitions change the attribute a_1 of the object O and send a new message $m'(O')$. “Irrelevant” attributes (such as a_3) need not be mentioned in a rule.

The specification of the medical system follows the techniques given in [9] for specifying the time-dependent behavior of object-oriented real-time systems. Time elapse is modeled by the tick rule

```
var C : Configuration .    var T : Time .
cr1 [tick] : {C} => {delta(C, T)} in time T
              if T <= mte(C) .
```

The function `delta` defines the effect of time elapse on a configuration, and the function `mte` defines the maximum amount of time that can elapse before some action must take place. These functions distribute over the objects and messages in a configuration and must be defined for single objects and messages. The tick rule advances time *nondeterministically* by *any* amount T less than or equal to `mte(C)`. To execute such rules, Real-Time Maude offers a choice of *time sampling strategies*, so that only *some* moments in time are visited. The choice of such strategies includes:

- Advancing time by a fixed amount Δ in each application of a tick rule.
- The *maximal* strategy, that advances time to the next moment when some action must be taken, as defined by `mte`. This corresponds to *event-driven simulation*.

Real-Time Maude’s *timed “fair” rewrite* command simulates *one* behavior of the system *up to a certain duration*. It is written with syntax `(tfrew t in time <= τ .)`, where t is the initial state and τ is a term of sort `Time`.

Real-Time Maude’s *timed search* command uses a breadth-first strategy to analyze all possible behaviors of the system, relative to the selected time sampling strategy, by checking whether a state matching a *pattern* and satisfying a *condition* can be reached from the initial state t within time τ . The command which searches for *one* state satisfying the search criteria has syntax

```
(tsearch [1] t =>* pattern such that cond
  in time <=  $\tau$  .)
```

Real-Time Maude also extends Maude’s *linear temporal logic model checker* [2] to check whether each behavior satisfies an *untimed* temporal logic formula. Finally, the `find latest` command finds how long it takes, in the worst case, to reach a desired state.

3 The Example

The example, proposed by a domain expert, interconnects a controller, a ventilation machine, and an x-ray de-

vice. The protocol is described in Section 5, and the requirements that the system should satisfy are given in Section 7. The message delay is supposed to be nondeterministic, bounded by a maximum delay, and the internal clocks of the devices may *drift*, i.e., go slower or faster than “real” time. Although the various components and the network may go down temporarily, there is no space in this paper to address the treatment of such failures.

4 An Efficient Model for Nondeterministic Message Delays

The end-to-end messaging delay is supposed to be nondeterministic in the X-ray case study, where the delay can be *any* time value in an interval $[minDly, maxDly]$. The straight-forward way to model such messaging delay in Real-Time Maude is to let the rule modeling the consumption of the message be enabled at *any* time within the time interval, and then use the time sampling strategy that advances time by a user-given default time increment (e.g., one time unit) in each application of the tick rule. However, analyzing such an intuitive model would be very inefficient in our case, where, during a “round” of 60 000 ms, messages are only present for a small fraction of that time. “Default” time sampling would force time advance to stop “unnecessarily” at each time unit. This leads to unnecessarily large states spaces that make exhaustive analysis unfeasible.

This paper presents a new model supporting *maximal time sampling* analysis of systems with nondeterministic message, so that time progress is not unnecessarily stopped when no message is in the state. The idea is simple: a message is equipped with an additional “timer.” When this timer expires, time progress stops, and either the message becomes “ripe” and must be read before time can progress further, or the timer is reset to a value δ , which denotes the time interval between each such “check” for message ripeness. For discrete time, if δ is one time unit, the message delay can indeed be any value in the given interval.

