Electronic Notes in Theoretical Computer Science 4 (1996)

Specifying Real-Time Systems in Rewriting Logic

Peter Csaba Ölveczky [a,b,1] José Meseguer [a,2]

[a] *Computer Science Laboratory*
SRI International
Menlo Park, U. S. A.

[b] *Department of Informatics*
University of Bergen
Bergen, Norway

Abstract

This work investigates the suitability of rewriting logic as a semantic framework for modeling real-time and hybrid systems. We present a general method to specify and symbolically simulate such systems in rewriting logic and illustrate it with a well-known benchmark. We also show how a wide range of real-time and hybrid system models can be naturally expressed and are unified within our approach. The relationships with timed rewriting logic [9,10] are also investigated.

1 Introduction

Rewriting logic is a flexible and expressive framework in which many different models of concurrent computation and many different types of systems can be naturally specified [13,16,12,14]. It seems therefore natural to investigate the question of how rewriting logic can be applied to the specification of real-time and hybrid systems. From the semantic point of view this offers the possibility of integrating real-time aspects with other features and models already supported by rewriting logic.

The first important research contribution exploring the application of rewriting logic to real-time specification has been the work of Kosiuczenko and Wirsing on *timed rewriting logic* (TRL) [9], an extension of rewriting logic where the rewrite relation is labeled with time stamps. TRL has been shown well-suited for giving object-oriented specifications of complex hybrid systems such

[1] Supported by the Norwegian Research Council.
[2] Supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, National Science Foundation Grant CCR-9224005, and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

as the steam-boiler [18]. In fact, rewriting logic object-oriented specifications in the Maude language [16] have a natural extension to TRL object-oriented specifications in *Timed Maude* [9,18].

The approach taken here is different. We do not extend rewriting logic at all. We instead investigate the question of how naturally, and with what degree of generality, can real-time systems be formally specified in standard rewriting logic. Our findings indicate that real-time and hybrid systems can in fact be specified quite naturally in rewriting logic, and that a wide range of frequently-used models of real-time and hybrid systems can be regarded as special instances of our theoretical approach.

In essence our approach can be summarized as follows. A time domain—satisfying quite general axioms, so as to allow both discrete and continuous, as well as linear and nonlinear time models—is an explicit parameter of the specification. In addition, the passage of time is viewed as a monoid action, acting on the states, whose result on different components of the system is specified by rewrite rules. The system can then *react* to such time actions and to stimuli from its environment, by performing control actions that are also specified by rewrite rules. In some cases such reaction rules must have an eager strategy, to ensure that real-time requirements are met.

In Section 2 we explain the general method for specifying real-time systems sketched above. We then illustrate in Section 3 the naturalness of the method, and its smooth integration with rewriting logic's support for object-oriented specification, by means of the well-known train intersection controller benchmark. The question of how generally rewriting logic can be used to express other real-time and hybrid system models is addressed in Section 4. We show in detail how a wide range of such models, including timed automata [3], hybrid automata [2], timed and phase transition systems [11], and timed extensions of Petri nets [1,17], can indeed be expressed in rewriting logic quite naturally and directly.

In Section 5 we study the relationships between our approach and TRL. We show that there is a map of entailment systems $\mathcal{M} : TRL \longrightarrow RWL$ sending each TRL specification to a corresponding specification in such a way that logical entailment is preserved. However, the translated theory $\mathcal{M}(\mathcal{T})$ can in general prove additional sentences. This is due to some intrinsic conceptual differences between both formalisms that our analysis reveals. However, for the cases of TRL theories $\mathcal{T}$ where $\mathcal{M}(\mathcal{T})$ exactly mirrors the deductions of $\mathcal{T}$ we provide a general method not only for performing corresponding deductions, but also for simulating through execution in rewriting logic the behavior of the system specified by $\mathcal{T}$.

Indeed, *symbolic simulation* of a real-time system's formal specification is one of the attractive features of our general approach. Perhaps a good way to see how rewriting logic specifications complement more abstract specifications such as temporal logic as well as more concrete, automaton-based ones, is to think of them as providing an *intermediate level*, that can substantially help in bridging the gap between specification and implementation by providing:

- a precise mathematical model of the system (the initial model [13]) against

which more abstract specifications can be proved correct;

- support for useful kinds of automated or semi-automated reasoning about the system at the rewriting logic and equational logic levels;
- support for executable specification and symbolic simulation;
- good system compositionality capabilities, through parameterization, module hierarchies, and object-oriented features.

Much more should be done to further investigate and exploit these possibilities. The formal tools already available, and those planned for the near future, will greatly help us and others in this regard.

2 Specifying Real-Time Systems in Rewriting Logic

This section introduces the rewriting logic techniques we use for specifying and reasoning about two different aspects of real-time systems. Section 2.1 gives abstract specifications of time. In Section 2.2 the idea of time as an action, acting on a system so as to change its state, is introduced, obtaining a framework where properties of the form "t rewrites to t' in time r" can be proved. In Section 2.3 these ideas are extended and used to simulate the actual behavior of systems. In this second case, we are interested in whether a term t, representing a given state, rewrites to a term t' in arbitrary time. The evolution in time of a system can then be observed by following the rewrite sequence starting with the initial state t. In Section 2.4 some rewriting strategies that should sometimes be included in a specification are briefly outlined.

Notation: We will use the symbols $r, r', r_1, \ldots$ to denote time *values*, $x_r, y_r, \ldots$ to denote *variables* of sort *Time*, and $t_r, t'_r, \ldots$ to denote *terms* of sort *Time*.

2.1 Time Models

Time is modeled abstractly by a commutative monoid $(Time, +, 0)$ with additional operators $\leq$, $<$, and $\dot{-}$ ("monus") satisfying the following Maude theory.

fth *TIME* **is**
protecting *BOOL*
  sort *Time*
  op $0 : \rightarrow Time$
  op $\_ + \_ : Time, Time \ \ [assoc\ \ comm\ \ id : 0]$
  ops $\_ < \_, \_ \leq \_ : Time, Time \rightarrow Bool$
  op $\_ \dot{-} \_ : Time, Time \rightarrow Time$
  vars $x_r, y_r, z_r, w_r : Time$
  ceq $x_r = 0$ **if** $(x_r + y_r) == 0$
  ceq $y_r = z_r$ **if** $x_r + y_r == x_r + z_r$
  eq $(x_r + y_r) \dot{-} y_r = x_r$
  ceq $x_r \dot{-} y_r = 0$ **if** $not(y_r \leq x_r)$

eq $x_r \leq x_r + y_r = true$
ceq $(x_r \leq y_r) = true$ **if** $(x_r < y_r)$
eq $(x_r < x_r) = false$
eq $(x_r \leq y_r) = (x_r < y_r)$ or $(x_r == y_r)$
ceq $x_r + y_r \leq z_r + w_r$ **if** $x_r \leq z_r$ and $y_r \leq w_r$
ceq $(x_r \dot{-} y_r) + y_r = x_r$ **if** $y_r \leq x_r$
endft

In this theory, it can be proved that the relation $\_ \leq \_$ is a partial order, that for all $x_r, y_r : Time$, $0 \leq x_r = true$, and that $y_r \leq x_r$ if and only if there exists a unique z_r (namely $x_r \dot{-} y_r$) such that $x_r = y_r + z_r$.

For simulation and executable specification purposes we will be interested in *computable* models of the above theory *TIME*. This means that all the operations are computable. By the Bergstra-Tucker Theorem [5], such models are finitely specifiable as initial algebras for a set E of Church-Rosser and terminating equations. For example, the nonnegative rational numbers can be so specified as a model of *TIME* by adding a subsort Rat_+ to the specification of rationals in [7], and extending it with an order and a monus operation in the obvious way. Similarly, the real algebraic numbers with the standard order are also computable [19], and therefore, have a finite algebraic specification with Church-Rosser and terminating equations. Note that just taking a constructive version of the real numbers will not yield a computable data type, because the equality and order predicates on the constructive reals are not computable [4].

In many cases, an additional time value ∞ is needed.

fth $TIME_\infty$ **is**
extending *TIME*
sort $Time_\infty$
subsort $Time \leq Time_\infty$
op $\infty : \rightarrow Time_\infty$
op $\_ \leq \_ : Time_\infty, Time_\infty \rightarrow Bool$
op $\_ \dot{-} \_ : Time_\infty, Time_\infty \rightarrow Time_\infty$
op $\_ + \_ : Time_\infty, Time_\infty \rightarrow Time_\infty$ $[assoc\ \ comm\ \ id : 0]$
var $x_r : Time$
eq $x_r \leq \infty = true$
endft

In case time is assumed to be linear, such as in the railroad crossing example in Section 3, time can be specified by the following theory:

fth *LTIME* **is**
extending *TIME*
op $\min : Time, Time \rightarrow Time$ $[comm]$
vars $x_r, y_r : Time$
ceq $x_r = y_r$ **if** $not(x_r < y_r)$ and $not(y_r < x_r)$
ceq $\min(x_r, y_r) = y_r$ **if** $y_r \leq x_r$
endft

This theory can also be extended with an ∞ time value as follows:

fth $LTIME_\infty$ **is**
extending $LTIME$, $TIME_\infty$
  op $\min : Time_\infty, Time_\infty \to Time_\infty \ [comm]$
  var $x_r : Time_\infty$
  eq $\min(\infty, x_r) = x_r$
endft

2.2 Time as an Action

Our proposal is to view the passage of time as an *action* that has an effect on each state of the system. As we have just explained, time is modeled as a commutative monoid $(Time, +, 0)$ with additional structure. Therefore, the action in question is axiomatized as a monoid action δ satisfying the usual axioms:

$$\delta(y, 0) = y$$
$$\delta(y, x_r + x'_r) = \delta(\delta(y, x_r), x'_r)$$

where y is a variable of any sort corresponding to the system's state.

