Formal Real-Time Model Transformations in MOMENT2

Artur Boronat[1] and Peter Csaba Ölveczky[2]

[1] Department of Computer Science, University of Leicester
[2] Department of Informatics, University of Oslo

Abstract. This paper explains how the MOMENT2 formal model transformation framework has been extended to support the formal specification and analysis of real-time model-based systems. We provide a collection of built-in timed constructs for defining the timed behavior of model-based systems that are specified with in-place model transformations. In addition, we show how an existing model-based system can be extended with timed features in a *non-intrusive* way (i.e, without modifying the class diagram) by using in-place *multi-domain* model transformations supported in MOMENT2. We give a real-time rewrite formal semantics to real-time model transformations, and show how the models can be simulated and model checked using MOMENT2's Maude-based analysis tools. In this way, MOMENT2 becomes a flexible, effective, automatic tool for specifying and verifying model-based real-time and embedded systems within the Eclipse Modeling Framework using graph transformation and rewriting logic techniques. We illustrate our approach on a simple round trip time protocol.

1 Introduction

In model-driven engineering (MDE), metamodels provide modeling primitives to represent software artifacts as models, and model transformations are the core technique to support software evolution in an automated manner [19]. These techniques have been applied to the development of real-time and embedded systems (RTES), such as automotive, avionics, and medical systems. Such RTES are often hard to design correctly, since subtle timing aspects impact system functionality, yet are safety-critical systems where a bad design can result in the loss of revenue and human lives. Therefore, there is a clear need for automated formal analysis of RTES designs.

Some of the MDE approaches to RTES, such as MARTE [12], provide modeling languages based on UML profiles, others, such as AADL [18], on domain-specific modeling languages. To enable formal analysis of designs, these languages typically have to be mapped in an *ad hoc* way onto *external* formalisms that support automated reasoning [1].

In contrast, the approach taken in the MOMENT2 project [6,3] is to formalize MOF metamodels in rewriting logic, providing for free (i) a formal semantics of the *structural* aspects of any modeling language with a MOF metamodel, and

D.S. Rosenblum and G. Taentzer (Eds.): FASE 2010, LNCS 6013, pp. 29–43, 2010.

(ii) automated formal reasoning in the MOMENT2 tool, e.g., to analyze whether a given model conforms to its metamodel. To provide a generic formal framework for *dynamic* system aspect, MOMENT2 has been extended with in-place model transformations. In this framework, the static semantics of a system is given as a class diagram describing the set of valid system states, system states are represented as object diagrams, and the dynamics of a system is defined as an in-place model transformation where the application of a model transformation rule involves a state transition in the system. To the best of our knowledge, MOMENT2 is the first model transformation tool with both simulation and LTL model checking capabilities that is integrated into EMF.

This paper describes our extension of MOMENT2 to support the formal specification and analysis of *real-time* model transformations by providing:

- a simple and expressive set of constructs, defined in an EMF metamodel, for specifying real-time behaviors;
- a precise formal semantics of real-time model transformation systems as *real-time rewrite theories* [13];
- a methodology for formally simulating and model checking such systems using Maude as a hidden, back-end formal framework; and
- an approach for adding real-time features to model-based systems in a *non-intrusive* way, i.e., without modifying their metamodel.

We illustrate our techniques by specifying and analyzing a simple round trip time protocol. Although this protocol is a software system, we use metamodels for describing its structural semantics in order to show how our approach can be applied in the setting of an EMF modeling language.

Our work should be seen as a first step towards an automatic, executable formalization of systems defined with modeling languages with real-time features where a class diagram corresponds to a metamodel, an object diagram to a well-formed model and model transformations specify the behavior of the system. In this way, a software engineer can use MDA standards and Eclipse Modeling Framework (EMF) technology to formally define and analyze RTES.

This paper is organized as follows. Section 2 provides some background on rewriting logic and MOMENT2. Section 3 presents our approach for specifying real-time model transformations, as well as our example and the formal real-time rewrite semantics of such transformations. Section 4 shows how our transformations can be subjected to Maude-based formal analyses. Section 5 discusses related work and Section 6 gives some concluding remarks.

2 Preliminaries

2.1 Rewriting Logic, Maude, and Real-Time Maude

In rewriting logic [10], a concurrent system is specified as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ where:

- (Σ, E) is a *membership equational logic* (MEL) [11] theory where Σ is an algebraic signature[1], and E is a set of conditional *equations* $t = t'$ **if** *cond* and conditional *membership* axioms $t : s$ **if** *cond* stating that the term t has sort s when *cond* holds. (Σ, E) specifies the system's state space as an algebraic data type.
- R is a set of (possibly conditional) *rewrite rules* of the form $t \longrightarrow t'$ **if** *cond* that describe all the *local transitions* in the system; such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t', *provided* the condition holds.