The sender sends a message m with nondeterministic message delay in the interval $[minDly, maxDly]$ by creating a message

$$dly(m, minDly, maxDly)$$

which is transformed into a term

$$dly(m, minDly, maxDly, \delta)$$

where the second parameter now is the timer value which denotes the time until the next check of message ripeness; the third parameter denotes the time until the longest possible message delay; and the fourth parameter is the time interval between successive checks for message ripeness:

```
sort DlyMsg .
  subsorts Msg < DlyMsg < NEMsgConfiguration .
op dly : Msg Time Time -> DlyMsg .
op dly : Msg Time NzTime Time -> DlyMsg [ctor] .

op  $\delta$  : -> NzTime .

var M : Msg .      vars T T' T'' : Time .
vars NZT NZT' : NzTime .

eq dly(M, T, T') = dly(M, T, T',  $\delta$ ) .
```

Whenever the “timer” expires (is 0), the message can become “ripe” (rule `msgRipe`), or it can reset its timer to check again after the chosen time interval (rule `contDly`) if the remaining *maximal* delay (denoted by a variable NZT of the sort NzTime of *non-zero* time values) is greater than 0. When the remaining maximal delay is 0, the message becomes ripe by the equation:

```
r1 [msgRipe] : dly(M, 0, NZT, NZT') => M .
r1 [contDly] : dly(M, 0, NZT, NZT') =>
                dly(M, min(NZT, NZT'), NZT, NZT') .
eq dly(M, T, 0, NZT) = M .
```

The function `delta` defines the effect of time elapse on delayed messages by decreasing the timer and the remaining maximal delay according to the elapsed time:

```
eq delta(dly(M, T, T', NZT), T'') =
    dly(M, T monus T'', T' monus T'', NZT) .
eq delta(M, T) = M .
```

The function `mte` ensures that time never advances beyond the expiration time of a timer, and that a ripe message m has $mte(m) = 0$, so that it must be consumed before time can progress in the system:

```
eq mte(dly(M, T, T', NZT)) = min(T, T') .
eq mte(M) = 0 .
```

In this model, and with maximal time sampling analysis, time progress stops at each instant when a message *could* be ripe, but otherwise time progress does not stop unnecessarily when no message is in the system (by far the most time in the X-ray example), allowing us to model check the X-ray system for behaviors up to hundreds of millions of milliseconds.

5 Modeling the Network Without Drifts

This section presents the Real-Time Maude model of the network of medical devices. For simplicity, this first model does not consider clock drifts (or equipment and network failures). Clock drifts are treated in Section 6.

Real-Time Maude has proved particularly useful for modeling distributed real-time systems in an object-oriented style [10, 11, 7]. The X-ray system is therefore

modeled in an such a style, even though the network contains only one device of each kind. Nothing is lost by applying object-oriented techniques, since this does not increase the number of states encountered in exhaustive analysis.

5.1 The Controller

The controller component is described as follows in the informal specification:

```
loop
block on (button_is_pushed signal);
check persistent memory for last_pause_time;
if ((current_time - last_pause_time)
    < (10 min + 10 msec))
    {tell user to wait until
      (last_pause_time + 10 min + 10 msec);
     exit;}
else {
  update last_pause_time in persistent memory to
    (current_time + 3 seconds);
  command VM to pause 2 seconds at
    current_time + 1 second;
  command XM to take a photo at
    current_time + 2 second;
}
end-loop
```

That is, when the user pushes a button, the controller checks whether at least 10 minutes + 10 milliseconds have passed since the ventilation machine was turned off. If so, the controller sends a message to the ventilation machine to take a two seconds pause after one second, and sends a message to the X-ray device to take an X-ray after two seconds.

The Controller class is declared as follows:

```
class Controller | clock : Time,
                  lastPauseTime : Time .
```

The attribute `clock` is the *local* clock of the controller ('current_time'), and the attribute `lastPauseTime` corresponds to `last_pause_time`.