In addition, other rewrite rules describe how a state on which time acts is transformed into an ordinary state. Intuitively, the meaning of δ is that for $t, t' \in T_{\Sigma - \{\delta\}}$, the sequent

$$\delta(t, r) \longrightarrow t'$$

is a valid rewrite deduction in the theory iff it is the case that whenever time has acted on t for r time units, it could rewrite to t'. The following simple example shows how rewriting logic can thus be used to deduce temporal properties of a system.

Example 2.1 Assume that time is modeled by the natural numbers, that if time acts on a term a for two time units, a can rewrite to b, which can then rewrite to c if time has acted for time 0, and, finally, c can rewrite to d in two time units. This system can be modeled by the rewriting logic theory with constants a, b, c, d of sort *State*, an operation $\delta : State, Nat \to State$, the above monoid action equations, and the set of rules

$$\{\delta(a, 2) \longrightarrow b,\ b \longrightarrow c,\ \delta(c, 2) \longrightarrow d\}.$$

It is easy to deduce that $\delta(a, 4) \longrightarrow d$, which means that when time has acted on a for 4 time units, it could rewrite to d.

2.3 Simulation and "Ticks"

Although the time monoid action itself is in some ways sufficient to reason about time change, in many cases we are interested in *simulating* in rewriting logic the behavior of a real-time system in terms of the ordinary states of which it is made up. Therefore, instead of just proving time elapse properties of the form $\delta(t, r) \longrightarrow t'$, we wish to start with a term t representing a given

state and then simulate the system, i.e., observe its evolution in time in the form of sequences of rewrites

$$t \longrightarrow t' \longrightarrow t'' \longrightarrow \cdots .$$

Some care must be taken in this case, since the understanding in the previous section was that, for t, t' not involving the δ operator, $t \longrightarrow t'$ is supposed to mean that t rewrites to t' in zero time ($t = \delta(t, 0) \longrightarrow t'$). We need to carefully change the intended meaning of the rewrite relation to let $t \longrightarrow t'$ denote, under appropriate assumptions, that t could rewrite to t' in arbitrary time.

Another problem stems from the fact that a simulation of this kind clearly requires "tick" rewrites, which model the elapse of time in a system. Using arbitrary rules of the form

$$t \longrightarrow \delta(t, t_r)$$

presents the risk of allowing rewrites such as $f(t, t') \longrightarrow f(\delta(t, t_r), t')$, i.e., rewrites in which time elapses only in a part of the system under consideration.

The solution to both of these problems is to assume that the states of the system are in a sort *State*, and then introduce a new sort *GlobalState* and a new symbol $\widehat{\ } : \mathit{State} \rightarrow \mathit{GlobalState}$ whose intended meaning is that the state t of the *whole* system under consideration is denoted by $\widehat{t}$. Therefore, $\widehat{t} \longrightarrow \widehat{t'}$ means that the whole system in state t rewrites to the state t' in some time, while

$$\delta(t, r) \longrightarrow t' \text{ and } t \longrightarrow t'$$

still mean that (the part) t (of a system) rewrites to t' in time r, and in no time, respectively.

Tick rules modeling the elapse of time therefore have the operator $\widehat{\ }$ at the top, to ensure uniform time elapse. Furthermore, since time may not elapse for certain deadlock states, the tick rules are in general conditional, and hence of the form

$$\widehat{t} \longrightarrow \widehat{\delta(t, t_r)} \textbf{ if } C.$$

Hence, a real-time system specified by a rewrite theory $\mathcal{R}$ with rules of the form $\delta(t, r) \longrightarrow t'$ (including instantaneous rules where $r = 0$) can be further specified for simulation purposes by means of a rewrite theory $\widehat{\mathcal{R}}$, where $\widehat{\mathcal{R}}$ contains $\mathcal{R}$ plus the following additional data:

(i) a new sort *GlobalState* and an operator $\widehat{\ } : \mathit{State} \rightarrow \mathit{GlobalState}$, and

(ii) tick rules of the form $\widehat{t} \longrightarrow \widehat{\delta(t, t_r)}$ **if** C.

Then, for any such $\widehat{\mathcal{R}}$ and terms t and t' not containing the symbol δ, a deduction $\widehat{\mathcal{R}} \vdash \widehat{t} \longrightarrow \widehat{t'}$ implies that there is an $r : \mathit{Time}$ such that $\mathcal{R} \vdash \delta(t, r) \longrightarrow t'$, and that then we can find a proof $\alpha : \widehat{t} \longrightarrow \widehat{t'}$ involving exactly n applications of "tick" rules, each advancing the time by $r_1, \ldots, r_n$, respectively, so that the additive equation $r = r_1 + \cdots + r_n$ holds. Therefore, in a simulation of this nature, we can not only observe the different evolutions in time of a system, but we can also measure the elapsed time by inspecting the rewrite proof corresponding to a given evolution.

Furthermore, if we only add the unconditional tick rules $\widehat{y} \longrightarrow \widehat{\delta(y, x_r)}$ for $y : State, x_r : Time$, we have $\widehat{\mathcal{R}} \vdash \widehat{t} \longrightarrow \widehat{t'}$ iff there is a time $r : Time$ such that $\mathcal{R} \vdash \delta(t, r) \longrightarrow t'$.

Example 2.2 (Example 2.1 cont.) Assuming that time only proceeds from states a and c, the system in Example 2.1 could be extended either by the tick rules $\{\widehat{a} \longrightarrow \widehat{\delta(a, 2)}, \widehat{c} \longrightarrow \widehat{\delta(c, 2)}\}$ or by the most general tick rules $\widehat{y} \longrightarrow \widehat{\delta(y, x_r)}$. In either case, we have $\forall t, t' \in T_{\Sigma - \{\delta\}}$, $\widehat{\mathcal{R}} \vdash \widehat{t} \longrightarrow \widehat{t'}$ iff $\delta(t, r) \longrightarrow t'$ for some $r \in \{0, 2, 4\}$.

In some cases, we could relax the general method by having tick rules of the form $\widehat{t} \longrightarrow \widehat{t'}$ **if** C which combine the effect of a tick and a "δ-rule". For instance, in Example 2.2, we could have tick rules $\widehat{a} \longrightarrow \widehat{b}$ and $\widehat{c} \longrightarrow \widehat{d}$ "directly" instead of the tick rules $\widehat{a} \longrightarrow \widehat{\delta(a, 2)}$ and $\widehat{c} \longrightarrow \widehat{\delta(c, 2)}$. While saving some rewrite steps and avoiding the symbol δ, this method has the disadvantage that it is not possible to extract information about the elapsed time from a proof $\alpha : \widehat{t} \longrightarrow \widehat{t'}$.

2.4 Simulation and Strategies

A real-time system runs, as it were, a "race against time". It often has to meet deadlines and to ensure that appropriate actions happen in a timely fashion. That is, among the different transitions that can be taken at a particular point in time, some may have top priority.

¿From a rewriting logic point of view this means that the specification of a real-time system may include not only the rewrite rules specifying its possible transitions, but also a *rewriting strategy*, which further constrains the correct rewrite behavior of the system. Using the reflective features of rewriting logic this can be done in a fully declarative way using a strategy language that is *internal*, and whose semantics is given by rewrite rules [6].

Some specifications do not need any strategies. When a strategy is needed, it has the following very simple form: the set $\mathcal{R}$ of rewrite rules is divided into a set $\mathcal{R}_{eager}$ of *eager* rules and a set $\mathcal{R}_{lazy}$ of *lazy* rules. Intuitively, an eager rule should be applied whenever enabled. Therefore, the rewriting strategy imposed by this division is that

no rule in $\mathcal{R}_{lazy}$ may be applied if any rule in $\mathcal{R}_{eager}$ is enabled.

If no eager rule is enabled, a one-step concurrent $\mathcal{R}_{lazy}$-rewrite [13] may take place. In case $\mathcal{R}_{eager}$ is empty, rules can be applied with no restrictions. In this paper, eager rules will be indicated by the keyword **eager**.

In the rewriting logic framework suggested above for real-time systems, the set $\mathcal{R}$ of rules can be split into sets $\mathcal{R}_{time}$ and $\mathcal{R}_{inst}$, where $\mathcal{R}_{time}$ lets time elapse in a system, and $\mathcal{R}_{inst}$ defines the instantaneous state changes. Furthermore, $\mathcal{R}_{time}$ can often be divided into two sets $\mathcal{R}_{tick}$ and $\mathcal{R}_{\delta}$, where $\mathcal{R}_{tick}$ lets time act on a system and $\mathcal{R}_{\delta}$ defines *how* time acts on it. Some of the state changes modeled by instantaneous rules are required to take place as soon as possible, that is, before time elapses. Therefore, the rules in $\mathcal{R}_{tick}$

are lazy, while $\mathcal{R}_{inst}$ may contain both eager and lazy rules.