Maude [7] is a high-performance implementation of rewriting logic that provides a set of formal analysis methods, including: *rewriting* for simulating *one* behavior of the system, *reachability analysis* for the verification of invariants, and *model checking* of linear temporal logic (LTL) properties. The Maude syntax is fairly intuitive. For example, a function symbol f is declared with the syntax `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its *sort*. Equations are written with syntax `eq` t `=` t', and `ceq` t `=` t' `if` *cond* for conditional equations, and rewrite rules are written with syntax `rl [`l`] :` t `=>` t' and `crl [`l`] :` t `=>` t' `if` *cond*. The mathematical variables in such statements are declared with the keywords `var` and `vars`. We refer to [7] for more details on the syntax of Maude.

We assume rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup A, R)$, where A is a set of axioms, so that both the equations E and the rules R are applied *modulo* the axioms A. That is, we rewrite not just terms t but rather A-equivalence classes $[t]_A$. The axioms A of associativity, commutativity, and identity of set union define *multisets*, and rewriting modulo these axioms corresponds to *multiset rewriting* that is directly supported in Maude. Rewriting multisets of objects linked by (possibly opposite) references exactly corresponds to graph rewriting, a correspondence that is systematically exploited in MOMENT2.

The Real-Time Maude language and tool [14] extend Maude to support the formal specification and analysis of *real-time rewrite theories*. The rewrite rules are divided into ordinary, *instantaneous* rewrite rules that are assumed to take zero time, and *tick rules* of the form `crl [`l`] : {`t`} => {`t'`} in time` u `if` *cond*, where u is a term (that may contain variables), denoting the *duration* of the rewrite, and `{_}` is a new operator that encloses the global state to ensure that time advances uniformly in all parts of the system. Real-Time Maude extends the Maude simulation, reachability, and LTL model checking features to timed systems, e.g., by providing time-bounded versions of these analysis methods, and by providing a set of *time-sampling strategies* to execute time-nondeterministic tick rules (see [14]).

2.2 MOMENT2: MOF, Models, and Model Transformations

MOMENT2 provides formal support for model-driven engineering by formalizing metamodels, defined using the MOF standard [15] and implemented in

[1] i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*.

EMF, as membership theories [6], and by formalizing model transformations as rewriting logic theories in Maude [5]. That is, MOMENT2 provides a simple well-known interface to software engineers, while providing a variety of formal analysis methods to analyze models and model transformations.

The algebraic semantics of MOF is defined as a "parametric" membership equational logic theory $\mathbb{A}$, so that for each MOF meta-model $\mathscr{M}$, we have a MEL specification $\mathbb{A}(\mathscr{M})$, and so that a model M conforms to the meta-model $\mathscr{M}$ iff its Maude representation is a term of sort *Model* in $\mathbb{A}(\mathscr{M})$. The representation of models as algebraic terms has the form `<< OC >>`, where `OC` is a multiset of objects of the form `< O : C | PS >`, with `O` an object identifier, `C` a class name, and `PS` a set of attributes and references between objects. This representation is automatically generated by MOMENT2 from models in the Eclipse Modeling Framework (EMF) [8]. A detailed definition of the mapping $\mathbb{A}$ is given in [4].

MOMENT2 provides support for developing, executing, and analyzing multi-model transformations, where several models might be involved. A pair $(\mathscr{M}, \mathscr{T})$, of a set of MOF metamodels $\mathscr{M}$ and a MOMENT2 model transformation definition $\mathscr{T}$, represents a model transformation, whose semantics is formally defined by a *rewrite theory* $\mathbb{R}(\mathscr{M}, \mathscr{T})$ that extends the MEL theory $\mathbb{A}(\mathscr{M})$.

A model transformation is defined as a set of production rules. Each such rule l of the form

`rl` *l* `{ nac` *dl nacl* `{` *NAC* `} such that` *cond* `;...`
`lhs {` *dl* `{` *L* `} }; rhs {` *dl* `{` *R* `} }; when` *cond*`;...}`

has a left-hand side L, a right-hand side R, a set of (possibly conditional) negative application conditions *NAC* and a condition with the *when* clause. L, R, and *NAC* contain model patterns, where nodes are object patterns and unidirectional edges are references between objects. For instance, in the pattern `A : Class1 { a = V, r = B : Class2 { .. },.. }` an object `A` of type `Class1` has an attribute `a`, whose value is bound to the variable `V`[2], and has a reference `r` that points to an object B of type `Class2`. Several models can be manipulated with a single production rule in MOMENT2. To identify which model should be matched by a given model pattern, we use the notion of *domain* that associates an identifier *dl* to an input model. See Section 3.2 for examples of model transformation rules.