The following message types are needed:

```
msg pushButton : -> Msg .
msg to `VentMachine`pause `2sec`in `1sec` : -> Msg .
msg to `X-ray`takeXray `in` `2sec` : -> Msg .
msg userWaitUntil_ : Time -> Msg .
```

The message `pushButton` models the push of the button by some user; the message `to VentMachine pause 2sec in 1sec` tells the ventilation machine that to pause for two seconds one second after getting the message; the message `to X-ray takeXray in 2sec` tells the X-ray device to take a photo two seconds after receiving the message; and the message `userWaitUntil t` tells the user to wait until time t to push the button.

The behavior of the controller is given by the following rewrite rule:

```
vars T T' T'' : Time .      vars C V X U : Oid .

r1 [readPushButtonMsg] :
  pushButton
  < C : Controller | clock : T, lastPauseTime : T' >
=>
  if not tooEarly(T, T') then --- take X-ray
    (< C : Controller | lastPauseTime : T + 3000 >
     dly(to VentMachine pause 2sec in 1sec,
        MIN-DELAY, 50)
     dly(to X-ray takeXray in 2sec, MIN-DELAY, 50))
  else --- too early!
    (< C : Controller | >
     dly(userWaitUntil (T' + 600000 + 10), 0, 50))
  fi .

op tooEarly : Time Time -> Bool .
eq tooEarly(T, T') = if T' == 0 then false else
                    (T - T') < (600000 + 10) fi .
```

When the Controller reads a `pushButton` message, the rule checks whether it is too early to take an X-ray. If so, a message is sent to the user asking him/her to wait until the given time. Otherwise, the controller updates its `lastPauseTime` attribute and sends messages to the ventilation and X-ray machines. These messages are assigned a delay interval `[MIN-DELAY, 50]`.

The time-dependent behavior of a controller object is defined by `mte` and `delta`. In our case, the controller alone should never force time elapse to stop. (Since `mte(pushButton) = 0`, time cannot advance when a `pushButton` message is present in the state.) Therefore, `mte` is defined as follows for Controller objects:

```
eq mte(< C : Controller | >) = INF .
```

The function `delta` defines the effect of time elapse on a controller object. In our case, the `clock` attributes should be increased according to the time elapsed:

```
eq delta(< C : Controller | clock : T >, T') =
  < C : Controller | clock : T + T' > .
```

5.2 The Ventilation Machine

The behavior of the ventilation machine is described informally as follows:

```
loop
  block on (pause command)
  {pause at (t + 1) for 2 seconds;}
endloop
```

That is, when it gets its pause request, it waits for one second, and then pauses for two seconds. The corresponding class has only one attribute, the state of the device:

```
class VentMachine | state : BreathingState .

sort BreathingState .
op breathing : -> BreathingState .
ops breatheUntil stopBreathing : Time
  -> BreathingState .
```

When the machine is in state `breathing`, it breathes normally. When it is in state `breatheUntil(t)`, the machine is still breathing, but will turn off its breathing in time t . Finally, in state `stopBreathing(t)`, the machine is pausing (i.e., not breathing) and will pause for t more time units. Therefore, `breatheUntil` and `stopBreathing` can be seen as timers.

The dynamics of the ventilation machine is defined by the following rules:

```

rl [VMreadPause] :
  (to VentMachine pause 2sec in 1sec)
  < V : VentMachine | >
=>
  < V : VentMachine | state : breatheUntil(1000) > .

rl [stopBreathing] :
  < V : VentMachine | state : breatheUntil(0) >
=>
  < V : VentMachine | state : stopBreathing(2000) > .

rl [restartBreathing] :
  < V : VentMachine | state : stopBreathing(0) >
=>
  < V : VentMachine | state : breathing > .

```

To define the timed behavior, the function `mte` is defined to return the time until the next timer expires (and is the infinity value `INF` when the object is in state `breathing`), and the function `delta` defines the effect of time elapse on an object by decreasing the “timer” values according to the elapsed time (the definitions are straight-forward and are omitted due to lack of space).

5.3 The X-ray Machine

The interlocking protocol for the X-ray device is described as follows in the informal specification:

```

loop
  block on (x-ray command)
  {take x-ray at (t + 2);}
endloop

```

That is, the device should take an X-Ray two seconds after getting a request.

