Specification of real-time and hybrid systems in rewriting logic \star

Peter Csaba Ölveczky^{a,b,1} and José Meseguer^a

^a Computer Science Laboratory, SRI International, Menlo Park, USA ^b Dept. of Informatics, University of Bergen, Norway

Abstract

This paper explores the application of rewriting logic to the executable formal modeling of real-time and hybrid systems. We give general techniques by which such systems can be specified as ordinary rewrite theories, and show that a wide range of real-time and hybrid system models, including object-oriented systems, timed automata [4], hybrid automata [2], timed and phase transition systems [28], and timed extensions of Petri nets [1,37], can indeed be expressed in rewriting logic quite naturally and directly. Since rewriting logic is executable and is supported by several language implementations, our approach complements property-oriented methods and tools less well suited for execution purposes. The relationships with the timed rewriting logic approach of Kosiuczenko and Wirsing [24,25] are also studied.

1 Introduction

This paper explores the application of rewriting logic to the *executable formal* modeling of real-time and hybrid systems. The general conceptual advantage of using a logic instead of using a specific model is that many different models can be specified in the same logic, each in its own terms, rather than by means of possibly awkward translations into a fixed model. The advantages of using rewriting logic as a semantic framework for concurrency models has been amply demonstrated (see the surveys [34,35]). This work shows that a

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

¹ Supported by The Norwegian Research Council.

number of well-known models of real-time and hybrid systems can likewise be naturally specified in the rewriting logic framework.

Thus, rewriting logic can be used to specify many different formal models of such systems in a unified logic. But, since rewriting logic is executable, and is supported by several language implementations [13,10,18], these models can be executed and can be formally analyzed in a variety of ways. This is in contrast to the most well-known formal methods tools for real-time and hybrid systems such as Kronos [17], STeP [9,30], and UPPAAL [7]. These are model checking tools which require the user to specify both the system and the formal properties the system should satisfy. The tools then try to check whether the system satisfies a given abstract property. However, these tools are not well suited for directly executing the system itself. The same can be said about HyTech [22], which takes a hybrid system description with some parameters unspecified, and returns the concrete values of the parameters which would make the system satisfy some given property. Of course, model checking tools have important strengths of their own. The point is that executable specification methods and tools can complement those strengths in new ways.

To see how rewriting logic complements more abstract specifications such as temporal logic as well as more concrete, automaton-based ones, one can think of it as covering an *intermediate level*, that can substantially help in bridging the gap between more abstract, property-oriented, specifications and actual implementations by providing:

- a precise mathematical model of the system (the initial model [32]) against which more abstract specifications can be proved correct by means of inductive theorem proving, model checking, and other techniques;
- support for other useful techniques of automated or semi-automated formal reasoning and analysis at the rewriting logic and equational logic levels, such as coherence [44], confluence, and strategy-based formal analysis;
- support for executable specification, rapid prototyping, and symbolic simulation;
- the possibility of generating correct implementations from specifications by theory transformations and code generation techniques.

We show that ordinary rewrite theories are sufficient to specify real-time systems in a natural way. Essentially, all we need is to include in the specification a *Time* data type satisfying appropriate equational properties. However, it is sometimes useful to highlight the real-time aspect by making explicit the duration information for some rewrite rules. We formalize this idea in Section 2 by means of *real-time rewrite theories*; but we show that, by adding an explicit clock, they are reducible to ordinary rewrite theories in a way that preserves all the expected properties. The naturalness of the specification method, and its smooth integration with rewriting logic's support for object-oriented specification, is explored and illustrated with examples in Section 3, in which we also address the question of how generally and naturally rewriting logic can be used to express a variety of real-time and hybrid system models. We show in detail how, besides objectoriented real-time systems, a wide range of such models, including timed automata [4], hybrid automata [2], timed and phase transition systems [28], and timed extensions of Petri nets [1,37], can indeed be expressed in rewriting logic quite naturally and directly.

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) [24,25], 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 as the steam-boiler [38], and has also been applied to give semantics to the SDL telecommunications specification language [42]. In fact, rewriting logic object-oriented specifications in the Maude language [33] have a natural extension to TRL object-oriented specifications in *Timed Maude* [24,38].

The approach taken here is different. As already mentioned, we argue that real-time systems can be specified in ordinary rewriting logic, and that reasoning about their behavior does not require a special inference system of their own, such as the one proposed in TRL. Even when special notation highlighting real-time aspects—such as that provided by real-time rewrite theories—is used, we show that this can essentially be regarded as syntactic sugar. This has the conceptual advantage of remaining within a simpler theoretical framework, and the practical advantage of being able to use the existing language implementations of rewriting logic to execute specifications. Therefore, it seems both conceptually and practically useful to study the relationships between our approach and TRL. We do so in Section 4, where we show that there is a map of logics $\mathcal{M}: TRL \longrightarrow RWL$ sending each TRL specification to a corresponding rewrite theory 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.

1.1 Prerequisites on rewriting logic and Maude

We assume familiarity with the basic concepts of rewriting logic as presented in [32,35]. We recall here only the most basic notions that we shall use. Rewriting logic specifications are *rewrite theories* of the form $\mathcal{R} = (\Sigma, E, L, R)$, where (Σ, E) is an equational theory, L is a set of labels, and R is a collection of

labeled rewrite rules of the form

$$[l]: t \longrightarrow t' \text{ if } \bigwedge_{i=1}^n u_i \longrightarrow v_i \land \bigwedge_{j=1}^m w_j = w'_j,$$

with $l \in L$, which is implicitly universally quantified by the variables appearing in the Σ -terms t, t', u_i, v_i, w_j , and w'_j . In this paper the equational theory (Σ, E) will always be assumed to be order-sorted [19]. That is, the set of sorts comes equipped with a partial order relation, with $s \leq s'$ interpreted as subset inclusion $A_s \subseteq A_{s'}$ in a model A. Furthermore, operation symbols can be subsort overloaded (as for example the addition symbol + for naturals, integers, and rationals). Such overloaded operators are required to yield the same result for the same arguments, regardless of the overloaded operator that is applied.