The semantics of model transformations in MOMENT2 is based on the *single-pushout approach* (SPO) for graph transformations [17]. A production rule is applied to a model when a match is found for the patterns in L, a match is *not* found for the patterns in *NAC*s, and the *when* clause holds. When a rule is applied, objects and references in $L \setminus R$ are removed, objects and references in $R \setminus L$ are created and objects and references in $L \cap R$ are preserved. According to the SPO semantics, all dangling edges are removed. In this paper we focus on *in-place* (multi-)model transformations, which use the same (multi-)model both as input and as output of the transformation. Such models usually represent *system states*, and the application of production rules correspond to *state transitions*.

[2] Variables are declared without types. The type of the variable is inferred from the information in the metamodel.

3 Real-Time Model Transformations in MOMENT2

This section shows how *timed* behaviors can be added to behavioral specifications in MOMENT2. *Ecore* is the modeling language used in EMF to define metamodels, which we use to specify the static view of a system. We provide a collection of built-in types for defining *clocks*, *timers*, and *timed values* so that: (i) the Ecore model $\mathscr{M}$ of an RTES can be extended with such types, and (ii) a system state can contain clocks, timers, and timed values.

In our approach, a *timed behavioral specification* of a system is given as a triple $(\mathscr{M}, \mathscr{T}, \delta)$, where $\mathscr{M}$ is the structural specification of the system, given as an Ecore model extended with our built-in timed constructs, $\mathscr{T}$ is an in-place model transformation in MOMENT2 defining the dynamics of the system, and δ is the time sampling strategy used to decide whether each moment in a discrete time domain is visited, or only those moments in time when a timer expires.

Section 3.1 introduces the built-in timed constructs and their metamodel. Section 3.2 illustrates the use of these constructs on a small and simplified, but prototypical, real-time protocol for finding the message exchange round trip time between two (in our case neighboring) nodes in a network. Section 3.3 discusses how we can define real-time behaviors without having to modify the (possibly "untimed") metamodel of a system. Section 3.4 defines the formal real-time rewrite semantics of our model transformations as a mapping

$$\mathbb{R}_T : (\mathscr{M}, \mathscr{T}, \delta) \ \mapsto \ (\Sigma_T, E_T, R_T),$$

taking a timed specification $(\mathscr{M}, \mathscr{T}, \delta)$ to a real-time rewrite theory (Σ_T, E_T, R_T).

3.1 Constructs for Defining Real-Time Model Transformations

The metamodel for the basic built-in constructs provided by MOMENT2 to support the specification of real-time model transformations is given in Fig. 1. We below give an intuitive explanation of these constructs, whose formal semantics is given in Section 3.4, and their use. Timed constructs specialize the `TimedConstruct` class that has a reference to the `EObject` class from the Ecore metamodel. This reference allows our timed constructs to point to any construct in an EMF model so that time features can be added to system states in a non-intrusive way as explained in Section 3.3.

Timer. A `Timer` whose `on` attribute is `true` *decreases* its `value` according to the elapsed time. When the `value` reaches 0, time advance is blocked, forcing the use of a model transformation rule *which also modifies the timer* by either turning off the `Timer` (that is, the `on` attribute is set to `false`), or by resetting the `value` attribute to the time until the timer should expire the next time. In this way, a timer is used to force an action to happen at (or before) a certain time. The `value` of a `Timer` whose `on` attribute is `false` does *not* change when time advances; neither can such a turned off `Timer` block time advance.

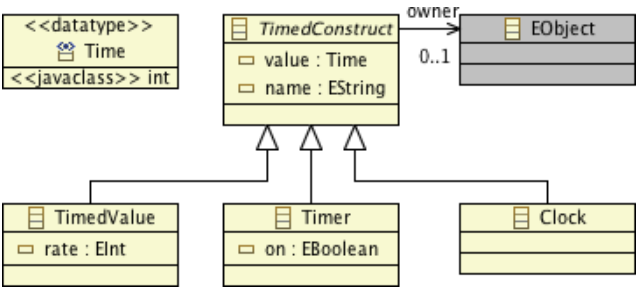

Fig. 1. Ecore metamodel of the predefined timed constructs

Clock. The `value` of a `Clock` is increased according to the elapsed time. A `Clock` with initial `value` 0, and whose value is not changed by a model transformation rule, therefore always denotes the current "model time."