Again, the corresponding class only has one attribute: the “state” of the device, which is either `idle`, `takingXray`, or is `wait(t)` when an X-ray is supposed to be taken after t time units:

```

class X-ray | state : XRstate .

sort XRstate .
ops idle takingXray : -> XRstate .
op wait : Time -> XRstate .

```

The dynamic behavior of the device is defined in a straight-forward way: When the device reads a

`takeXray` message, it goes to state `wait(2000)` (rule `XrayReadMsg`); when this timer has reached zero, the system takes an X-ray, which is represented by going to a state `takingXray` (rule `takeXray`); finally, after taking an X-ray, the device goes to state `idle` (rule `idle`):

```

rl [XrayReadMsg] :
  (to X-ray takeXray in 2sec)
  < X : X-ray | >
=>
  < X : X-ray | state : wait(2000) > .

rl [takeXray] :
  < X : X-ray | state : wait(0) >
=>
  < X : X-ray | state : takingXray > .

rl [idle] :
  < X : X-ray | state : takingXray >
=>
  < X : X-ray | state : idle > .

```

As for the timed behavior, `mte` is defined to be zero when the device is in state `takingXray`, to model that this is an instantaneous action, so that the `idle` rule must be taken before time can elapse:

```

eq mte(< X : X-ray | state : idle >) = INF .
eq mte(< X : X-ray | state : takingXray >) = 0 .
eq mte(< X : X-ray | state : wait(T) >) = T .

```

Finally, time advance affects an X-ray device by decreasing the wait timer if the device is a countdown state:

```

eq delta(< X : X-ray | state : idle >, T) =
  < X : X-ray | state : idle > .
eq delta(< X : X-ray | state : takingXray >, T) =
  < X : X-ray | state : takingXray > .
eq delta(< X : X-ray | state : wait(T) >, T') =
  < X : X-ray | state : wait(T minus T') > .

```

6 Modeling Clock Drifts

The clocks of the devices are supposed to be subject to *clock drifts*. This section shows how easily our model can be modified to include clock drifts.

Assume that the drift rate of the local clock of, respectively, the controller, the ventilation machine, and the X-ray device is, respectively, Δ_c , Δ_{vm} , and Δ_x :

```
ops  $\Delta_c$   $\Delta_{vm}$   $\Delta_x$  : -> Rat .
```

The local clock of device d advances $(1 + \Delta_d)$ time units when “real” time advances one time unit. (Positive Δ_d therefore means that the local clock is too fast; negative Δ_d means that the device clock is too slow.)

The definition of `delta` must now take the drift into account, so that when time advances by t time units, the local *clocks* increase by $(1 + \Delta_d) * t$ time units, and *timers* decrease by the same amount. The definition of `delta` for the first two devices is therefore modified to:

```

eq delta(< C : Controller | clock : T >, T') =
  < C : Controller | clock : T + (T' + (T' * ΔC)) > .
eq delta(< X : X-ray | state : idle >, T) =
  < X : X-ray | state : idle > .
eq delta(< X : X-ray | state : takingXray >, T) =
  < X : X-ray | state : takingXray > .
eq delta(< X : X-ray | state : wait(T) >, T') =
  < X : X-ray | state :
    wait(T minus (T' + (T' * ΔX))) > .

```

The function `mte` must also be modified accordingly, to reflect the fact that a timer needs different time to expire than its nominal value:

```

eq mte(< C : Controller | clock : T >) = INF .
eq mte(< X : X-ray | state : idle >) = INF .
eq mte(< X : X-ray | state : takingXray >) = 0 .
eq mte(< X : X-ray | state : wait(T) >) =
  T / (1 + ΔX) .

```

There is no need to change any rewrite rules. Furthermore, since only one device uses clock values, there is no need to model clock synchronization.