We make frequent use of the *initial model* construction $\mathcal{T}_{\mathcal{R}}$ associated to a rewrite theory \mathcal{R} , in which rewrite proofs $\alpha : t \longrightarrow t'$, derivable from the rules in \mathcal{R} using the rules of deduction of rewriting logic, are equated modulo a natural notion of *proof equivalence* [32,35]. However, $\mathcal{T}_{\mathcal{R}}$ has to be understood in an order-sorted sense, so that for each sort s we have an associated category $(\mathcal{T}_{\mathcal{R}})_s$, with arrows $\alpha : t \longrightarrow t'$ equivalence classes of proofs with t, t' ground terms of sort s, and with arrow composition corresponding to application of the transitivity rule.

Throughout the text we often use Maude-like notation [13] to present specific rewrite theories. For the most part this notation is self-explanatory. In the case of object-oriented modules, we explain their syntax and basic assumptions in Section 3.5.

2 Time models and real-time rewrite theories

After specifying equationally the general requirements for the models of time that we will consider (Section 2.1) we propose a general notion of *real-time rewrite theory*, consisting of an ordinary rewrite theory, where rewrite rules affecting the whole system have associated time-duration expressions (Section 2.2). We then show that real-time rewrite theories form a category (Section 2.3) and that they can be reduced to *ordinary* rewrite theories by adding an explicit clock to the global state in a way that preserves all the expected properties (Section 2.4). We finish the section with a discussion of several issues and specification techniques for real-time rewrite theories (Section 2.5). Time is modeled abstractly by a commutative monoid (Time, +, 0) with additional operators \leq , <, and \div ("monus") satisfying the following Maude theory.

```
fth TIME is
protecting BOOL
    sort Time
    op 0: \rightarrow Time
          -+: Time Time \rightarrow Time [assoc comm id:0]
    op
    ops \_ < \_, \_ \le \_: Time Time \rightarrow Bool
    op \_ \doteq \_: 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
          (x_r + y_r) \div y_r = x_r
    eq
    ceq x_r \div y_r = 0 if not(y_r \le x_r)
    eq
          x_r \leq x_r + y_r = true
    eq
         (x_r < x_r) = false
    eq (x_r \le y_r) = (x_r < y_r) or (x_r == y_r)
    ceq x_r + y_r \leq z_r + w_r = true if x_r \leq z_r and y_r \leq w_r
    ceq (x_r - y_r) + y_r = x_r if y_r \leq x_r
endft
```

In this theory, it can for example 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 $x_r \div y_r$) 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 [8], 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 [19], 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 [40], 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 [6].

We will in some examples in this paper need to extend the time domain with a new value ∞ and/or to require that the time domain is linear. The following

theory gives an abstract specification of the time domain extended with a value ∞ .

```
fth TIME_{\infty} is
including TIME
                   Time_{\infty}
      \mathbf{sort}
      subsort Time \leq Time_{\infty}
                   \infty: \rightarrow Time_{\infty}
      op
                   \_ \leq \_: Time_{\infty} Time_{\infty} \rightarrow Bool
      op
                   \_ \doteq \_: Time_{\infty} \ Time_{\infty} \rightarrow Time_{\infty}
      op
                   \_+\_: Time_{\infty} Time_{\infty} \rightarrow Time_{\infty} [assoc \ comm \ id:0]
      op
                   x_r: Time
      var
                   x_r \leq \infty = true
      eq
endft
```

Linear time can be specified by the following theory:

```
fth LTIME is

including 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 \le x_r

endft
```

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

```
fth LTIME_{\infty} is

including LTIME, TIME_{\infty}

op min : Time_{\infty} Time_{\infty} \rightarrow Time_{\infty} [comm]

var x_r : Time_{\infty}

eq min(\infty, x_r) = x_r

endft
```

Notation: We will use symbols r, r', r_1, \ldots to denote time values and x_r, y_r, \ldots to denote variables of the sort of the time domain.

2.2 Real-time rewrite theories

After recalling the notion of a theory morphism between equational theories, we define real-time rewrite theories; they are used to specify real-time systems in rewriting logic and contain duration information for some rules. Rules are divided into *tick rules*, that model the elapse of time on a system, and *instantaneous rules*, that model change that can be approximated to take zero time. Having a tick rule $t \longrightarrow t'$ could lead to rewrites $f(t, u) \longrightarrow f(t', u)$, i.e., rewrites where time only elapses in a part of the system under consideration. To ensure uniform time elapse we introduce a new sort *System*, with no subsorts, and a free constructor $\{ _ \} : State \longrightarrow System$ with the intended meaning that $\{t\}$ denotes the whole system, which is in state t. Uniform time elapse is ensured if the global state always has the form $\{t\}$ and every tick rule is of the form $\{t\} \longrightarrow \{t'\}$.

Definition 1 An equational theory morphism $H : (\Sigma, E) \to (\Sigma', E')$ consists of a map $H : sorts(\Sigma) \to sorts(\Sigma')$, and a mapping sending each function $symbol^2 f : s_1 \ldots s_n \to s$ in Σ to a Σ' -term H(f) of sort H(s), such that its set of variables is contained in the set $x_1 : H(s_1), \ldots, x_n : H(s_n)$, and such that for each axiom $(\forall y_1 : s_1, \ldots, y_k : s_k) \ l = r$ if C in E,

$$E' \models (\forall y_1 : H(s_1), \dots, y_k : H(s_k)) \ H^*(l) = H^*(r) \ \text{if} \ H^*(C)$$

holds, for H^* the straightforward extension of H to terms and to equations in the condition C.

Definition 2 A real-time rewrite theory $\mathcal{R}_{\phi,\tau}$ is a tuple $(\mathcal{R}, \phi, \tau)$, where $\mathcal{R} = (\Sigma, E, L, R)$ is a rewrite theory, such that³:

- ϕ is an equational theory morphism ϕ : TIME $\rightarrow (\Sigma, E)$ where TIME is the theory defined in Section 2.1,
- the time domain is functional; that is, whenever $\alpha : r \longrightarrow r'$ is a rewrite proof in \mathcal{R} and r is a term of sort $\phi(Time)$, then α is equivalent to the identity proof r,
- (Σ, E) contains a designated sort that we usually call State and a specific sort System with no subsorts or supersorts and with only one operator

$$\{ _ \} : State \rightarrow System$$

which satisfies no non-trivial equations, and

• τ is an assignment of a term $\tau_l(x_1, \ldots, x_n)$ of sort $\phi(Time)$ to each rewrite rule in \mathcal{R} of the form

(†)
$$[l]: u(x_1,\ldots,x_n) \longrightarrow u'(x_1,\ldots,x_n)$$
 if $C(x_1,\ldots,x_n)$

where u and u' are terms of sort System.