Timed Value. The `TimedValue` construct is similar to the `Clock` construct. The difference is that, whereas the `value` of a `Clock` is increased by the amount of elapsed time, the `value` of a `TimedValue` object is increased by *the elapsed time multiplied with the* `rate`, which may be a negative number.

3.2 Example: A Round Trip Time Protocol

For an example that uses both clocks and timers, consider a very simple protocol for finding the *round trip time* between two neighboring nodes in a network; that is, the time it takes for a message to travel from source to destination, and back.

The initiator starts a round of this protocol by sending a *request* message to the other node and recording the time at which it sent the request. When the responder receives the request message, it immediately sends back a *reply* message. When the initiator receives the reply message, it can easily compute the round trip time using its local clock. Since the network load may change, and messages may get lost, the initiator starts a new round of the protocol *every* 50 time units. We assume that the message transmission time is between 2 and 8 time units; in addition, any message could be lost for some reason.

Figure 2 shows the structural model for this example as a class diagram, defined as an Ecore model in EMF. The attribute `rtt` of a `Node` denotes the latest computed round trip time value; `lastSentTime` denotes the time that the last request message was sent; `roundTimer` points to the timer upon whose expiration the node starts another round of the RTT protocol; and the `Clock` denotes the local clock of the node.

To model the fact that the transmission delay of a message is between 2 and 8 time units, each message has an associated clock (to avoid that the message is read too early) and a timer (to ensure that the message is not read too late).

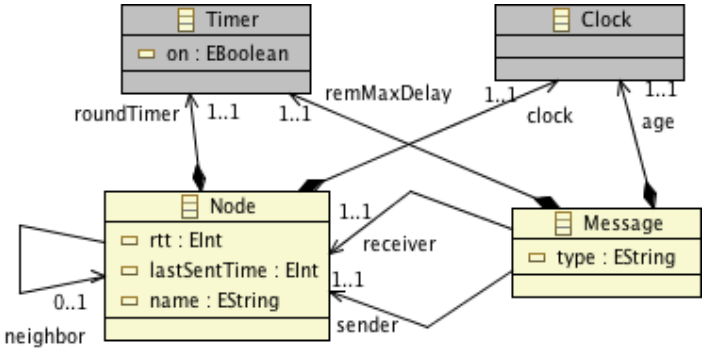

Fig. 2. Class diagram for the RTT example

The following rule models the transformation when an active round timer of a node `A` expires (that is, equals 0). As seen in the right-hand side of the rule, the node then sets the value of `lastSentTime` to the current time (as given by its local clock), resets its timer to expire in 50 time units, and generates a `request` message. The generated message sets its `age` clock to 0, and sets its timer to 8, ensuring that the message is read (or lost) within 8 time units[3]:

```
rl sendRequest {
   lhs { model {
      A : Node {
         clock = C : Clock { value = TIME },
         neighbor = B : Node { },
         roundTimer = RT : Timer { value = 0, on = true }  }
   }};
   rhs { model {
      A : Node {
         clock = C : Clock { value = TIME },
         neighbor = B : Node { },
         roundTimer = RT : Timer { value = 50, on = true },
         lastSentTime = TIME  }
      M : Message {
         age = MA : Clock { value = 0 },
         sender = A : Node {},
         receiver = B : Node { },
         remMaxDelay = RMD : Timer { value = 8, on = true },
         type = "request" }
   }};  }
```

The next rule models the reception (and consumption) of a `request` message. Since message transmission takes at least 2 time units, this can only happen when the `age` clock of the message is greater than or equal to 2. As a result of applying this rule, a `reply` message is created, and sent back to the node `A`:

[3] Variable names are capitalized in our model transformation rules.

```
rl replyRequest {
   lhs { model {
      B : Node { }
      M : Message {
         age = MA : Clock { value = MSGAGE },
         sender = A : Node { },
         receiver = B : Node { },
         type = "request" }
   } };
   rhs { model {
      B : Node { }
      NEW-MSG : Message {
         age = MA2 : Clock { value = 0 },
         sender = B : Node { },
         receiver = A : Node { },
         remMaxDelay = RMD : Timer { value = 8, on = true },
         type = "reply" }
   } };
   when MSGAGE >= 2; }
```