Furthermore, tick rules should not advance the time beyond a point at which an eager instantaneous state change could have taken place. In this paper, this is ensured by appropriate conditions on the tick rules, and is not part of the rewriting strategy. However, for sufficiently complex systems it may be more convenient to preclude advancing time too much not by conditions on tick rules, but instead by a more sophisticated strategy than eagerness.

Although this is not a semantic requirement, for simulation purposes it can be quite convenient to apply the rules in $\mathcal{R}_\delta$ with an eager strategy. This has the advantage of being able to determine at what time an instantaneous rule was applied by inspection of a sequence of rewrites. However, by using the exchange rule stating equivalence of rewrite proofs [13], if the rules in R_δ are applied with a different strategy, it is always possible to normalize the proof so that such times can still be determined.

3 Example: Railroad Crossing in Rewriting Logic

In this section we show how the described framework for specifying real-time systems in rewriting logic can be used to give a Maude specification of a railroad crossing controller.

3.1 The Problem

The generalized railroad crossing (see, e.g. [8]) is a benchmark example of real-time systems. The system operates a gate at a railroad crossing. The crossing I lies in a region of interest R. A set of trains travel through R on multiple tracks. A sensor system determines when each train enters the region R (so that the gate can be down when, later on, the train enters the intersection I), and when it exits the intersection I.

The control program reacts to these enter and exit messages by sending messages for raising and lowering a gate to the environment. The system must satisfy that, whenever there is a possibility that a train is in the intersection, the gate should be down. However, the gate should be up as much as possible. We assume that:

- more than one train can be in R on the same track at the same time, and
- the minimum time for a train to enter I after entering R is R_to_I, the time to lower a gate (either from a raising position or when the gate is up) is denoted $time\_lower$. Hence, the (minimum) time ϵ from the time a train enters R until the gate must be lowered (in case it is not down or lowering already) is given by $\epsilon = R\_to\_I - time\_lower \geq 0$.

The solution should consist of the following parts:

(i) a specification of the control program, and

(ii) a model of the environment, which is used for simulation and validation purposes

satisfying the above requirements.

3.2 Outline of the Solution

The structure of the solution is straightforward: a *Xing* object represents the state of the control program, and an *env* object provides a simplistic model of the environment. The total system is the composition of these two objects.

Since for the purpose of controlling the crossing it does not matter on which tracks the trains are located, in the crossing object trains are represented by a multiset of time values, where a value r indicates that there is a train in the region, and that it could reach the intersection in time r; a value 0 indicates that a train could be in I.

The crossing object also includes the gate status. State *down* indicates that the gate is lowering or is down. Otherwise it is *up*.

The $TIME_\infty$ theory is extended to multisets of time as follows:

fmod $MULTI\text{-}TIME[T :: LTIME_\infty]$ **is**
  sort $Multi\_time$
  subsort $Time \leq Multi\_time$
  op $\emptyset_t : \rightarrow Multi\_time$
  op $\_\ \_ : Multi\_time, Multi\_time \rightarrow Multi\_time\ [assoc\ \ com\ \ id : \emptyset_t]$
  op $least : Multi\_time \rightarrow Time_\infty$
  op $\_ \dot{-} \_ : Multi\_time, Time \rightarrow Multi\_time$
  vars $x_r, y_r : Time$
  var $m : Multi\_time$
  eq $least(\emptyset_t) = \infty$
  eq $least(x_r\ \ m) = \min(x_r, least(m))$
  eq $\emptyset_t \dot{-} x_r = \emptyset_t$
  eq $(y_r\ m) \dot{-} x_r = (y_r \dot{-} x_r)\ (m \dot{-} x_r)$
endfm

The whole system, which we assume consists of only one crossing and one environment object, is given as a parameterized object-oriented module $XING[T :: TIME_\infty]$ with the following declarations:

protecting $MULTI\text{-}TIME[T]$
  sorts $Gatestate, GlobalConfiguration$
  ops $up, down : \rightarrow Gatestate$
  op $\delta : Configuration, Time \rightarrow Configuration$
  op $\hat{} : Configuration \rightarrow GlobalConfiguration$
  ops $R\_to\_I, time\_lower, time\_raise, time\_car : \rightarrow Time$
  vars $x_r, y_r : Time$
  vars $x, e : Oid$
  vars $m, m' : Multi\_time$
  class $Xing \mid trains : Multi\_time, gstate : Gatestate$
  msgs $enterR, exitI : \rightarrow Msg$

The *enterR* message is handled as follows:

$$\textbf{eager}\ (enterR)\langle x : Xing | trains : m\rangle \longrightarrow \langle x : Xing | trains : m\ R\_to\_I\rangle.$$

When a train exits I, the gate could be raised if there is time for a car to pass after the gate is up, and before the gate needs to be lowered again, otherwise it stays down:

$$\begin{aligned}\textbf{eager}\ & (exitI)\langle x:Xing|trains:0\ m, gstate:down\rangle \longrightarrow \\ & \textbf{if}\ least(m) - time\_lower \geq time\_car + time\_raise \\ & \textbf{then}\ \langle x:Xing|trains:m, gstate:up\rangle(raise) \\ & \textbf{else}\ \langle x:Xing|trains:m, gstate:down\rangle.\end{aligned}$$

Whenever the gate is up (or raising) and some train could have reached a time at which the gate should be lowered, it is lowered:

$$\begin{aligned}\textbf{eager}\ & \langle x:Xing|trains:time\_lower\ \ m, gstate:up\rangle \longrightarrow \\ & \langle x:Xing|trains:time\_lower\ \ m, gstate:down\rangle(lower).\end{aligned}$$

Time acts on a *Xing* object in the following way:

$$\textbf{eager}\ \delta(\langle x:Xing|trains:m\rangle, x_r) \longrightarrow \langle x:Xing|trains:m \dot{-} x_r\rangle.$$

3.3 The Environment

The behavior of the environment is quite simple. It consumes *lower* and *raise* messages that cause the appropriate actions; it produces *enterR* messages at any time, and it can produce *exitI* messages within certain time constraints relative to *enterR* messages. Here we assume that the time from the instant a train reaches R until it exits I is between ϵ_1 and ϵ_2 where $\epsilon_2 > \epsilon_1$ and $\epsilon_1 > R\_to\_I$.

In the environment object the set of trains is represented as a multiset of times, where for each train we keep the amount of time that must elapse for it to exit the region I. For validation purposes only, two other attributes are added to the environment:

- An attribute *trains_to_I*, a multiset of time values, where a value r represents a train which will enter I in time r. A value 0 indicates that the train has entered the intersection.
- An attribute *gpos* which models the state of the gate accurately, where *lowering*(r) (resp. *raising*(r)) means that the gate is being lowered (resp. raised) and will be down (resp. up) in time r.

We define the environment as an object in a class *env* in the same module *XING* by

sort *Gateposition*
ops *lowering*, *raising* : *Time* → *Gateposition*
ops $\epsilon_1, \epsilon_2 : \rightarrow Time$
class *env* | *trains*, *trains_to_I* : *Multi_time*, *gpos* : *Gateposition*
msgs *lower*, *raise* : → *Msg*

Messages from the crossing object are treated as follows:

$$\begin{aligned}
&\textbf{eager } (lower)\langle e : env | gpos : raising(x_r)\rangle \longrightarrow \\
&\qquad \langle e : env | gpos : lowering(time_lower)\rangle \\
&\textbf{eager } (raise)\langle e : env | gpos : lowering(x_r)\rangle \longrightarrow \\
&\qquad \langle e : env | gpos : raising(time_raise)\rangle
\end{aligned}$$

$$\begin{aligned}
&\textbf{eager } (raise)\langle e : env | gpos : raising(x_r)\rangle \longrightarrow \\
&\qquad \langle e : env | gpos : raising(x_r)\rangle \\
&\textbf{eager } (lower)\langle e : env | gpos : lowering(x_r)\rangle \longrightarrow \\
&\qquad \langle e : env | gpos : lowering(x_r)\rangle.
\end{aligned}$$

The *enterR* messages are created arbitrarily, but not whenever possible (therefore, the rule is not eager):

$$\begin{aligned}
&\langle e : env | trains : m, trains_to_I : m'\rangle \longrightarrow \\
&\qquad \langle e : env | trains : m \;\; x_r, trains_to_I : m' \;\; y_r\rangle (enterR) \\
&\qquad \textbf{if } \epsilon_1 \leq x_r \leq \epsilon_2 \text{ and } R_to_I \leq y_r < x_r.
\end{aligned}$$

Whenever a train leaves I, an *exitI* message must be sent:

$$\begin{aligned}
&\textbf{eager } \langle e : env | trains : m\, 0, trains_to_I : m'\, 0\rangle \longrightarrow \\
&\qquad \langle e : env | trains : m, trains_to_I : m'\rangle (exitI).
\end{aligned}$$