² Since the variables x_1, \ldots, x_n are ordered, the assignment $f \mapsto H(f)$ can alternatively be understood as an assignment $f(x_1, \ldots, x_n) \mapsto H(f)$.

³ We give a definition based on loose semantics of rewrite theories. Real-time rewrite theories can be defined in a similar way in an initial semantics setting.

Notation: We will write

$$[l]: u(x_1,\ldots,x_n) \xrightarrow{\tau_l(x_1,\ldots,x_n)} u'(x_1,\ldots,x_n)$$
if $C(x_1,\ldots,x_n)$

for a rule l of sort *System* with duration τ_l . If $\tau_l(x_1, \ldots, x_n)$ equals $\phi(0)$, the rule l will often be written

$$[l]: u(x_1,\ldots,x_n) \longrightarrow u'(x_1,\ldots,x_n)$$
 if $C(x_1,\ldots,x_n)$.

We will also write $Time_{\phi}$, 0_{ϕ} , and $+_{\phi}$ for, respectively, $\phi(Time)$, $\phi(0)$, and $\phi(+)$.

We call rules of the form (\dagger) global rules. A global rule l is a tick rule if its duration $\tau_l(x_1, \ldots, x_n)$ is different from 0_{ϕ} for some instances of its variables, and is an *instantaneous rule* otherwise. The rules not of the form (\dagger) are called *local* rules, because they do not act on the system as a whole, but only on some system components. They are always instantaneous. Intuitively, instantaneous rules take zero time.

The total time elapse $\tau(\alpha)$ of a rewrite $\alpha : \{t\} \longrightarrow \{t'\}$ of sort *System* is defined as the sum of the time elapsed in each tick rule application in α , and can easily be extracted from the proof:

Definition 3 Let $(\mathcal{R}, \phi, \tau)$ be a real-time rewrite theory with $\mathcal{R} = (\Sigma, E, L, R)$ and let Time denote the time domain $(T_{\Sigma,E})_{Time_{\phi}}$ viewed as a monoid and therefore as a category with a single object 0, and with the time values as arrows in the usual way. The time extraction functor

$$\tau: \mathcal{T}_{\mathcal{R}_{System}} \to Time$$

which gives the total time elapse $\tau(\alpha)$ of a proof $\alpha : t \longrightarrow t'$, with t, t' ground terms of sort System, is defined as follows:

- $\tau(t) = 0_{\phi}$ for every term (seen as an arrow) in $\mathcal{T}_{\mathcal{R}_{Sustem}}$,
- $\tau(\{\alpha\}) = 0_{\phi}$ for a proof term whose top operator is the constructor $\{ _ \}$,
- $\tau(l(\alpha_1,\ldots,\alpha_n)) = \tau_l(t_1,\ldots,t_n)$ if l is a (system) rule of the form (\dagger) and $\alpha_1: t_1 \longrightarrow t'_1, \ldots, \alpha_n: t_n \longrightarrow t'_n$ are proofs, and
- $\tau(\alpha; \beta) = \tau(\alpha) +_{\phi} \tau(\beta)$ for proofs α and β .

This definition does not depend on the choice of representative proof terms. That is, if α and β are two equivalent proofs of terms of sort *System* in a real-time rewrite theory $(\mathcal{R}, \phi, \tau)$, then $\tau(\alpha) = \tau(\beta)$.

Given a real-time rewrite theory \mathcal{R} , a *computation* is a non-extensible sequence $t_0 \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$ or an infinite sequence $t_0 \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$ of one-step \mathcal{R} -

rewrites $t_i \longrightarrow t_{i+1}$, with t_i and t'_i ground terms, starting with a given initial term t_0 of sort *System*. It should be noted that since we model time elapse *explicitly* (by rewrite rules), the requirement that the total time elapse in a computation is infinite is not needed. Time elapse is totally up to the specifier – we allow both terminating computations and infinite computations with finite total time elapse.

2.3 A category of real-time rewrite theories

The notion of theory morphism – also called theory interpretation – between real-time rewrite theories plays an important role in this work. We give a definition of theory morphism between real-time rewrite theories based on loose semantics and preservation of durations of rewrites. Morphisms based on properties of the initial models of theories, and morphisms having less restrictive requirements on the relationships between the durations in the rewrites could be defined in a similar way. We begin by defining theory morphisms between ordinary rewrite theories.

Definition 4 A rewrite theory morphism from a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ to another rewrite theory $\mathcal{R}' = (\Sigma', E', L', R')$ consists of:

- an equational theory morphism $H : (\Sigma, E) \to (\Sigma', E')$, and
- a map $H: L \longrightarrow L'$ of labels such that for each rule $[l]: t \longrightarrow t'$ if C in R the rule

$$[H(l)]: H^*(t) \longrightarrow H^*(t') \text{ if } H^*(C)$$

is in R' up to a renaming of its variables.

Rewrite theory morphisms compose in the expected way and define a category RWTh of rewrite theories.

Definition 5 A real-time rewrite theory morphism from a real-time rewrite theory $(\mathcal{R}, \phi, \tau)$ to a real-time rewrite theory $(\mathcal{R}', \phi', \tau')$ is a rewrite theory morphism $H : \mathcal{R} \to \mathcal{R}'$ such that:

- $\phi' = H \circ \phi$,
- H maps the designated sort of the states in \mathcal{R} to the designated state sort in \mathcal{R}' , maps the sort System to itself, and leaves the constructor $\{_\}$ unchanged, and
- *H* preserves the duration of the tick rules in the sense that for each rule *l* in *R* of sort System,

$$E' \vdash H^*(\tau_l(x_1:s_1,\ldots,x_n:s_n)) = \tau'_{H(l)}(x_1:H(s_1),\ldots,x_n:H(s_n)).$$

It is easy to check that the usual composition of rewrite theory morphisms defines a category RTRWTh with real-time rewrite theories as objects and real-time rewrite theory morphisms as arrows.