The following rule models the reception of the `reply` message. The receiver `A` defines the round trip time `rtt` to be the difference between the current time (as given by its local clock) and the value of `lastSentTime`:

```
rl getReplyAndComputeRtt {
   lhs { model {
      A : Node {
         lastSentTime = LASTTIME,
         clock = C : Clock { value = TIME } }
      M : Message {
         age = MA : Clock { value = MSGAGE },
         receiver = A : Node { },
         type = "reply" }
   } };
   rhs { model {
      A : Node {
         lastSentTime = LASTTIME,
         clock = C : Clock { value = TIME },
         rtt = TIME - LASTTIME }
   } };
   when MSGAGE >= 2; }
```

The last rule models the possibility of a message being lost in transition:

```
rl messageLoss {
   lhs { model { M : Message { }  } };
   rhs { model { } }; }
```

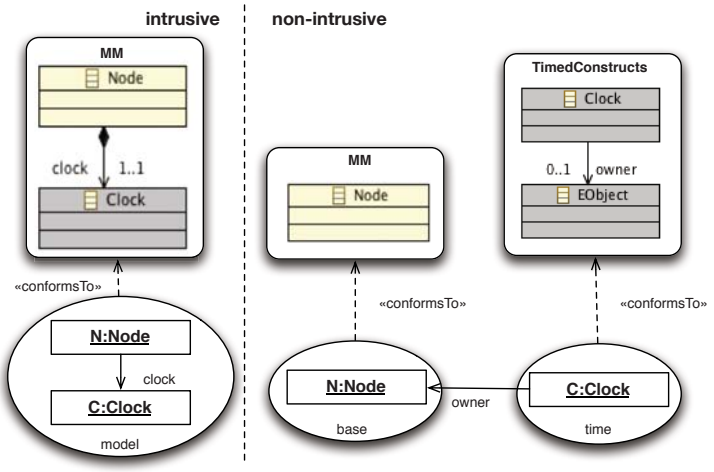

Fig. 3. Intrusive and non-intrusive approaches for adding time features

3.3 Non-intrusive Model-Based Real-Time Behavioral Specification

The presented approach might not seem the most suitable to specify a system that has been deployed, since it requires changing the structural design of the application, so that timer and clock objects can be defined in the state. We refer to this approach as *intrusive*. We therefore also provide a *non-intrusive* approach, in which the *user-defined* metamodel of the system is not modified. These two approaches are illustrated in Fig. 3 and explained below.

In the *intrusive* approach, the user-defined metamodel imports the built-in timed constructs. In this case, the extension of the system with time features is performed internally in the system by modifying the classes that constitute the static view of the system. For example, in the metamodel each node can have a clock. A state of such a system is a self-contained model that may contain time features. The behavioral specification can then be defined as an in-place model transformation with one single domain.

In the *non-intrusive* approach, the user-defined metamodel is not modified and remains agnostic from timed constructs. The extension of a system with time features is performed at the model level by defining an external model *time* that contains timed constructs. These timed constructs are related to objects in a model by means of `owner` external references. The timed behavioral specification of the system is defined with a *multi-model* transformation with two domains, one for the model and one for the time model that extends the initial model. Therefore, a system state consists of two separate model: *base* and *time*.

We next present a non-intrusive version of our round trip time protocol. Since the base system is not aware of time features, two steps are needed in order to attach real-time features to the objects: *(i)* a new independent model with timers and clocks extends the base model that corresponds to the initial state of the system; and *(ii)* the behavioral specification of the system is given as an

in-place *multi-model* transformation, where both the base model and the time model constitute the system state.

In the initial state of the RTT protocol, a clock and a timer are associated to the sender. We present the `getReplyAndComputeRtt` rule in the model transformation. The left-hand side pattern consists of two domains: in the `base` domain the pattern matches a reply message `M` that is sent to a node `A`, and in the `time` domain the pattern matches a clock `C` that is *owned by* the node `A`, a clock `MA` that is *owned by* the message `M`, the round trip time value of the node `A` and the last time a message was sent by `A`. The values for the round trip time and the last time a message was sent are stored in external objects in the `time` domain. In the right-hand side of the rule, the message `M` is removed from the `base` domain and the new round trip time value is computed in the `time` domain:

```
rl getReplyAndComputeRtt {
   lhs {
     base { M : Message { receiver = A : Node { }, type = "reply" }}
     time { C : Clock { value = TIME, owner = A : Node{}}
       RTT : TimedConstruct { value = RTT, owner = A : Node{}}
       LST : TimedConstruct { value = LASTTIME, owner = A : Node{}}
       MA : Clock { value = MSGAGE, owner = M : Message { }}}};
   rhs {
     base { A : Node { } }
     time { C : Clock { value = TIME, owner = A : Node{}}
       RTT : TimedConstruct { value = TIME - LASTTIME, owner = A : Node{}}
       LST : TimedConstruct { value = LASTTIME, owner = A : Node{}}}};
   when MSGAGE >= 2 ; }
```