Time acts on an *env* object according to the following rules:

$$\begin{aligned}
&\textbf{eager } \delta(\langle e : env | trains : m, trains_to_I : m', gpos : lowering(y_r)\rangle, x_r) \longrightarrow \\
&\quad \langle e : env | trains : m \dot{-} x_r, trains_to_I : m' \dot{-} x_r, gpos : lowering(y_r \dot{-} x_r)\rangle \\
&\textbf{eager } \delta(\langle e : env | trains : m, trains_to_I : m', gpos : raising(y_r)\rangle, x_r) \longrightarrow \\
&\quad \langle e : env | trains : m \dot{-} x_r, trains_to_I : m' \dot{-} x_r, gpos : raising(y_r \dot{-} x_r)\rangle.
\end{aligned}$$

3.4 The Combined System

Intuitively, the system can proceed in time until either the gate is up and a train could have reached the position where the gate should be lowered, or a train must exit region I, or a train enters R. Note that in this example, we use prefix notation for the symbol $\widehat{\ }$.

$$\begin{aligned}
&tick_1 : \widehat{\ }(\langle x : Xing | trains : m, gstate : up\rangle\langle e : env | trains : m'\rangle) \longrightarrow \\
&\qquad \widehat{\ }(\delta(\langle x : Xing | trains : m, gstate : up\rangle\langle e : env | trains : m'\rangle, x_r)) \\
&\qquad \textbf{if } (m == \emptyset_t) \text{ or } (x_r \leq least(m) \dot{-} time_lower \text{ and } x_r \leq least(m')).
\end{aligned}$$

In case the gate is down, the system can proceed until a train leaves the region I, which is when a train value in the environment is 0:

$$tick_2 : \widehat{\ }(\langle x : Xing | trains : m, gstate : down \rangle \langle e : env | trains : m' \rangle) \longrightarrow$$
$$\widehat{\ }(\delta(\langle x : Xing | trains : m, gstate : down \rangle \langle e : env | trains : m' \rangle, x_r))$$
$$\textbf{if } x_r \leq least(m').$$

The fact that time increases non-deterministically by $x_r \leq least(m') \dot{-} time\_lower$ and $x_r \leq least(m')$ instead of by $x_r = \min(least(m) \dot{-} time\_lower, least(m'))$ allows non-deterministic rewrites where $enterR$ messages are sent at arbitrary times.

In general, time acts independently on each object:

$$\delta(yz, x_r) = \delta(y, x_r)\, \delta(z, x_r)$$

for $y, z : Configuration$. Note that, given the eagerness with which all messages are processed and the δ-rules are applied, there are never any messages present in a configuration when a tick rule is applied.

Once we have a specification of this kind, we can execute it, to simulate the behavior of the intended system, and to uncover some possible bugs in the specification itself. This form of symbolic simulation can already prove certain properties of the system as sequents derivable from the specification. In addition, the initial model of the rewriting logic specification [13] provides a precise mathematical model against which formal statements about the behavior that the system must exhibit can be verified. For example, one could show that whenever a train is in the intersection (represented by a value 0 in the environments $trains\_to\_I$ attribute), the gates are down. One way of proving this is to show that whenever the initial configuration rewrites to a state of the form

$$\widehat{\ }(\langle x : Xing \rangle \langle e : env | trains\_to\_I : 0 \;\; m, gpos : g \rangle M)$$

(for M a multiset of messages), then the value of g is $lowering(0)$.

4 Rewriting Logic as a Semantic Framework for Real-Time Systems

This section illustrates how a variety of models of real-time systems have a natural translation into rewriting logic. Apart from Section 4.1, we concentrate on rewriting logic specifications that can be used directly for "simulating" the corresponding systems. The method indicated in Section 4.1 can be used to reason about quantitative properties of the systems so specified.

4.1 Timed Automata

Omitting details about initial states and acceptance conditions, a timed automaton (see, e.g., [3]) consists of:

- a finite alphabet Σ,
- a finite set S of states,

- a finite set C of clocks,
- a set $\Phi(C)$ of clock constraints defined inductively by
$$\phi := c \leq k \mid k \leq c \mid \neg\phi \mid \phi_1 \wedge \phi_2$$
where c is a clock in C and k is a constant in the set of nonnegative rationals, and
- a set $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ of transitions. The tuple $\langle s, s', a, \lambda, \phi\rangle$ represents a transition from state s to state s' on input symbol a. The set $\lambda \subseteq C$ gives the clocks to be reset with this transition, and ϕ is a clock constraint over C.

Given a timed word (i. e. a sequence of tuples $\langle a_i, r_i\rangle$ where a_i is an input symbol and r_i is the time at which it occurs), the automaton starts at time 0 with all clocks initialized to 0. As time advances the values of all clocks change, reflecting the elapsed time; that is, the state of the automaton can change not only by the above transitions, but also by the passage of time, with all the clocks being increased by the same amount. At time r_i the automaton changes state from s to s' using some transition of the form $\langle s, s', a_i, \lambda, \phi\rangle$ reading input a_i, if the current values of the clocks satisfy ϕ. With this transition the clocks in λ are reset to 0, and thus start counting time again.

A timed automaton can be naturally represented in rewriting logic. The time domain and its associated constraints $\Phi(C)$ are equationally axiomatized in an abstract data type satisfying the theory *TIME*. Then, the tuple $\langle s, c_1, \ldots, c_n\rangle$ represents an automaton in state s such that the values of the clocks in C are $c_1, \ldots, c_n$. Each transition $\langle s, s', a, \lambda, \phi\rangle$ is then expressed as a rewrite rule
$$a : \langle s, c_1, \ldots, c_n\rangle \longrightarrow \langle s', c_1', \ldots, c_n'\rangle \textbf{ if } \phi(c_1, \ldots, c_n)$$
where $c_i' = 0$ if $c_i \in \lambda$, and $c_i' = c_i$ otherwise. In addition, a rule
$$tick : \langle x, c_1, \ldots, c_n\rangle \longrightarrow \langle x, c_1 + x_r, \ldots, c_n + x_r\rangle$$
(where $x, x_r, c_1, \ldots, c_n$ all are variables) is added to represent the elapse of time. It is easy to show that the resulting rewriting logic specification faithfully represents the timed automaton.

Using the ideas of "time as an action" and of tick rules advancing time for a system as a whole, we can give a somewhat more detailed specification of a timed automaton. We leave the transition rewrite rules unchanged, but replace the rewrite rule
$$tick : \langle x, c_1, \ldots, c_n\rangle \longrightarrow \langle x, c_1 + x_r, \ldots, c_n + x_r\rangle$$
by the following two rules:
$$tick : \langle s, \widehat{c_1, \ldots}, c_n\rangle \longrightarrow \delta(\langle x, \widehat{c_1, \ldots}, c_n\rangle, x_r),$$
$$\delta : \delta(\langle x, c_1, \ldots, c_n\rangle, x_r) \longrightarrow \langle x, c_1 + x_r, \ldots, c_n + x_r\rangle.$$
That is, we decompose the original tick into two steps, namely the action of time and the subsequent passage to the resulting new state. In this way, we have two possibilities available to us. On the one hand, time elapse properties of the timed automaton such as a rewrite $\delta(\langle s_0, \ldots\rangle, 50) \longrightarrow \langle s_n, \ldots\rangle$ can be

proved about the system. On the other hand, for simulation purposes the rewrite rules can be used to derive all the possible runs of the automaton in the form:

$$\langle s_0, \widehat{0, \ldots}, 0\rangle \xrightarrow{tick} \delta(\langle s_0, \widehat{0, \ldots}, 0\rangle, r) \xrightarrow{\delta} \langle s_0, \widehat{r, \ldots}, r\rangle \xrightarrow{a} \langle s_1, \widehat{r, \ldots}, 0\rangle \xrightarrow{tick} \cdots$$

4.2 Hybrid Automata

The time model of *hybrid automata* [2] is the real numbers. However, to get a computable data type, we should replace the reals by a computable subfield $\mathbb{R}$, such as the algebraic real numbers. A hybrid automaton is given by a tuple $\langle V_D, Loc, Act, Inv, Edg\rangle$ (we omit acceptance conditions and initial states) where:

- V_D is a finite set of real-valued data variables, defining the data space Σ_D, that is, Σ_D is the function space $[V_D \to \mathbb{R}]$.
- Loc is a finite set of *locations* (corresponding to "states" in untimed automata).
- Act is a labeling function that assigns to each location $l \in Loc$ a set Act_l of *activities*. An activity is a function from $\mathbb{R}_+$ to Σ_D. A system in state $\langle l, \overline{v}\rangle$ evolves to $\langle l, f(r)\rangle$ in time r whenever f is an activity of location l such that $\overline{v} = f(0)$.
- Inv is a labeling function that assigns to each location $l \in Loc$ an *invariant* $Inv(l) \subseteq \Sigma_D$.
- Edg is a finite set of *transitions*. Each transition $e = (l, \mu, l')$ consists of a source location l, a target location l', and a transition relation $\mu \subseteq \Sigma_D^2$. For each location l there is a *stutter transition* (l, Id, l) where $Id = \{(\overline{v}, \overline{v}) \mid \overline{v} \in \Sigma_D\}$.

At any time instant, the state $\langle l, \overline{v}\rangle$ of a hybrid system specifies a control location and values for all data variables, i.e., the state space is $Loc \times \Sigma_D$. The state can change in two ways: (1) by an instantaneous transition that changes the entire state according to the transition relation, or (2) by elapse of time that changes only the values of data variables in a continuous manner according to the activities of the current location. The system may stay at a location only if the invariant at the location is true. The invariants of a hybrid automaton thus enforce the progress of the underlying discrete transition system: some transition must be taken before the invariant of the location is false.