7 Analyzing the Specification

This section describes how the network can be simulated and model checked w.r.t. properties stated by the domain expert. Some of the properties are *metric* temporal properties, in which the duration between events is important. Real-Time Maude—as well as most other formal tools for real-time systems, with Kronos [13] a notable exception—can only model check *untimed* temporal properties. Prompted by this case study, we have developed a general method for model checking bounded response properties for object-oriented Real-Time Maude specifications. Due to lack of space, this method will be presented elsewhere. Nevertheless, Section 8 shows how to model check the bounded response properties for the X-ray system.

7.1 Defining Initial States

The system is an open system with an unspecified environment. To execute the system, we define a simple “user,” that pushes the button every `pushInterval` milliseconds (triggered by the expiration of the `pushButtonTimer`) and consumes the `userWaitUntil` messages:

```

class User | pushButtonTimer : TimeInf,
  pushInterval : Time .

rl [pushButton] :
  < U : User | pushButtonTimer : 0, pushInterval : T >
=>
  < U : User | pushButtonTimer : T >
  dly(pushButton, MIN-DELAY, 50) .

rl [readUserWait] :
  (userWaitUntil T)

```

```

  < U : User | >
=> < U : User | > .

```

```

eq delta(< U : User | pushButtonTimer : T >, T') =
  < U : User | pushButtonTimer : T minus T' > .
eq mte(< U : User | pushButtonTimer : T >) = T .

```

Finally, the drift rates of the clocks and an initial state are defined:

```

eq ΔVM = -1/10 .   eq ΔC = 1/10 .   eq ΔX = 1/20 .
ops u vm xr ct : -> Oid [ctor] . --- object names
op initState : -> GlobalSystem .
eq initState =
  {< u : User | pushButtonTimer : 0,
    pushInterval : 60000 >
  < ct : Controller | clock : 0, lastPauseTime : 0 >
  < vm : VentMachine | state : breathing >
  < xr : X-ray | state : idle >} .

```

7.2 Defining the Time Sampling Strategy

With our model of communication, all events take place when some “timer” expires. Therefore, as proved in [8], the system can be analyzed using the *maximal* time sampling strategy, which advances time until the next timer expires.

7.3 Simulation

A first quick analysis of the system can be done by Real-Time Maude’s rewrite command, that simulates *one* sequence of rewrite steps from the initial state:

```

Maude> (tfrew initState in time <= 600500 .)

Result ClockedSystem :
  {< ct : Controller | clock : 660033, ... >
  < u : User | pushButtonTimer : 59970, ... >
  < vm : VentMachine | state : breatheUntil(1000) >
  < xr : X-ray | state : wait(1979) >} in time 600030

```

This seems fine: the ventilator will breathe for another 1,000 milliseconds (according to its own clock), and the X-ray will take a photo in about two seconds.

7.4 Requirements to Analyze

The requirements to analyze, as given by the domain expert, are the following:³

1. The ventilation machine must be pausing when an X-ray is taken.
2. The ventilation machine cannot pause for more than *two seconds* each time.
3. The ventilation machine cannot pause more than once in *ten minutes*.

³We consider only the requirements for non-failing components.

4. When the button is pushed, an X-ray should be taken within *three seconds*.

The last three requirements correspond to *metric* properties that cannot be directly analyzed using Real-Time Maude. In this section, we analyze the first requirement and a weaker version of the fourth requirement (which does not hold in general, since it is in conflict with the third requirement). The other requirements are model checked in Section 8.