3.4 Formal Semantics of Real-Time Model Transformations

This section presents the formal *real-time rewriting logic* [13] semantics of real-time model transformations in MOMENT2. As mentioned, our semantics extends the (untimed) rewriting logic semantics of model transformations given in [5]. In particular, all data types and rewrite rules defining the semantics of model transformations are inherited. Those rewrite rules are now considered to be *instantaneous* rewrite rules modeling *instantaneous change*. The real-time rewrite semantics adds the single tick rule

$$\texttt{crl [tick] : \{<<}\ OC\ \texttt{>>\} => \{<<}\ \texttt{delta}(OC, X)\ \texttt{>>\}}\ \texttt{in time}\ X\ \texttt{if}\ X\ \texttt{<=}\ \texttt{mte}(OC)$$

where OC is a variable of the sort `ObjCol` denoting multisets of objects representing instances of classes in the rewrite semantics of model transformations, and X is a variable of the sort $Time$ denoting the time domain.

Following the guidelines given in [14] for defining object-oriented real-time systems, the function `delta` defines the effect of time elapse on a system, and the function `mte` defines the *m*aximum *t*ime *e*lapse possible in a system before some action must be taken. These functions are defined as follows:

```
op delta : ObjCol Time -> ObjCol .

var OC : ObjCol .  var O : Oid .  var G : Int .  var PS : PropertySet .
vars T T' : Time .  var vTimer : Timer .  var vTDV : TimedValue .  var vClock : Clock .

eq delta(< O : vTimer | property : "value" = T', property : "on" = true, PS > OC, T) =
         < O : vTimer | property : "value" = (T' monus T) , property : "on" = true, PS >
         delta(OC, T) .
```

The above equation defines the effect of the elapse of time `T` on a collection of objects consisting of a `Timer` object, whose `on` attribute is `true`, and other objects (captured by the variable `OC`). The effect of the time elapse is to decrease the `value` of the timer by `T`,[4] and to recursively apply the function `delta` to the remaining objects `OC`.

Likewise, the effect of the elapse of time `T` on a `Clock` object is to increase its `value` by `T`, and the effect of that time elapse on a `TimedValue` object with rate `G` is to increase the `value` by `G * T`:

```
eq delta(< O : vClock | property : "value" = T', PS >  OC, T) =
         < O : vClock | property : "value" = (T' + T) , PS >  delta(OC, T) .

eq delta(< O : vTDV | property : "value" = T', property : "rate" = G , PS > OC, T) =
         < O : vTDV | property : "value" = (T' + (G * T)) , property : "rate" = G, PS >
         delta(OC, T) .
```

The following equation matches "otherwise" (`owise`), that is, when none of the above equations can be applied. In those cases, time elapse has no effect on a collection of objects. In particular, time does not effect user-defined objects:

```
eq delta(OC, T) = OC [owise] .
```

The function `mte` should ensure that time does not advance beyond the expiration time of an active timer. Therefore, `mte` of a collection of objects returns the smallest `value` of the active `Timer`s in the state (`INF` denotes "infinity"):

```
op mte : ObjCol -> TimeInf .
eq mte(< O : vTimer | property : "value" = T , property : "on" = true , PS > OC) =
       minimum(T , mte(OC)) .
eq mte(OC) = INF [owise] .
```

Although timed MOMENT2 model transformations have a real-time rewriting semantics, the current implementation of MOMENT2 is not based on the Real-Time Maude tool. Nevertheless, Real-Time Maude features such as, e.g., selecting the time sampling strategy with which to execute tick rules, and performing *time-bounded* reachability and LTL model checking analyses, *are* available in MOMENT2. For example, the tool allows the user to choose between advancing time by *one time unit* in each application of the tick rule, or advancing time until a timer expires.

[4] The function `monus` is defined by x `monus` $y = \max(0, x - y)$.

4 Formal Analysis

The MOMENT2 tool provides a spectrum of formal analysis methods, in which a model transformation system can be subjected to Maude analyses such as simulation, reachability analysis, and linear temporal logic (LTL) model checking. Simulation explores *one* sequence of model transformations from an initial model. Reachability analysis analyzes *all* possible model transformation sequences from an initial model to check whether some model *pattern* can be reached, and LTL model checking analyzes whether all such sequences satisfy a given LTL formula.