Real-time theories internalized in rewriting logic 2.4

By adding a clock to the state, a real-time theory $(\mathcal{R}, \phi, \tau)$ can be transformed into an ordinary rewrite theory without losing timing information. A state in such a clocked system is of the form $\langle t, r \rangle$ with t the global state of sort System, and r a value of sort $Time_{\phi}$, which intuitively is supposed to denote the total time elapse in a computation if in the initial state the clock had value 0_{ϕ} .

Definition 6 The internalizing functor $(_)^C$ from the category RTRWTh of real-time rewrite theories to the category RWTh of rewrite theories takes a realtime rewrite theory $(\mathcal{R}, \phi, \tau)$ to a rewrite theory $\mathcal{R}^{C}_{\phi, \tau} = (\Sigma^{C}_{\phi, \tau}, E^{C}_{\phi, \tau}, L^{C}_{\phi, \tau}, R^{C}_{\phi, \tau})$ as follows:

- the sorts in Σ^C_{φ,τ} are those in R together with a new sort ClockedSystem,
 the operations in Σ^C_{φ,τ} are those in R together with a new free constructor

 $\langle _, _ \rangle$: System Time_{\phi} \rightarrow ClockedSystem,

- the axioms in $E_{\phi,\tau}^{C}$ are unchanged from those in \mathcal{R} ,
- $R^{C}_{\phi,\tau}$ contains the local rules in \mathcal{R} of sorts other than System, together with $a rule^4$

$$[l^{C}_{\phi,\tau}(x_{1},\ldots,x_{n},x_{r})]:\langle u(x_{1},\ldots,x_{n}),x_{r}\rangle \longrightarrow$$
$$\langle u'(x_{1},\ldots,x_{n}),x_{r}+_{\phi}\tau_{r}(x_{1},\ldots,x_{n})\rangle \text{ if } C(x_{1},\ldots,x_{n})$$

for each rule

$$[l(x_1,\ldots,x_n)]: u(x_1,\ldots,x_n) \longrightarrow u'(x_1,\ldots,x_n)$$
 if $C(x_1,\ldots,x_n)$

in \mathcal{R} of sort System, where x_r is a variable of sort Time_{ϕ} which is not in the list x_1, \ldots, x_n .

The internalizing functor is defined as expected on arrows in RTRWTh; i.e., an arrow H in RTRWTh is mapped to H^{C} , which coincides with H on \mathcal{R} ,

 $^{^4}$ In the unlikely case that any condition C of a rule in $\mathcal R$ contains a conjunct $v \rightarrow v'$ of sort System, each such conjunct is replaced by a conjunct $\langle v, 0_{\phi} \rangle \longrightarrow \langle v', y_r \rangle$ in the condition in $\mathcal{R}^{C}_{\phi,\tau}$, where y_r is a fresh variable of sort $Time_{\phi}$.

leaves the new sort and operator unchanged, and takes a label $l_{\phi,\tau}^c$ of a rule of sort ClockedSystem to the label $(H(l))^C$.

For the sake of a simpler exposition, in the rest of the paper we will assume that no condition of a rewrite rule in a real-time theory contains a rewrite conjunct of sort *System*. We also assume, without loss of generality, that no variable of sort *System* is introduced in the condition of a rule.

Proposition 7 The mapping $(_)^C$ above defines a functor from RTRWTh to RWTh.

Proposition 8 Let U be the forgetful functor from the category RTRWTh of real-time rewrite theories to the category RWTh of rewrite theories defined by

 $U((\mathcal{R}, \phi, \tau)) = \mathcal{R} \text{ and } U(H) = H.$

Then the map of rewrite theories $\pi_{(\mathcal{R},\phi,\tau)}: \mathcal{R}_{\phi,\tau}^C \to \mathcal{R}$ defined by:

- mapping each sort and operator in $\Sigma_{\phi,\tau}^C$ other than ClockedSystem and $\langle _, _ \rangle$ identically to themselves,
- mapping ClockedSystem to System, mapping the operator $\langle _, _ \rangle$ to the term x_1 :System, and
- mapping each label $l_{\phi,\tau}^C$ to the label l

defines a natural transformation $\pi : (_)^C \Rightarrow U$.

Since a rewrite theory morphism $H: \mathcal{R} \to \mathcal{R}'$ induces a forgetful functor U_H : $\mathcal{R}'-Sys \to \mathcal{R}-Sys$ in the opposite direction for the corresponding categories of models, our natural rewrite theory morphism $\pi: \mathcal{R}_{\phi,\tau}^C \to \mathcal{R}$ induces a forgetful functor $U_{\pi}: \mathcal{R}-Sys \to \mathcal{R}_{\phi,\tau}^C-Sys$. In particular, the initial model $\mathcal{T}_{\mathcal{R}}$ is sent to the $\mathcal{R}_{\phi,\tau}^C$ -system $U_{\pi}(\mathcal{T}_{\mathcal{R}})$ and, by initiality of $\mathcal{T}_{\mathcal{R}_{\phi,\tau}^C}$, we have a unique $\mathcal{R}_{\phi,\tau}^C$ homomorphism $\pi: \mathcal{T}_{\mathcal{R}_{\phi,\tau}^C} \to U_{\pi}(\mathcal{T}_{\mathcal{R}})$ such that:

- π takes objects and arrows of every sort except *ClockedSystem* to themselves,
- $\pi(\langle t, r \rangle) = t$ for each object $\langle t, r \rangle$ in $(\mathcal{T}_{\mathcal{R}^C_{+-}})_{ClockedSystem}$,

•
$$\pi(\langle \alpha, \beta \rangle : \langle t, r \rangle \longrightarrow \langle t', r' \rangle) = \alpha : t \longrightarrow t',$$

• $\pi(l_{\phi,\tau}^C(\alpha_1, \dots, \alpha_n, \alpha_{n+1}) : \langle t, r \rangle \longrightarrow \langle t', r' \rangle) = l(\alpha_1, \dots, \alpha_n) : t \longrightarrow t', \text{ and}$
• $\pi(\alpha; \beta) = \pi(\alpha); \pi(\beta).$

The map π expresses the essential semantic equivalence between the initial models of a real-time theory $(\mathcal{R}, \phi, \tau)$ and that of its clocked representation $\mathcal{R}_{\phi,\tau}^{C}$ in the precise sense that, as we shall see:

(1) if $\alpha : t \longrightarrow t'$ is an arrow in $\mathcal{T}_{\mathcal{R}_{System}}$ with $\tau(\alpha) = r$, then, for each value r' of sort $Time_{\phi}$ there is a unique arrow $\alpha' : \langle t, r' \rangle \longrightarrow \langle t', r' +_{\phi} r \rangle$ in