7.4.1 Checking an Untimed Safety Property.

The first property is easy to check by searching for a “bad” state in which an X-ray is taken (i.e., the `X-ray` object is in state `takingXray`) while the ventilation machine is breathing. The function

```
op isBreathing : BreathingState -> Bool .
var BS : BreathingState .
eq isBreathing(breathing) = true .
eq isBreathing(breatheUntil(T)) = true .
eq isBreathing(BS) = false [owise] .
```

returns `true` when the ventilation machine is breathing. Since the reachable state space is infinite, we restrict our search to behaviors up to 100,000,000 ms (almost 28 hours) to allow the search to terminate:

```
Maude> (tsearch [1] initState =>*
      {C:Configuration
      < xr : X-ray | state : takingXray >
      < vm : VentMachine | state : BS >}
      such that isBreathing(BS)
      in time <= 100000000 .)
```

Real-Time Maude returned ‘No solution’ in 109 seconds.

7.4.2 Taking an X-ray within Three Seconds.

The fourth requirement says that an X-ray is taken within three seconds after the user pushes the button, provided that no X-ray has been taken in the last 10 minutes. To avoid dealing with old button pushes, I analyze the following weaker property: in all possible behaviors from the initial state (in which the user is ready to push the button), an X-ray will be taken within three seconds. Real-Time Maude’s `find latest` command is used to check *both* whether the desired state will always be reached within three seconds, and, if so, how long it takes in the worst case to reach the state:

```
Maude> (find latest initState =>*
      {C:Configuration
      < xr : X-ray | state : takingXray >}
      in time <= 3000 .)

Result: { ... < xr : X-ray | state : takingXray >}
      in time 42100/21
```

The result shows that an X-ray is always taken within 2005 milliseconds, which makes sense, since the maximum total message delay is 100 milliseconds, and 2 seconds in the X-ray device corresponds to 1905 milliseconds “real” time.

8 Model Checking Bounded Response

Using *bounded* temporal operators in some timed temporal logic (see, e.g., [1]), the second requirement (the ventilation machine should not pause for more than two seconds at a time) can be expressed by the *bounded response* formula

$$\Box (\text{“machine pausing”} \longrightarrow \Diamond_{\leq 2sec} \text{“machine breathing”})$$

which states that each state in which the machine is pausing will be followed in two seconds or less by a state in which the machine is breathing.

Real-Time Maude does not support model checking of timed (or “metric”) temporal logic. Instead, I have defined a general method where the timed LTL model checking problem $\mathcal{R}, t_0 \models \Box (p \longrightarrow \Diamond_{\leq r} q)$, for p and q atomic propositions, is transformed into the untimed model checking problem $\tilde{\mathcal{R}}, \tilde{t}_0 \models \Box (C_{p,q} \leq r)$, where $C_{p,q}$ is a “clock” that measures the time since p held without q holding in the meantime. This transformation adds a “clock” $C_{p,q}$ to the system, and updates the clock as follows:

1. If the clock $C_{p,q}$ is turned off, and a state satisfying $p \wedge \neg q$ is reached, the clock is set to 0 and is turned on.
2. The clock is turned off when a state satisfying q is reached.
3. A clock that is on is increased according to the elapsed time in the system.

In particular, for the useful class of “flat” object-oriented Real-Time Maude specifications, this transformation can easily be automated. Details will be presented elsewhere.

For the medical system, the “clock” $C_{p,q}$ is represented by an object of the following class

```
class BRClock | clock : Time, status : OnOff .
sort OnOff .      ops on off : -> OnOff .
```

where the `clock` attribute measures the time during which the machine has been pausing uninterruptedly. The rules defining the behavior of the ventilation machine need to be transformed to the following rules, where the clock’s `status` attribute is `on` when the machine is *not* breathing:

```
r1 [VMreadPause] :
  (to VentMachine pause 2sec in 1sec)
  < V : VentMachine | >   < B : BRClock | >
=>
  < V : VentMachine | state : breatheUntil(1000) >
  < B : BRClock | status : off > .
```

```

r1 [stopBreathing] :
  < V : VentMachine | state : breatheUntil(0) >
  < B : BRCLock | >
=>
  < V : VentMachine | state : stopBreathing(2000) >
  < B : BRCLock | status : on, clock : 0 > .

r1 [restartBreathing] :
  < V : VentMachine | state : stopBreathing(0) >
  < B : BRCLock | >
=>
  < V : VentMachine | state : breathing >
  < B : BRCLock | status : off > .