As in [13], where a transition τ from state s to state s' is modeled by a rewrite rule $\tau : s \longrightarrow s'$, it is assumed that the transitions Edg can be expressed by rewrite rules

$$\langle l, \overline{v}\rangle \longrightarrow \langle l', \overline{v'}\rangle.$$

To specify the continuous behavior of a system, one needs to know the maximum time such that, given a state $\langle l, \overline{v}\rangle$, control can stay at location l performing the activity f without violating the invariant of location l. We

assume that, for each location l, this is given by a function

$$max\_stay_l : Act_l \to Time_\infty$$

The tick rules of the system are given by

$$tick : \langle l, f(0)\rangle \longrightarrow \langle l, f(x_r)\rangle \textbf{ if } x_r \leq max\_stay_l(f)$$

for all locations l and for each activity f in l.

4.3 Timed Transition Systems

A *timed transition system* (TTS) [11], whose time domain is the set $\mathbb{N}$ of natural numbers, is a transition system (with a finite number of transitions) where each transition τ is equipped with a "lower bound" $l_\tau \in \mathbb{N}$ and an "upper bound" $u_\tau \in \mathbb{N} \cup \{\infty\}$. A transition τ cannot be taken if it hasn't been enabled uninterruptedly for at least l_τ time units, and if τ is enabled at time n, then either τ is disabled or taken somewhere in the interval $[n, n+u_\tau]$. In the TTS model, there is no continuous change of state.

As for hybrid automata, we assume that the underlying untimed transition system can be specified in rewriting logic as shown in [13]. A TTS is represented in rewriting logic by just adding to each state one clock for each transition, where the clock c_i has value *nil* if transition τ_i is not enabled, and t_i if the transition has been enabled continuously for time t_i (without being taken). The symbol *nil* is therefore an element of a supersort of the natural numbers, satisfying the equation $nil + x = nil$ for $x : Nat$. We also assume that for each transition τ_i, there is a predicate $enabled_i$ such that $enabled_i(s)$ is true if transition τ_i is enabled on state s and false otherwise.

A state of the system is thus represented as a term $\langle s, c_1, \ldots, c_n\rangle$, where s is the state of the transition system, and the c_i's are the clocks. A transition

$$\tau_i : s \longrightarrow s'$$

in the TTS is modeled by a rewrite rule

$$\tau_i : \langle s, c_1, \ldots, c_n\rangle \longrightarrow \langle s', c_1', \ldots, c_n'\rangle \textbf{ if } c_i \geq l_{\tau_i}$$

where for all $j = 1, \ldots, n$,

$c_j' =$ **if** $not(enabled_j(s'))$ **then** nil **else if** $c_j == nil$ or $i == j$ **then** 0 **else** c_j.

Time can elapse if no transition must be taken:

$$tick : \langle x, c_1, \ldots, c_n\rangle \longrightarrow \langle x, c_1 + x_r, \ldots, c_n + x_r\rangle \textbf{ if } \bigwedge_i (c_i + x_r \leq u_{\tau_i} \text{ or } c_i == nil)$$

4.4 Phase Transition Systems

Phase transition systems (PTS's) [11] extend timed transition systems to hybrid systems. In a PTS the set V of variables defining the state space is divided into two parts: the set V_c of continuous real variables, again, for $\mathbb{R}$ a computable subfield of the reals, and the set V_d of discontinuous real variables. The continuous variables change their values with the elapse of time according to activities of the form

$$P \Longrightarrow \dot{v}_{c_i} t,$$

where P is a Boolean expression over the discontinuous variables and t is a term over V. We assume without loss of generality that the continuous behavior of each continuous variable is described by one such activity.

The set $\mathcal{T}$ of instantaneous transitions is, as in the TTS case, equipped with upper and lower bounds.

To give a rewriting logic specification, the set V of variables must be finite, and the following functions must be computable:

- the effect of letting time act on a continuous variable v_{c_i}, given the current state, is assumed given by a function
$$g_i : \mathbb{R}^{m+n}, \mathbb{R}_+ \to \mathbb{R}$$
where $g_i(v_{c_1}, \ldots, v_{c_m}, v_{d_1}, \ldots, v_{d_n}, r)$ gives the value of v_{c_i} when the system proceeds r time units from the state $\langle v_{c_1}, \ldots, v_{c_m}, v_{d_1}, \ldots, v_{d_n} \rangle$, and
- a function
$$f : \mathbb{R}^{m+n} \to \mathbb{R}$$
which takes a state as argument and gives the maximum time the system can proceed without changing the enabledness of a transition.

We represent the state of a PTS as the term

$$\langle v_{c_1}, \ldots, v_{c_m}, v_{d_1}, \ldots, v_{d_n}, c_1, \ldots, c_k \rangle$$

where v_{c_i} (v_{d_i}) denotes the current value of the PTS variable v_{c_i} (resp. v_{d_i}) and c_i indicates (as in the timed transition system case) for how long transition τ_i has been enabled without being taken.

The discrete parts of the system are given as in the TTS case, i.e.,

$$\tau_i : \langle \overline{v_c}, \overline{v_d}, c_1, \ldots, c_k \rangle \longrightarrow \langle \overline{v'_c}, \overline{v'_d}, c'_1, \ldots, c'_k \rangle \textbf{ if } l_{\tau_i} \leq c_i$$

where for all $j = 1, \ldots, n$,

$$c'_j = \textbf{ if } not(enabled_j(s')) \textbf{ then } nil \textbf{ else if } c_j == nil \text{ or } i == j \textbf{ then } 0 \textbf{ else } c_j.$$

Specifying the elapse of time we must ensure that:

- all continuous variables are updated,
- if any enabling condition changes, then the corresponding clocks are updated, and
- time cannot elapse beyond the point at which an enabled transition must be taken.

The following tick rules handle these cases:

$$tick_1 : \langle v_{c_1}, \ldots, v_{c_m}, \overline{v_d}, c_1, \ldots, c_k \rangle \longrightarrow$$
$$\langle g_1(\overline{v_c}, \overline{v_d}, x_r), \ldots, g_m(\overline{v_c}, \overline{v_d}, x_r), \overline{v_d}, c_1 + x_r, \ldots, c_k + x_r \rangle$$
$$\textbf{if } \textstyle\bigwedge_i (c_i + x_r \leq u_{\tau_i} \text{ or } c_i == nil) \text{ and } x_r < f(\overline{v_c}, \overline{v_d})$$

$$tick_2 : \langle v_{c_1}, \ldots, v_{c_m}, \overline{v_d}, c_1, \ldots, c_k \rangle \longrightarrow$$
$$\langle g_1(\overline{v_c}, \overline{v_d}, x_r), \ldots, g_m(\overline{v_c}, \overline{v_d}, x_r), \overline{v_d}, c'_1, \ldots, c'_k \rangle$$
$$\textbf{if } \textstyle\bigwedge_i (c_i + x_r \leq u_{\tau_i} \text{ or } c_i == nil) \text{ and } x_r = f(\overline{v_c}, \overline{v_d})$$

where for all $j = 1, \ldots, n$,

$$c_j' = \textbf{ if } not(enabled_j(s')) \textbf{ then } nil \textbf{ else if } c_j == nil \textbf{ then } 0 \textbf{ else } c_j + x_r.$$

In the rule $tick_2$, time is advanced until an enabling condition changes; therefore, all enabling conditions must be reevaluated to their new values $c_1', \ldots, c_k'$.

4.5 Timed Petri Nets

Petri nets have been extended to model real-time systems in different ways (see e. g. [1,17]). Three of the most used time extensions are the following:

(i) Each *transition* α has an associated time interval $[l_\alpha, u_\alpha]$. A transition fires as soon as it can, but the resulting tokens are delayed, i. e., they are not visible in the system until some time $t \in [l_\alpha, u_\alpha]$ after the transition fires.

(ii) Each *place* p has a duration t_p. A token at place p cannot participate in a transition until it has been at p for at least time t_p.

(iii) Each transition α is associated with a time interval $[l_\alpha, u_\alpha]$ and the transition α cannot fire before it has been continuously enabled for at least time l_α. Also, the transition α cannot have been enabled continuously for more than time u_α without being taken.

We only treat the first two cases. The third case can be given a treatment similar to that of timed transition systems.

The translation into rewriting logic of these first two cases is based on the rewriting logic representation of untimed Petri nets given in [13], where the state of a Petri net is represented by a multiset of places—where if place p has multiplicity n we interpret this as the presence of n tokens at the place—and where the transitions correspond to rewrite rules on the corresponding multisets of pre- and post-places.

In our representation, a token at a place p is denoted by a term

$$[p].$$

A token that will be available at place p in time r is represented by the term

$$dly(p, r).$$

A state of a timed Petri net is a multiset of these two forms of placed tokens, where multiset union is represented by juxtaposition.

Time acts on a placed token as follows:

$$\delta([p], x_r) \longrightarrow [p]$$
$$\delta(dly(p, x_r), x_r) \longrightarrow [p]$$

and time distributes over multisets of placed tokens:

$$\delta(\emptyset, x_r) = \emptyset$$
$$\delta(xy, x_r) = \delta(x, x_r)\, \delta(y, x_r),$$

where x, y range over multisets of placed tokens and $\emptyset$ denotes the empty multiset.