 $(\mathcal{T}_{\mathcal{R}_{\phi,\tau}^{C}})_{ClockedSystem}$ such that $\pi(\alpha': \langle t, r' \rangle \longrightarrow \langle t', r' +_{\phi} r \rangle) = \alpha : t \longrightarrow t'$, and

(2) whenever $\alpha : \langle t, r \rangle \longrightarrow \langle t', r' \rangle$ is an arrow in $(\mathcal{T}_{\mathcal{R}_{\phi,\tau}^{C}})_{ClockedSystem}$ then $r' = r +_{\phi} \tau(\pi(\alpha)).$

These two properties are immediate consequences of the following

Theorem 9 Let $(\mathcal{R}, \phi, \tau)$ be a real-time rewrite theory and let $\alpha : t \longrightarrow t'$ be an arrow in $\mathcal{T}_{\mathcal{R}_{System}}$ (therefore, with t and t' ground terms of sort System). Then, for each value r in the time domain, there is a unique arrow

$$\alpha':\langle t,r\rangle \longrightarrow \langle t',r'\rangle$$

in $(\mathcal{T}_{\mathcal{R}_{\phi,\tau}^{C}})_{ClockedSystem}$ such that $\pi(\alpha') = \alpha$, and, in addition, $r' = r +_{\phi} \tau(\alpha)$.

The theorem can be proved by induction on the structure of the proof terms by first proving the theorem for one-step rewrites, and then proving it for all proofs between terms of sort *System* using the facts that every proof factorizes into a sequence of one-step rewrites and that π distributes over one-step rewrite proofs.

The above theorem implies that, whenever $\alpha : \langle t, r \rangle \longrightarrow \langle t', r' \rangle$ is an arrow in $\mathcal{T}_{\mathcal{R}^{C}_{\phi,\tau}}$, then the arrow $\pi(\alpha) : t \longrightarrow t'$ satisfies $r +_{\phi} \tau(\pi(\alpha)) = r'$. It also implies that π , viewed as a functor $\pi : (\mathcal{T}_{\mathcal{R}^{C}_{\phi,\tau}})_{ClockedSystem} \to (\mathcal{T}_{\mathcal{R}})_{System}$, is full and faithful and is an opfibration [5].

2.5 Discussion

We discuss several system specification issues and techniques, including the time of local actions, tick rules, and rewrite strategies.

2.5.1 The time when local actions occur is generally underdetermined

For simulation purposes it may be desirable to observe the time at which an instantaneous local action takes place in a rewrite. However, an arrow in the initial model, that is, an equivalence class of proofs, does not give the exact time (relative to the initial state) when such a local action is applied. If, for example, $[l]: a \longrightarrow b$ and $[tick]: \{f(x, y_r)\} \xrightarrow{2} \{f(x, y_r + 2)\}$ are two rules, then the moment in time (relative to $\{f(a, 0)\}\}$) when the instantaneous action l took place in the rewrite $tick(l, 0): \{f(a, 0)\} \longrightarrow \{f(b, 2)\}$ of duration 2 is underdetermined. This is because, by the exchange law – that equates equivalent proofs in rewriting logic – this rewrite is equivalent to the rewrite

proofs $\{f(r,0)\}$; tick(b,0) and tick(a,0); $\{f(r,2)\}$, representing the rewrite sequences $\{f(a,0)\} \longrightarrow \{f(b,0)\} \longrightarrow \{f(b,2)\}$ and $\{f(a,0)\} \longrightarrow \{f(a,2)\}$ $\longrightarrow \{f(b,2)\}$, where the local action l takes place in a rewrite of duration 0 and either at time 0 or at time 2. By the sequentialization property of rewrite proofs [32], it is always possible to extract from a proof the possible relative times when a certain rule could have been applied in the proof.

2.5.2 Specifying the tick rules

For simulation of a system having a continuous time domain, the tick rules will in general be of the form

$$\{t\} \xrightarrow{x_r} \{t'(x_r)\}$$
 if $x_r \leq mte(t)$ and $C(t)$

or otherwise of the exact same form, but replacing $x_r \leq mte(t)$ by $x_r < mte(t)$, where x_r denotes the time advanced by the tick, mte(t) computes the maximum time elapse permissible to ensure timeliness of time-critical actions, and the condition $x_r \leq mte(t)$ (resp. $x_r < mte(t)$) ensures that time elapse may halt temporarily for the possible application of a non-time-critical rule, that is, a rule modeling an action which could occur somewhat "arbitrarily" in time. The introduction of the variable x_r in the righthand side requires additional execution strategies for its instantiations, which is not surprising, since it models behavior which is nondeterministic in time. Allowing for real nondeterminism in timed behavior in this way may lead to Zeno behavior of the system and it is up to the strategy to instantiate the righthand side variable so as to avoid that, whenever possible.

2.5.3 Eager and lazy rules

In general, it is not sufficient to ensure that time elapse "stops" whenever necessary. It must also be ensured that time does not tick past each stop before all the necessary instantaneous actions are performed. In particular, an application of a rule often enables a lot of other instantaneous rules that must be taken immediately, and it must be ensured that all these actions are performed before time elapses again. A rule may, for example, produce a message which must be consumed before time elapses again.

In many cases it is possible to add conditions on the tick rules such that time will not elapse if some time-critical rule is enabled, but this may considerably complicate the specification. Instead of computing the enabledness condition of every time-critical rule explicitly, it seems more convenient to use the rewriting logic notion of *internal rewrite strategy* [13,11,12,15,14], whose execution is well supported by Maude's reflective features, to deal with these enabledness and

priority aspects using a simple strategy.

The idea is to divide the rules in a real-time rewrite theory into *eager* and *lazy* rules and restrict possible rewrites by requiring that *lazy rules are applied* sequentially, and a lazy rule may only be applied when no eager rule is enabled. The intuition is that the eager rules are the time-critical rules that must always be taken when enabled, i. e., time may not elapse while an eager rule is enabled. Tick rules and non-time-critical instantaneous rules are lazy. Our treatment of timed Petri nets in Section 3.8 gives an example of the convenience of using this strategy.