Since the MOMENT2 implementation does not target Real-Time Maude, the above analysis methods are all *untimed*. Nevertheless, we can also easily perform *time-bounded* analyses by just adding a single unconnected `Timer`—whose initial value is the time bound—to the initial state. When this timer expires, time will not advance further in the system, since no rule resets or turns off the timer. The ability to perform time-bounded analysis is not only useful *per se*, but also makes (time-bounded) LTL model checking analysis possible for systems with an infinite reachable state space, such as our RTT example, that can otherwise not be subjected to LTL model checking.

4.1 Formal Analysis of the RTT Protocol

In MOMENT2, the model instance that corresponds to the initial state is used as input model for model transformations. To analyze our round trip time protocol specification, we have added a timer set to 500 to the initial model instance. The reachable state space is therefore restricted to those states that are reachable within 500 time units. Without such a restriction, the reachable state space would be infinite, since the clock values can grow beyond any bound.

We use MOMENT2's search command to verify the safety property that the recorded `rtt` value of a node is either 0 (reply message not yet received) or is some value in the interval $[4, 16]$. This property can be verified (for behaviors up to 500 time units) by searching for a reachable state where the recorded `rtt` value is *not* within the desired set of values; that is, by searching for a node `N : Node { rtt = RTT }` whose round trip time value is `RTT <> 0 and (RTT < 4 or RTT > 16)`. We search for one counterexample, and without any bound on the depth of the search tree (`[1,unbounded]`):

```
search [1,unbounded] =>*
           domain model { N : Node { rtt = RTT } }
               such that RTT <> 0 and (RTT < 4 or RTT > 16)
```

We use time-bounded LTL model checking to verify the stability property that once an `rtt` value in the desired range $[4, 16]$ has been recorded, the value of the `rtt` attribute stays in this interval. We first define a parametric state proposition `okRTTValue`, so that `okRTTValue(`n`)` holds in states where the recorded round trip time value of a node n is in the interval $[4, 16]$:

```
domain model { N : Node { rtt = RTT, name = ID } } |= okRTTValue(ID)
               = (RTT >= 4 and RTT <= 16) ;
```

We can then model check the following LTL formula to verify our desired stability property for node `"A"`:

```
[] (okRTTValue("A") -> [] (okRTTValue("A")))
```

5 Related Work

A prominent early work in the closely related field of timed *graph* transformations is the work by Gyapay, Varró, and Heckel in [9], where the authors define a model of timed graph transformations based on a timed version of Petri nets. In their model (the "GVH model"), each object has an associated time stamp, called *chronos*, denoting the last time the object participated in a transition. All transitions are *eager* and have durations.[5] This is quite different from our model, where transitions are non-eager and instantaneous, and where timeliness of desired actions is achieved by using timers. Another difference is that in the Petri-net-based semantics of the GVH model, executions may be *time-inconsistent* in that a transition firing at time t_1 may take place *after* a transition firing at a later time $t_2 > t_1$. However, in [9] it is shown that such time-inconsistent executions can be rearranged to "equivalent" time-consistent executions. Our real-time model transformations have a real-time rewrite theory semantics, where all computations are time-consistent [13]. Comparing the "timed" expressiveness of the two approaches is nontrivial. On the on hand, each GVH-execution cannot be "directly" simulated in our framework, since we cannot have time-inconsistent computations in real-time rewrite theories. Likewise, we do not know whether there exists a semantics-preserving encoding of our models as GVH-models.

Becker and Giese use a timed extension of graph transformation systems for verifying RTES [2], where clocks are used as attributes in graphs. Their work focuses on inductively verifying safety properties over all possible states in a system, as opposed to *automatically* verifying the system from a given initial state as in model checking approaches like ours, where more expressive linear temporal logic properties can be verified.

A recent paper by Rivera, Vicente-Chicote, and Vallecillo [16] also advocates the use of in-place model transformations to complement metamodels with timed behavioral specifications. However, the time models are completely different. In their approach, they do not add any explicit constructs for defining time behavior (so as not to modify metamodels); instead the transformation rules have time intervals denoting the duration interval of each local action. In our approach, timed constructs are included in the system state but their semantics is encoded in MOMENT2, providing a simpler setting in which time does not have to be explicitly manipulated in the behavioral specification of the system. Despite this simplicity, we have shown how our approach allows us to consider timed requirements in a non-intrusive way through multi-model transformations.

[5] Alternatively, one may view transitions as instantaneous transitions that apply at time Δ after becoming enabled.