Transitions are represented by rewrite rules. In the first case of timed Petri nets, also known as *interval timed Petri nets* [1], each transition α has an associated interval $[l_\alpha, u_\alpha]$. Assume that the transition α consumes two tokens from place a, and one token from place b, and produces one token each at places c and d. Since the duration of the transition is any time in the interval $[l_\alpha, u_\alpha]$, the resulting tokens are not visible for a time within this interval. Hence the transition α can be represented by the following rewrite rule:

$$\textbf{eager } [a][a][b] \longrightarrow dly(c, x_r)\ dly(d, x_r) \textbf{ if } l_\alpha \leq x_r \leq u_\alpha.$$

In the second version of timed Petri nets, each place p has an associated duration t_p, and a token must have been at a place p for at least time t_p before it can be used in any transition. This is equivalent to saying that the produced token cannot be visible before time t_p after the producing transition took place. Hence the transition that consumes two tokens from place a and one from place b, and which produces one token each at c and d is represented in rewriting logic by the rule

$$\textbf{eager } [a][a][b] \longrightarrow dly(c, t_c)\ dly(d, t_d).$$

As usual, the elapse of time (in both versions) is modeled by tick rules. In order to ensure that time does not proceed beyond the time when a transition could fire (that is, when time has acted on a token $dly(p, r)$ for time r), the operator *least_tick* is used. It takes as argument a multiset of placed tokens, returns the time until one or more non-available tokens become available, and is defined in the following way:

$$\begin{aligned}
least_tick(\emptyset) &= \infty \\
least_tick([p]) &= \infty \\
least_tick(\delta([p], x_r)) &= 0 \textbf{ if } 0 < x_r \\
least_tick(\delta(dly(p, x_r), y_r)) &= x_r \dot{-} y_r \\
least_tick(x\ y) &= \min(least_tick(x), least_tick(y)).
\end{aligned}$$

The tick rule then allows time to elapse until the first dly-token becomes visible [3]:

$$tick : \widehat{N} \longrightarrow \delta(N, \widehat{least\_tick}(N)).$$

In this case the operator $\widehat{\ }$ is needed. Otherwise, time could elapse only in parts of the whole system N.

In both versions of timed Petri nets transitions are supposed to fire as soon as possible. This is accomplished by the strategy described in Section 2.4 that triggers all eager instantaneous rules until none of these can be applied, followed by one application of the tick rule. No explicit eagerness is required for the rules in R_δ, since time will not elapse (by a time value greater than 0) if these are not applied whenever enabled.

[3] A more general tick rule $\widehat{N} \longrightarrow \delta(\widehat{N, x_r})$ **if** $x_r \leq least\_tick(N)$ would not do any good, since nothing can be done until the next unavailable token becomes available.

5 Relationship to Timed Rewriting Logic

In this section we investigate the relationship between timed rewriting logic and the described framework for specifying real-time systems directly in rewriting logic. After briefly introducing TRL in Section 5.1, we propose in Section 5.2 a translation from TRL into rewriting logic. In this translation, the translation of any derivable TRL-sequent in a TRL theory is derivable in the corresponding rewriting logic theory. The converse is in general not true. We explain the reasons for this discrepancy in Section 5.3. They are due to some conceptual differences between TRL and our method of specifying real-time systems in rewriting logic. However, in Section 5.5 we show how the proposed translation in many cases can be extended to simulate systems specified in TRL by means of their rewriting logic translations.

5.1 Timed Rewriting Logic

Rewriting logic has been extended by Kosiuczenko and Wirsing to handle real-time systems in their *timed rewriting logic* (TRL) [9,10]. TRL has been shown well-suited for giving object-oriented specifications of complex hybrid systems such as the steam-boiler [18] and has been illustrated by a number of specifications of simpler real-time systems. A translation into ordinary rewriting logic can illuminate the conceptual relationships between both formalisms. Also, since rewriting logic seems easier to implement than TRL and in fact several such implementations already exist, such a translation can also provide a convenient path to make TRL specifications executable.

In TRL each rewrite step is labeled with a time stamp. TRL rules are sequents of the form $t \xrightarrow{r} t'$, for r a ground term of sort *Time*. Their intuitive meaning is that t evolves to t' in time r.

More specifically, assume given a TRL theory, i.e., a set of equations and timed rewrite rules satisfying the theory *TIME* [4]. Then, the set of derivable sequents consists of all rules in the specification, and all sequents which can be derived by equational reasoning and by using the deduction rules in Figure 1, where $\mathcal{V}(t)$ denotes the set of free variables in t.

This deduction system extends and modifies the rules of deduction in rewriting logic with time stamps as follows:

- Reflexivity is dropped as a general axiom, to allow specifying hard real-time systems. Reflexivity would not allow describing hard real-time systems since (parts of) the system could stay idle for an arbitrary long period of time. For specifying soft real-time systems particular reflexivity axioms can be added.
- Transitivity yields the addition of the time stamps. If t_1 evolves to t_2 in time r_1 and t_2 evolves to t_3 in time r_2, then t_1 evolves to t_3 in time $r_1 + r_2$.
- The synchronous replacement rule enforces uniform time elapse in all com-

[4] They impose in some cases further requirements, such as *TIME* being an Archimedean monoid. This could of course be easily accomodated.

Timed transitivity (TT):

$$\frac{t_1 \xrightarrow{r_1} t_2 \qquad t_2 \xrightarrow{r_2} t_3}{t_1 \xrightarrow{r_1+r_2} t_3}$$

Synchronous replacement (SR):

$$\frac{t_0 \xrightarrow{r} t_0', \;\; t_{i_1} \xrightarrow{r} t_{i_1}', \ldots, t_{i_k} \xrightarrow{r} t_{i_k}'}{t_0(t_1/x_1, \ldots, t_n/x_n) \xrightarrow{r} t_0'(t_1'/x_1, \ldots, t_n'/x_n)}$$

where $\{x_{i_1}, \ldots, x_{i_k}\} = \mathcal{V}(t_0) \cap \mathcal{V}(t_0')$.

Compatibility with equality (EQ):

$$\frac{t_1 = u_1,\ r_1 = r_2,\ t_2 = u_2,\ t_1 \xrightarrow{r_1} t_2}{u_1 \xrightarrow{r_2} u_2}$$

Renaming of variables (RV):

$$x \xrightarrow{r} x \qquad \text{for all } x \in X, r \in T_{\Sigma_{Time}}$$

Fig. 1. *Deduction rules in timed rewriting logic.*

ponents of a system: a system rewrites in time r iff all its components do so. Synchronous replacement combined with irreflexivity also induces maximal parallelism, which means that no component of a process can stay idle.

- The renaming rule assures that timed rewriting is independent of the names of variables. Observe that the renaming axiom does not imply that $t \xrightarrow{r} t$ holds for all terms t.

Example 5.1 From the timed rewrite specification $\{f(x) \xrightarrow{1} g(x), g(x) \xrightarrow{1} h(x), a \xrightarrow{2} b\}$, where time is modeled by the natural numbers, the sequent $f(a) \xrightarrow{2} h(b)$ can be deduced by first deducing $f(x) \xrightarrow{2} h(x)$ using transitivity and then applying the synchronous replacement rule. Due to the lack of $\xrightarrow{r}$-reflexivity, neither $a \xrightarrow{2} a$ nor $f(a) \xrightarrow{2} h(a)$ are derivable. Note that $f(a) \xrightarrow{2} h(b)$ can not be deduced without using the synchronous replacement rule.

5.2 Timed Rewriting Logic in Rewriting Logic

In this section we define a mapping from timed rewriting logic to rewriting logic. In other words, given a TRL specification $\mathcal{T}$, there is a mapping $\mathcal{M}$ sending $\mathcal{T}$ to $\mathcal{M}(\mathcal{T})$, and sending TRL sequents $t \xrightarrow{r} t'$ to sequents $\mathcal{M}(t \xrightarrow{r} t')$ in rewriting logic, such that $\mathcal{T} \vdash t \xrightarrow{r} t'$ implies that $\mathcal{M}(\mathcal{T}) \vdash \mathcal{M}(t \xrightarrow{r} t')$ for all terms t, t'.

We restrict our treatment to TRL theories where no extra variables are introduced in the righthand side of a rule. The reason for this restriction is that if $f(x) \xrightarrow{2} g(x, y)$ and $g(x, y) \xrightarrow{2} h(y)$ are two rules, any system t' that appears in $h(t)$ as a result of the second rule, must have evolved for time 2 from a system t in $g(u, t)$. However, by transitivity of the rules, the sequent $f(x) \xrightarrow{4} h(y)$ is derivable, which means that *any* system t could replace y in $h(y)$, including the systems which have not evolved for time 2.

The idea of the translation from TRL to rewriting logic is that a TRL sequent $t \xrightarrow{r} t'$ ("t evolves in time r to t'") maps to a rewriting logic sequent $\delta(t, r) \longrightarrow t'$ ("if time has acted on t for time r, then it rewrites to t'") for ground terms t, t'. If t contains a variable x, the subsystem t_i by which x is instantiated in t should have evolved r time units in t'. A sequent $t \xrightarrow{r} t'$ is therefore mapped to

$$\delta(t, r) \longrightarrow t'(\delta(x_1, r)/x_1, \ldots, \delta(x_n, r)/x_n)$$

for the set $\{x_1, \ldots, x_n\} \supseteq \mathcal{V}(t') \subseteq \mathcal{V}(t)$ of variables.