Notation: Whenever an eager strategy should be used, the eager and lazy rules will be preceded by the keywords **eager** and **lazy**, respectively.

3 Specifying models of real-time and hybrid systems in rewriting logic

This section discusses how a variety of models of real-time and hybrid systems can be obtained as special cases of real-time rewriting.

3.1 Timed automata

We show how a timed automaton (see, e.g., [4,3]) can be specified in rewriting logic. Omitting details about initial states and acceptance conditions, a timed automaton 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

$$\phi ::= c \le k \mid k \le c \mid \neg \phi \mid \phi_1 \land \phi_2$$

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 run ρ of a timed automaton with n clocks is an infinite sequence

$$\rho: \{s_0, v_{0_1}, \dots, v_{0_n}\} \xrightarrow[\tau_1]{a_1} \{s_1, v_{1_1}, \dots, v_{1_n}\} \xrightarrow[\tau_2]{a_2} \{s_2, v_{2_1}, \dots, v_{2_n}\} \xrightarrow[\tau_3]{a_3} \cdots$$

for states s_0, s_1, \ldots , values v_{0_i}, v_{1_i}, \ldots of clock *i* such that $v_{0_j} = 0$ for all *j*, and such that for $i \ge 1$ there is a transition $\langle s_{i-1}, s_i, a_i, \lambda_i, \phi_i \rangle$ where the clock valuation $\langle v_{(i-1)_1} + \tau_i - \tau_{i-1}, \ldots, v_{(i-1)_n} + \tau_i - \tau_{i-1} \rangle$ satisfies the clock constraint ϕ_i , and v_{i_k} is 0 if clock *i* is in λ_i and $v_{i_k} = v_{(i-1)_k} + \tau_i - \tau_{i-1}$ otherwise.

A timed automaton can be naturally represented in rewriting logic as follows. The time domain and its associated constraints $\Phi(C)$ are equationally axiomatized in an abstract data type satisfying the theory *TIME*. The term $\{s, c_1, \ldots, c_n\}$ 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 expressed as an instantaneous rewrite rule

$$[a]: \{s, c_1, \ldots, c_n\} \longrightarrow \{s', c_1', \ldots, c_n'\} \text{ if } \phi(c_1, \ldots, c_n)$$

where $c'_i = 0$ if $c_i \in \lambda$, and $c'_i = c_i$ otherwise. In addition, a rule

$$[tick]: \{x, c_1, \dots, c_n\} \xrightarrow{x_r} \{x, c_1 + x_r, \dots, c_n + x_r\}$$

(where x, x_r, c_1, \ldots, c_n are all variables) is added to represent the elapse of time.

The rewriting logic translation simulates the timed automaton in the precise sense that there is a run ρ of the automaton as defined above if and only if there is a rewrite sequence

$$\{s_0, 0, \dots, 0\} \xrightarrow{\alpha_1} \{s_1, v_{1_1}, \dots, v_{1_n}\} \xrightarrow{\alpha_2} \{s_2, v_{2_1}, \dots, v_{2_n}\} \xrightarrow{\alpha_3} \cdots$$

such that α_i is equivalent to a proof term of the form β_i ; $a_i(\ldots)$, with β_i a (possibly empty) sequence of tick applications, and where $\tau(\beta_i) = \tau_i - \tau_{i-1}$ for τ_i, τ_{i-1} the corresponding time values in the run ρ .

There are at least two ways of modifying the specification to simulate the behavior of the automaton on only those timed words satisfying a given acceptance condition. It is possible to define a computable predicate $has_computation$, so that $has_computation(s, r_1, \ldots, r_n)$ holds if and only if there exist an *accepted* timed word "starting" in state *s* with (rational-numbered) values r_1, \ldots, r_n of the clocks c_1, \ldots, c_n (such a predicate is computable, and therefore finitely specifiable by Church-Rosser and terminating equations [8], since defining such a predicate reduces to the emptiness problem for timed automata which is decidable [4]). In this way, we obtain a rewrite theory whose computations simulate the behavior of the automaton on *accepted* timed words by adding the condition **if** $has_computation(x', c'_1, \ldots, c'_n)$ to every rule of the form $\{x, c_1, \ldots, c_n\} \longrightarrow \{x', c'_1, \ldots, c'_n\}$, including the tick rule.

A more modular, alternative way of restricting the rewrites to simulate automata behavior on accepted words only would be to encode the accepting states (or sets of states for Muller-automata) as predicates in the rewrite theory, and then use the internal strategies at the metalevel of rewriting logic to restrict the application of the rules, so that only accepted timed words are executed.

3.2 Hybrid Automata

The time model of hybrid automata [2] (also called just hybrid systems) is the nonnegative real numbers. However, to get a computable data type, we should replace the reals by a computable subfield \mathbb{R}_+ , such as the rationals or the algebraic real numbers. A hybrid automaton is given by a tuple $\langle V_D, Loc, Lab, Act, Inv, Edg \rangle$ where:

- V_D is a finite set of data variables, each ranging over a given data sort, defining the data space Σ_D , that is, Σ_D is the set of sort-consistent valuations \overline{v} of V_D .
- Loc is a finite set of *locations* (corresponding to "states" in untimed automata).
- The state space of a hybrid automaton is $Loc \times \Sigma_D$.
- Lab is a set of synchronization labels, including the label τ .
- 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 . For each activity f in l and each time value r there is an activity f^r in l defined by $f^r(r') = f(r+r')$.
- 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 $e = (l, \mu, l', a)$ consists of a source location l, a target location l', a transition relation $\mu \subseteq \Sigma_D^2$, and a synchronization label a. For each location l there is a stutter transition

$$(l, Id, l, \tau)$$
 where $Id = \{(\overline{v}, \overline{v}) \mid \overline{v} \in \Sigma_D\}.$

The state of an automaton 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, where state $\{l, \overline{v}\}$ evolves to $\{l, f(r)\}$ in time r whenever f is an activity of location l such that $\overline{v} = f(0)$. The system may stay at a location only if the invariant at that location remains true. The invariants of a hybrid automaton thus enforce the progress of the underlying discrete transition system. That is, some transition must be taken before the invariant of the location is false.