```

The timed behavior of the BRCLock object is defined to increase its clock according to the time elapsed (without taking the drift of the ventilation machine into account) when the status is on. The specification is otherwise not changed (apart from adding a BRCLock object to the initial state). The following command then checks whether the ventilation machine can pause for more than two seconds:

```

Maude> (tsearch [1] initStateWithClock =>*)
  {C:Configuration
  < br : BRCLock | clock : T:Time >}
  such that T:Time > 2000
  in time <= 300000 .)

Solution 1
C:Configuration -->
  < vm : VentMachine | state : stopBreathing(0) > ...
T:Time --> 20000/9 ; TIME_ELAPSED:Time --> 10000/3

```

The result shows that the requirement does not hold, as the ventilator may pause for 2.22 seconds, since its internal clock is a little slow. Similar analysis shows that the ventilation machine cannot pause for more than 2.5 seconds.

9 Analyzing the Third Requirement

The third requirement says that the ventilator cannot pause more than once in ten minutes. This property can be combined with the second requirement and can be expressed in a future fragment of some metric linear temporal logic as $\square (pausing \longrightarrow \diamond_{\leq 2sec} (\square_{\leq 10min} breathing))$. To analyze this property, I add an additional clock attribute timeSinceLastPause to the ventilation machine, and search for whether this clock, which measures the time since the last pause, can be less than 10 minutes when the ventilation takes another pause.

10 Concluding Remarks

The fairly small case study required addressing some general issues for analyzing networks of embedded devices, including: modeling clock drift, efficiently model checking with nondeterministic messaging delays, and analyzing bounded response properties. The presented techniques

allowed model checking behaviors up to millions of time units. Although the example itself is quite simple, successfully addressing its challenges (except for dealing with failures), together with the fact that Real-Time Maude has successfully been applied to advanced state-of-the-art network and scheduling applications, gives us ample reason to believe that Real-Time Maude is a good candidate for analyzing realistic networks of medical devices.

Follow-up work should develop symbolic analysis techniques to analyze systems which are parametric in, say, the clock drifts, and should further develop methods for model checking metric temporal logic properties in object-oriented Real-Time Maude specifications.

Acknowledgments. I gratefully acknowledge Lui Sha for proposing the case study for Real-Time Maude, and The Research Council of Norway for financial support.

References

- [1] R. Alur and T. Henzinger. Logics and models of real time: A survey. In *Real Time: Theory in Practice*, volume 600 of *LNCS*, pages 74–106. Springer, 1992.
- [2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, *LNCS* 4350, 2007.
- [3] J. Goldberg. Medical device "plug-and-play" interoperability program. http://mdnpn.org/Home_Page.html.
- [4] M. Katelman, J. Meseguer, and J. Hou. Redesigning the LMST wireless protocol by formal modeling and statistical model checking. In *FMOODS'08*, 2008. To appear.
- [5] I. Lee, G. J. Pappas, R. Cleaveland, J. Hatcliff, B. H. Krogh, P. Lee, H. Rubin, and L. Sha. High-confidence medical device software and systems. *IEEE Computer*, 39(4), 2006.
- [6] A. S. Lofsky. Turn your alarms on! *APSF Newsletter: The Official Journal of the Anesthesia Patient Safety Foundation*, 19(4):43, 2005.
- [7] P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In *FASE'06*, volume 3922 of *LNCS*, 2006.
- [8] P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):5–27, 2007.
- [9] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
- [10] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29:253–293, 2006.
- [11] P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. In *FMOODS'07*, *LNCS* 4468, 2007.
- [12] L. Sha and A. Agrawala, editors. *Report of NSF Workshop on Distributed Real-Time and Embedded Systems Research in the Context of GENI*, 2006.
- [13] S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1–2), 1997.