The mapping $\mathcal{M}$ from TRL-sequents to rewriting logic sequents is then given by

$$\mathcal{M}(t \xrightarrow{r} t') = \delta(t, r) \longrightarrow t'(\delta(x_1, r)/x_1, \ldots, \delta(x_n, r)/x_n)$$

where the set of free variables in t' is a subset of $\{x_1, \ldots, x_n\}$.

This map $\mathcal{M}$ is extended to a map sending a TRL theory $\mathcal{T} = \langle \Sigma, E, R \rangle$ satisfying the above restrictions of no extra variables in righthand sides to a rewriting logic theory $\mathcal{M}(\mathcal{T})$ by

$$\begin{aligned}
\mathcal{M}(\Sigma) &= \Sigma \cup \{\delta : s, \mathit{Time} \to s \mid s \in \mathit{sorts}(\Sigma)\} \\
\mathcal{M}(E) &= E \cup \{(\forall x : s, x_r, y_r : \mathit{Time})\ \ \delta(x, 0) = x,\ \delta(\delta(x, x_r), y_r) = \delta(x, x_r + y_r) \\
&\qquad \mid s \in \mathit{sorts}(\Sigma)\} \\
\mathcal{M}(R) &= \{\mathcal{M}(\rho) \mid \rho \in R\}
\end{aligned}$$

Therefore, $\mathcal{M}$ can naturally be understood as a map of logics. Specifically, as a map $\mathcal{M} : \mathit{TRL} \longrightarrow \mathit{RWL}$ from the entailment system [15] of TRL to that of rewriting logic.

Theorem 5.2 *Let $\mathcal{T}$ be a TRL specification and let $\mathcal{M}$ be defined as above. Then for all terms t, t', r*

$$\mathcal{T} \vdash t \xrightarrow{r} t' \text{ implies } \mathcal{M}(\mathcal{T}) \vdash \mathcal{M}(t \xrightarrow{r} t').$$

As a corollary of this theorem, which can be easily proved by induction on the size of the proof $t \xrightarrow{r} t'$, we obtain that $\mathcal{T} \vdash t \xrightarrow{r} t'$ implies $\mathcal{M}(\mathcal{T}) \vdash \delta(t, r) \longrightarrow t'$ for all ground terms t, t', and r.

Example 5.3 (Example 5.1 cont.) The translation of the TRL specification in Example 5.1 is given by the equations for the action δ and the rules

$$\{\delta(f(x), 1) \longrightarrow g(\delta(x, 1)), \delta(g(x), 1) \longrightarrow h(\delta(x, 1)), \delta(a, 2) \longrightarrow b\}.$$

The sequent $\delta(f(a), 2) \longrightarrow h(b)$ corresponding to $f(a) \xrightarrow{2} h(b)$ is obtained by $\delta(f(a), 2) \longrightarrow \delta(g(\delta(a, 1)), 1) \longrightarrow h(\delta(a, 2)) \longrightarrow h(b)$, where use of the equation $\delta(\delta(x, x_r), y_r) = \delta(x, x_r + y_r)$ is not shown explicitly.

5.3 Differences Between TRL and its Rewriting Logic Translation

Even though $t \xrightarrow{r} t'$ implies $\delta(t, r) \longrightarrow t'$ for ground terms, the converse is not necessarily true. In this section the differences between deduction in TRL and in its translation into rewriting logic are outlined.

5.3.1 Zero-Time Idling

In the rewriting logic translation, a TRL sequent $t \xrightarrow{0} t$ translates to $\delta(t,0) \longrightarrow t(\delta(x_1,0)/x_1, \ldots, \delta(x_n,0)/x_n)$, which due to the axiom $\delta(x,0) = x$ is equal to $t \longrightarrow t$, which is always deducible in rewriting logic. However, in TRL, $t \xrightarrow{0} t$ is not necessarily valid. This obviously indicates a difference between both systems, since the notion of "zero-time idling" is always available in our approach but not in TRL.

5.3.2 Non-Right-Linear Rules

Given the TRL theory $\{f(x) \xrightarrow{2} g(x,x), a \xrightarrow{2} b, a \xrightarrow{2} c\}$, the term $f(a)$ rewrites to either $g(b,b)$ or $g(c,c)$ in time two, but will *not* rewrite to $g(b,c)$. In the rewriting logic translation $\{\delta(f(x),2) \longrightarrow g(\delta(x,2),\delta(x,2)), \delta(a,2) \longrightarrow b, \delta(a,2) \longrightarrow c\}$ there is a rewrite $\delta(f(a),2) \longrightarrow g(\delta(a,2),\delta(a,2)) \longrightarrow g(b,c)$.

The difference depends on how one models the fork of a process. The rule $f(x) \xrightarrow{r} g(x,x)$ can be understood as a fork of the (sub)process t in the system $f(t)$. In the TRL setting, the actual "fork" (the point in time when the two instances of the process x can behave independently of each other) is taking place at the end of the time period of length r in the rule. In the rewriting logic setting, the "forking" took place at the beginning of the time period of duration r [5].

5.3.3 Problems Related to Synchronicity in TRL

Another aspect in which TRL and our rewriting logic translation are different is illustrated by the following TRL specification:

$$\{f(a,y) \xrightarrow{2} g(a,y), g(x,y) \xrightarrow{2} h(x,y), h(x,c) \xrightarrow{2} k(x,c), a \xrightarrow{4} d, b \xrightarrow{4} c\}.$$

Due to the strong synchronicity requirements in TRL, $f(a,b)$ cannot be rewritten, even though the b (in the place of y), and a (for x), could be rewritten in time 4. In many cases, it would however be natural to assume that the system represented by $f(a,b)$ rewrites to $k(d,c)$ in time 6.

In the rewriting logic translation, $\delta(f(a,b),6)$ rewrites to $k(d,c)$.

5.4 Aging in TRL

To overcome the strong requirements of synchronicity in TRL, which caused the differences in Sections 5.3.2 and 5.3.3, the special symbol *age* is introduced in [9,10]. It aims at making a term t, which rewrites in time r', "accessible" to synchronous rewrites in time r with $r' \geq r$, by making it visible as $age(t,r)$.

Formally, with aging, the following two deduction rules are added to the TRL deduction rules given in Figure 1. In both deduction rules, $t \xrightarrow{r+r'} t'$ is

[5] Note that in the rewriting logic setting, adding a rule $\delta(k(x),2) \longrightarrow f(\delta(x,2))$ to the system above gives $\delta(k(x),4) \longrightarrow g(\delta(x,4),\delta(x,4))$, hence a "fork" which took place too early. Such behavior can be avoided by requiring that the variable x in the rule $\{\delta(f(x),2) \longrightarrow g(\delta(x,2),\delta(x,2))$ has a "non-δ-sort" (see Section 5.5).

assumed to be a timed rewrite *rule* in the specification.

$$age_1: \ \frac{}{t \xrightarrow{r} age(t,r)} \qquad age_2: \ \frac{}{age(t,r) \xrightarrow{r'} t'}$$

The *age* operator also satisfies the axiom $age(age(t,r),r') = age(t,r+r')$ for all terms t and time values r, r'.

With aging, the "fork" differences disappear, since (assuming $g(x,y) \xrightarrow{0} g(x,y)$) we have $f(a) \xrightarrow{2} g(age(a,2), age(a,2)) \xrightarrow{0} g(b,c)$ for the system in the example of Section 5.3.2, and the strong synchronicity is loosened, as illustrated by the fact that in Section 5.3.3, $f(a,b) \xrightarrow{6} k(d,c)$ is derivable, since $f(a,b) \xrightarrow{2} g(a, age(b,2))$, $g(a, age(b,2)) \xrightarrow{2} h(age(a,2),c)$, and $h(age(a,2),c) \xrightarrow{2} k(d,c)$ are derivable.

Unfortunately, the deduction rules for aging lead to counterintuitive results, as illustrated by the following example:

Example 5.4 Given the TRL theory $\{f(x) \xrightarrow{2} g(x), f(b) \xrightarrow{2} g(c), a \xrightarrow{2} b\}$, one would expect that $f(a) \xrightarrow{2} g(c)$ is *not* derivable. However, $f(x) \xrightarrow{2} age(f(x),2)$ and $age(f(x),2) \xrightarrow{0} g(x)$ are derivable, and so are $f(b) \xrightarrow{2} age(f(b),2)$ and $age(f(b),2) \xrightarrow{0} g(c)$.

The sequents $f(x) \xrightarrow{2} age(f(x),2)$ and $a \xrightarrow{2} b$ give $f(a) \xrightarrow{2} age(f(b),2)$ by synchronous replacement, which in turn rewrites to $g(c)$ using $age(f(b),2) \xrightarrow{0} g(c)$. Transitivity gives the undesired sequent $f(a) \xrightarrow{2} g(c)$.

We can summarize the situation as follows. We have seen that the rewriting translation of a TRL theory $\mathcal{T}$ is looser than $\mathcal{T}$ itself, in some cases with some pleasant consequences. If we attempt to tighten the correspondence between both systems by adding aging rules to TRL, we get indeed closer, but we unfortunately encounter paradoxical examples in the reformulation of TRL.