A run of a hybrid automaton is a finite or infinite sequence

$$\rho: \{l_0, \overline{v_0}\} \mapsto_{f_0}^{r_0} \{l_1, \overline{v_1}\} \mapsto_{f_1}^{r_1} \{l_2, \overline{v_2}\} \mapsto_{f_2}^{r_2} \cdots$$

where l_0, l_1, \ldots denote locations, $\overline{v_0}, \overline{v_1}, \ldots$ denote valuations of the variables V_D, r_0, r_1, \ldots denote time values, and f_0, f_1, \ldots denote activities in respective locations l_0, l_1, \ldots , and such that for all *i* it is the case that $f_i(0) = \overline{v_i}, f_i(r) \in Inv(l_i)$ for all $0 \leq r \leq r_i$, and that the state $\{l_{i+1}, \overline{v_{i+1}}\}$ is obtained by taking a transition from the state $\{l_i, f_i(r_i)\}$.

We specify hybrid automata in rewriting logic by representing a sort-consistent valuation $\overline{v} = \{x_1 \mapsto v_1, \ldots, x_m \mapsto v_m\}$ by a term $\langle v_1, \ldots, v_m \rangle$ in a sort *Valuation*, and by representing a global state $\{l, \overline{v}\}$ of a hybrid automaton by the term $\{l, \langle v_1, \ldots, v_m \rangle\}$ of sort *System*. However, since the definition of hybrid automata is very general, we restrict our treatment to a subclass of hybrid automata satisfying some natural requirements. Specifically, we require that the set of activities Act_l for a location l must be generated by a finite set

$$ActGen_l = \{f_i^l : \Sigma_D \times \mathbb{R}_+ \to \Sigma_D \mid 1 \le i \le n_l\}$$

of computable functions, called activity generators, where each f_i^l satisfies the property

$$f_i^l(f_i^l(\overline{v}, r), r') = f_i^l(\overline{v}, r + r') \text{ if } f_i^l(\overline{v}, 0) = \overline{v}.$$

Then, the set Act_l of activities for a location l is generated from $ActGen_l$ as follows,

$$Act_{l} = \{ f : \mathbb{R}_{+} \to \Sigma_{D} \mid (\exists f_{i}^{l} \in ActGen_{l}, \overline{v} \in \Sigma_{D}, r \in \mathbb{R}_{+}) \\ f_{i}^{l}(\overline{v}, 0) = \overline{v} \land f = \lambda x_{r} \cdot f_{i}^{l}(\overline{v}, r + x_{r}) \}.$$

Furthermore, we require that for each location l and activity generator $f_i^l \in ActGen_l$, there is a computable function

$$max_stay_{f_i^l}: \Sigma_D \to \mathbb{R}_+ \cup \{\infty\}$$

where $max_stay_{f_i^l}(\overline{v})$ denotes the amount of time a system in state $\langle l, \overline{v} \rangle$ can stay at location l performing the activity-function f_i^l , without violating the invariant of location l. We also require that there is a computable predicate

$$Inv: Loc \times \Sigma_D \to Bool$$

where $Inv(l, \overline{v})$ holds if and only if the state \overline{v} does not violate the invariant of location l. Finally, we require that each transition (l, μ, l', a) in Edg can be expressed by a finite number of rewrite rules of the form $[a] : \{l, \langle v_1, \ldots, v_n \rangle\} \longrightarrow \{l', \langle v'_1, \ldots, v'_n \rangle\}$ if C, with the v_i and v'_i possibly containing variables.

The class of hybrid automata satisfying the above requirements can be represented by real-time rewrite theories as follows. The functions f_i^l , $max_stay_{f_i^l}$, and Inv can be given a finitary equational axiomatization since they are assumed computable [8]. For each transition (l, μ, l', a) in the hybrid automaton, the translation of a hybrid automaton contains the corresponding rule(s)

$$[a]: \{l, \langle v_1, \dots, v_n \rangle\} \longrightarrow \{l', \langle v'_1, \dots, v'_n \rangle\} \text{ if } C \wedge Inv(l', \langle v'_1, \dots, v'_n \rangle) = true$$

where the last conjunct in the condition must be added to the translation of the (underlying "untimed") transition to ensure that the resulting state satisfies the invariant of location l'. The tick rules of the system associate to each location l and each activity generator f_i^l a rewrite rule of the form

$$[tick_i^l]: \{l, V\} \xrightarrow{x_r} \{l, f_i^l(V, x_r)\}$$
 if $x_r \leq max_stay_{f_i^l}(V)$ and $f_i^l(V, 0) = V$

where V is a variable of sort Valuation.

Note that eager/lazy strategies are not needed here, since a transition becomes "eager" in a hybrid automaton when max_stay of the location is 0, in which case time cannot advance. Due to the presence of idle transitions in hybrid automata, there is a run ρ as above in the automaton if and only if there is a computation

$$\{l_0, \overline{v_0}\} \xrightarrow{\alpha_0} \{l_1, \overline{v_1}\} \xrightarrow{\alpha_1} \{l_2, \overline{v_2}\} \xrightarrow{\alpha_2} \cdots$$

(of one-step rewrites) in the rewrite translation with $\tau(\alpha_i) = r_i$ for each *i*.

3.3 Timed transition systems

A timed transition system (TTS) [28,27] consists of a finite set of data variables defining the state space Σ_D of all sort-consistent valuations \overline{v} of the variables, and a finite number of transitions $a: \Sigma_D \to 2^{\Sigma_D}$. Each transition a is equipped with a "lower bound" l_a and an "upper bound" u_a where $0 \leq l_a \leq u_a \leq \infty$. A transition a cannot be taken if it has not been enabled uninterruptedly for at least time l_a , and if a is enabled at any time r, then a must be taken somewhere in the interval $[r, r + u_a]$, unless it is disabled during this time by some other transition.

Again, we assume that the underlying untimed transition system can be specified in rewriting logic, and that a valuation $\overline{v} = \{x_1 \mapsto v_1, \ldots, x_m \mapsto v_m\}$ is represented in rewriting logic by a tuple $\langle v_1, \ldots, v_m \rangle$ of sort Valuation. A TTS can then be represented in rewriting logic by just adding to each state one clock for each transition, such that the state in the rewriting translation is a term

$$\{\langle v_1,\ldots,v_m\rangle,c_1,\ldots,c_n\},\$$

where $\langle v_1, \ldots, v_m \rangle$ is the state of the transition system and each c_i is a "clock" value which is *nil* if transition a_i is not enabled, and is r_i if the transition a_i has been enabled continuously for time r_i (without being taken). The symbol *nil* is an element of a supersort of the sort *Time* of the time domain, satisfying the equation $nil + x_r = nil$ for $x_r : Time$. We also assume that for each transition a_i is enabled on state \overline{v} and false otherwise.