In [20], Syriani and Vangheluwe model the PacMan as graph transformations extended with time. The timed behavior of their model is that the system remains in a state as long as specified by *time-advance* of that state, or until some input is received in some port. The system then performs a transition. This time behavior falls within our timed model, where *timed-advance* is simply modeled as a timer. The paper [20] presents no formal model of their framework, and the only form of analysis is simulation using a tool implemented in Python.

Using meta-models for Giotto and the E-Machine in the GME toolkit, Szemethy used the GReAT *untimed* graph rewrite system to transform time-triggered Giotto source code into schedule-carrying E-Machine code [21].

Several approaches use a model-driven development methodology for modeling real-time systems. For instance, AADL [18] and MARTE [12] are used for safety-critical avionics and automotive systems. These approaches only consider the specification of RTES, and formal analysis is provided by translating the model of a system into an external formalism with verification capabilities. Our work provides an automatic, direct mechanism to formalize and analyze model-based RTES by means of in-place model transformations in MOMENT2. MOMENT2 encodes the metamodels and model transformation into rewriting logic and leverages Maude verification techniques to analyze model transformations.

6 Concluding Remarks

This paper has shown how the formal model transformation specification and analysis tool MOMENT2 has been extended to real-time model transformations by providing a few simple and intuitive constructs for describing timed behavior. We have also shown how the multi-modeling capabilities of MOMENT2 can be exploited to define timed behaviors in a *non-intrusive* way; i.e., without changing the given metamodel to specify real-time behaviors. We have given a real-time rewrite semantics to real-time model transformation systems, and have shown how such systems can be subjected to (unbounded and time-bounded) reachability and LTL model checking analyses. We believe that our timed model is easy to understand and is suitable to define advanced real-time model transformations, as indicated by our example. In addition, and in contrast to most competing approaches, both simulation and LTL model checking are integrated into our tool. Furthermore, through MOMENT2, our methods can be automatically applied to EMF-based systems and modeling languages so that rewriting logic techniques are made available in mainstream model-driven development processes for analysis purposes.

Acknowledgments. We gratefully acknowledge financial support by The Research Council of Norway and by the EU project SENSORIA IST-2005-016004.

References

1. André, C., Mallet, F., de Simone, R.: Modeling time(s). In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 559–573. Springer, Heidelberg (2007)
2. Becker, B., Giese, H.: On Safe Service-Oriented Real-Time Coordination for Autonomous Vehicles. In: ISORC 2008. IEEE, Los Alamitos (2008)
3. Boronat, A., Meseguer, J.: Algebraic semantics of OCL-constrained metamodel specifications. In: TOOLS-EUROPE 2009. LNBIP, vol. 33. Springer, Heidelberg (2009)
4. Boronat, A.: MOMENT: a formal framework for MOdel manageMENT. PhD in Computer Science, Universitat Politènica de València (UPV), Spain (2007), http://www.cs.le.ac.uk/people/aboronat/papers/2007_thesis_ArturBoronat.pdf
5. Boronat, A., Heckel, R., Meseguer, J.: Rewriting Logic Semantics and Verification of Model Transformations. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 18–33. Springer, Heidelberg (2009)
6. Boronat, A., Meseguer, J.: An Algebraic Semantics for MOF. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 377–391. Springer, Heidelberg (2008)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
8. Eclipse Organization: The Eclipse Modeling Framework (2007), http://www.eclipse.org/emf/
9. Gyapay, S., Varró, D., Heckel, R.: Graph transformation with time. Fundam. Inform. 58(1), 1–22 (2003)
10. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
11. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
12. Object Management Group: The official OMG MARTE Web site (2009), http://www.omgmarte.org/
13. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoretical Computer Science 285, 359–405 (2002)
14. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
15. OMG: Meta Object Facility (MOF) 2.0 Core Specification, ptc/06-01-01 (2006)
16. Rivera, J., Vicente-Chicote, C., Vallecillo, A.: Extending visual modeling languages with timed behavioral specifications. In: XII Iberoamerican Conference on Requirements Engineering and Software Environments, IDEAS 2009 (2009)
17. Rozenberg, G.: Handbook of Grammars and Computing by Graph Transformation, vol. 1. World Scientific Publishing Company, Singapore (1997)
18. SAE: AADL (2007), http://www.aadl.info/
19. Schmidt, D.C.: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
20. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with time for simulation-based design. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 91–106. Springer, Heidelberg (2008)
21. Szemethy, T.: Case study: Model transformations for time-triggered languages. ENTCS 152, 175–190 (2006)