5.5 Simulation of TRL Theories in Rewriting Logic

In spite of the above mentioned discrepancies between the two formal systems, in practice there are interesting examples for which a TRL theory $\mathcal{T}$ and its translation $\mathcal{M}(\mathcal{T})$ behave in exactly similar ways for ground terms, in the sense that given $t, t' \in T_\Sigma$, $r \in T_{\Sigma_{Time}}$ we have

$$\mathcal{T} \vdash t \xrightarrow{r} t' \iff \mathcal{M}(\mathcal{T}) \vdash \mathcal{M}(t \xrightarrow{r} t').$$

If we find ourselves in such a good situation, it becomes interesting to use rewriting logic not only to mirror the deduction of the TRL theory $\mathcal{T}$, but also for simulation purposes, so as to observe the behavior of the real-time system specified by $\mathcal{T}$. This can be done using ideas similar to those in Section 2.3. That is, we can extend $\mathcal{M}(\mathcal{T})$ by adding an appropriate global sort and tick rules to a rewrite theory, denoted $\mathcal{T}^t$, that will allow us to simulate the system specified by $\mathcal{T}$. However, instead of having tick rules of the form

$$\widehat{x} \longrightarrow \widehat{\delta(x,t_r)},$$

as in Section 2.3, the tick rules will now be of the form

$$\widehat{x} \longrightarrow \widehat{y} \ \textbf{if} \ \delta(x,x_r) \longrightarrow y$$

and terms will be given sorts in a way such that only terms not containing δ can be substituted for y. In this way, exactly all possible TRL-rewrites can be deduced, while we in a simulation trace never will see a term containing the symbol δ (they will be hidden in the conditions allowing each rewrite).

Formally, given an order-sorted TRL theory $\mathcal{T} = \langle\langle S, \leq, \Sigma\rangle, E, R\rangle$, the translation $\mathcal{T}^t$ is defined as follows:

$$\begin{aligned}
S^t &= S \cup \{s^t \mid s \in S\} \cup \{whole_system\} \\
\leq^t &= \text{the reflexive-transitive closure of } \leq \cup \{s \leq s^t \mid s \in S\} \\
\Sigma^t &= \Sigma \cup \{f : s_1^t, \ldots, s_n^t \to s^t \mid f : s_1, \ldots, s_n \to s \in \Sigma \wedge n \geq 1\} \\
&\quad \cup \{\delta : s^t, Time \to s^t \mid s \in S\} \\
&\quad \cup \{\widehat{\ } : s^t \to whole_system \mid s \in S\}.
\end{aligned}$$

Since the sorts in S are only used to know which terms do not contain δ symbols, the variables in the equations and rules should be of sorts s^t ($s \in S$), hence

$$\begin{aligned}
E^t = \{&(\forall x_1 : s_1^t, \ldots, x_n : s_n^t)\ \ t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)\ \mid \\
&(\forall x_1 : s_1, \ldots, x_n : s_n)\ \ t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n) \in E\} \\
\bigcup\{&(\forall x_r, y_r : Time, x : s^t)\ \ \delta(\delta(x, x_r), y_r) = \delta(x, x_r + y_r),\ \delta(x, 0) = x \mid s \in S\}.
\end{aligned}$$

The set R^t of rules contains the translation of the TRL rules (but relaxing each sort s to s^t),

$$\begin{aligned}
\{&(\forall x_1 : s_1^t, \ldots, x_n : s_n^t)\ \ \delta(t(x_1, \ldots, x_n), r) \longrightarrow t'(\delta(x_1, r)/x_1, \ldots, \delta(x_n, r)/x_n)\ \mid \\
&(\forall x_1 : s_1, \ldots, x_n : s_n)\ \ t(x_1, \ldots, x_n) \xrightarrow{r} t'(x_1, \ldots, x_n) \in R\}
\end{aligned}$$

plus the tick rules

$$\{(\forall x : s, y : s', x_r : Time)\ \ \widehat{x} \longrightarrow \widehat{y}\ \textbf{if}\ \delta(x, x_r) \longrightarrow y \mid s, s' \in S\}.$$

It is easy to show that

$$\mathcal{M}(\mathcal{T}) \vdash \delta(t, r) \longrightarrow t'(\delta(x_1, r)/x_1, \ldots, \delta(x_n, r)/x_n)$$

iff

$$\mathcal{T}^t \vdash \delta(t, r) \longrightarrow t'(\delta(x_1, r)/x_1, \ldots, \delta(x_n, r)/x_n)$$

since $\mathcal{T}^t$ and $\mathcal{M}(\mathcal{T})$ only differ in the sorts of terms containing δ, the sort s^t in $\mathcal{T}^t$ contains the same set of terms as the sort s in $\mathcal{M}(\mathcal{T})$, and all variables of sort s in $\mathcal{M}(\mathcal{T})$ have sort s^t in the rules and equations of $\mathcal{T}^t$. Therefore,

$$\forall r \in T_{\Sigma_{Time}}, \forall t, t' \in T_\Sigma,\ \ \mathcal{T} \vdash t \xrightarrow{r} t' \iff \mathcal{M}(\mathcal{T}) \vdash \mathcal{M}(t \xrightarrow{r} t')$$

implies

$$\forall r \in T_{\Sigma_{Time}}, \forall t, t' \in T_\Sigma,\ \ \mathcal{T} \vdash t \xrightarrow{r} t' \iff \mathcal{T}^t \vdash \delta(r, t) \longrightarrow t'.$$

It can then be proved that

- a system $\widehat{t}$ rewrites to a non-δ-term $\widehat{t'}$ in $\mathcal{T}^t$ iff in TRL t rewrites to t' in $\mathcal{T}$:

$$\forall t, t' \in T_\Sigma,\ \ (\exists r \in T_{\Sigma_{Time}} \mid \mathcal{T} \vdash t \xrightarrow{r} t') \iff \mathcal{T}^t \vdash \widehat{t} \longrightarrow \widehat{t'}$$

- in this translation, no system rewrites to a δ-term:

$$\forall t \in T_\Sigma, \forall t' \in T_{\Sigma^t},\ \ \mathcal{T}^t \vdash t \longrightarrow t' \ \text{ implies } \ t' \in T_\Sigma.$$

The tick rule introduces two variables, x_r and y. This reflects the fact that it is in general undecidable in TRL whether a term rewrites in time r $(r > 0)$, and, even if it is known that t rewrites in time r, it is also in general undecidable whether t rewrites to a given term t' in time r.

6 Concluding Remarks

We have presented a general method for specifying real-time and hybrid systems in rewriting logic, have illustrated it with a well-known benchmark, and have shown how a wide range of real-time and hybrid system models can be naturally expressed in rewriting logic.

The present work should be further extended in several directions, including the following:

- systematic study of the relationships with more abstract levels of specification such as the duration calculus and timed temporal logics;
- study of notions of *distributed real-time* within the rewriting logic formalism;
- further exploration of the use of rewriting strategies in real-time specifications;
- further development of proof techniques and tools, and of symbolic simulation capabilities, within rewriting logic itself;
- development of a good collection of case studies and executable specifications, and further exploration of how other real-time formalisms can be expressed.

Acknowledgments. We cordially thank Saddek Bensalem, Manuel Clavel, Piotr Kosiuczenko, Narciso Martí-Oliet, Sigurd Meldal, Joseph Sifakis, Carolyn Talcott, and Martin Wirsing for their comments and suggestions, that have helped us in the development of these ideas and in improving their presentation.

References

[1] W. M. P. van der Aalst. Interval timed coloured Petri nets and their analysis. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 453–472, 1993.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[3] Rajeev Alur and David Dill. The theory of timed automata. In J.W. de Bakker, G. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, 1991.

[4] Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer, 1985.

[5] J. A. Bergstra and J. V. Tucker. Algebraic specification of computable and semicomputable data types. *Theoretical Computer Science*, 50:137–181, 1987.

[6] Manuel G. Clavel and José Meseguer. Reflection and strategies in rewriting logic. 1996. In this volume.

[7] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992. Originally given as lecture at *Seminar on Types*, Carnegie-Mellon University, June 1983; several draft and technical report versions were circulated since 1985.

[8] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proc., IEEE Real-Time System Symposium, San Juan*, 1994.

[9] P. Kosiuczenko and M. Wirsing. Timed rewriting logic, 1995. Working material for the 1995 Marktoberdorf International Summer School "Logic of Computation".

[10] P. Kosiuczenko and M. Wirsing. Timed rewriting logic for the specification of time-sensitive systems. *Science of Computer Programming*, 1996. To appear.

[11] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*, 1991.

[12] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.

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

[14] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Concur'96*, 1996. To appear.

[15] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

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

[17] S. Morasca, M. Pezzè, and M. Trubian. Timed high-level nets. *The Journal of Real-Time Systems*, 3:165–189, 1991.

[18] P. C. Ölveczky, P. Kosiuczenko, and M. Wirsing. An object-oriented algebraic steam-boiler control specification. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Application: Specifying and Programming the Steam-Boiler Control*. Springer, 1996. To appear.

[19] Michael Rabin. Computable algebra: General theory and theory of computable fields. *Transactions of the American Mathematical Society*, 95:341–360, 1960.