Assuming that each transition a_i in the underlying *untimed* transition system can be modeled by (zero or more) rewrite rules of the form $[a_i] : \langle v_1, \ldots, v_m \rangle \longrightarrow \langle v'_1, \ldots, v'_m \rangle$ if C, we model each such transition a_i in the *timed* system by the corresponding instantaneous rewrite rule(s)

$$[a_i]: \{\langle v_1, \ldots, v_m \rangle, c_1, \ldots, c_n\} \longrightarrow \{\langle v'_1, \ldots, v'_m \rangle, c'_1, \ldots, c'_n\} \text{ if } C \land (c_i \ge l_{a_i})$$

for all i = 1, ..., n, where for each j = 1, ..., n, c_j is a time variable, and

$$c'_j = \mathbf{if} \ not(enabled_j(\langle v'_1, \dots, v'_m \rangle)) \mathbf{then} \ nil$$

else $\mathbf{if} \ c_j == nil \ \mathrm{or} \ i == j \mathbf{then} \ 0 \mathbf{else} \ c_j$

The following tick rule ensures, for each transition a_i , that time will not elapse past the moment when a_i would have been enabled for time u_{a_i} without being taken:

$$[tick]: \{V, c_1, \dots, c_n\} \xrightarrow{x_r} \{V, c_1 + x_r, \dots, c_n + x_r\}$$

if $\bigwedge_i (c_i + x_r \leq u_{a_i} \text{ or } c_i == nil),$

again, for V, c_1, \ldots, c_n , and x_r variables of the appropriate sorts.

It is then easy to show that there is a computation

$$\{\overline{v_0}, 0, \ldots, 0\} \xrightarrow{\alpha_1} \{\overline{v_1}, r_{1_1}, \ldots, r_{1_n}\} \xrightarrow{\alpha_2} \{\overline{v_2}, r_{2_1}, \ldots, r_{2_n}\} \xrightarrow{\alpha_3} \cdots$$

of one-step rewrites α_i in the rewriting logic specification of a timed transition system if and only if there is a discrete trace [28]

$$\langle \overline{v_0}, 0 \rangle \rightarrow \langle \overline{v_1}, r_1 \rangle \rightarrow \langle \overline{v_2}, r_2 \rangle \rightarrow \cdots$$

(with possibly bounded total time elapse) of pairs of states $\overline{v_i}$ and time values r_i such that for all *i*, either $\overline{v_i} = \overline{v_{i+1}}$ or $\overline{v_{i+1}} \in a(\overline{v_i})$ holds for some transition *a* in the corresponding timed transition system which has been continuously enabled for at least its minimum delay l_a , and such that transitions are never continuously enabled for a time longer than their maximum time delay without being taken. Furthermore, $\tau(\alpha_i) = r_i - r_{i-1}$ (where $r_0 = 0$). Notice, that the implicit eagerness of a transition is due to its upper bound u_{a_i} , so that time will not elapse if such an eager transition does not fire. Therefore, there is no need for introducing explicit eager/lazy strategies.

3.4 Phase transition systems

Phase transition systems (PTSs) [28,27] extend timed transition systems to hybrid systems⁵. We give here only a brief overview of a representation of PTSs in rewriting logic. The reference [39] gives more details about the translation. Intuitively, the PTS model extends the TTS model by letting time act on each valuation according to a function

$$\delta: \Sigma_D \times \mathbb{R}_+ \to \Sigma_D$$

where $\delta(\overline{v}, r)$ denotes the state of the PTS after time has acted on a system in state \overline{v} for time r. The set \mathcal{T} of instantaneous transitions is, as in the TTS

 $^{^{5}}$ Note that some authors instead use the expression *phase transition system* for the hybrid systems extension of the *clocked transition system* [29,23] model.

case, equipped with upper and lower bounds. Furthermore, time cannot elapse past a moment when the enabling condition of a transition changes. Since the action of time can change the enabling of transitions, we assume that there is a computable function

enabling_change :
$$\Sigma_D \to \mathbb{R}_+ \cup \{\infty\}$$

which takes a state as argument and gives the maximum time the system can proceed without changing the enabling of a transition.

The global state and the (instantaneous) transitions in \mathcal{T} are modeled in rewriting logic as for the TTS case. That is, the global state has the form $\{\langle v_1, \ldots, v_m \rangle, c_1, \ldots, c_n\}$ with $\langle v_1, \ldots, v_m \rangle$ a valuation, and each c_i a time value denoting how long transition a_i has been continuously enabled (which is *nil* if a_i is not enabled). The functions δ and *enabling_change* are defined on terms of the sort *Valuation*. The following tick rule ensures, in addition to the TTS requirement, that time cannot elapse beyond the latest moment when a transition must be taken, that all state components are updated according to their continuous behavior, and that the corresponding clocks are updated when an enabling condition changes:

$$[tick]: \{V, c_1, \dots, c_n\} \xrightarrow{x_r} \{\delta(V, x_r), c'_1, \dots, c'_n\}$$

if $\bigwedge_i (c_i + x_r \leq u_{a_i} \text{ or } c_i == nil) \land (x_r \leq enabling_change(V))$

again, for V, c_1, \ldots, c_n , and x_r variables of appropriate sorts, where for all $k = 1, \ldots, n$,

 $c'_k = \mathbf{if} \ not(enabled_k(\delta(V, x_r)) \mathbf{then} \ nil \mathbf{else} \mathbf{if} \ c_k == nil \mathbf{then} \ 0 \mathbf{else} \ c_k + x_r.$

3.5 Object-oriented real-time systems

In a concurrent object-oriented system, the concurrent state, which is usually called a *configuration*, has typically the structure of a *multiset* made up of objects and messages. Therefore, we can view configurations as built up by a binary multiset union operator which we can represent with empty syntax as

 $_$: Configuration Configuration \rightarrow Configuration [assoc comm id: null]

where the multiset union operator $__$ is declared to satisfy the structural laws of associativity and commutativity and to have identity *null*. The subsort

declaration

$$Object, Msg \leq Configuration$$

states that objects and messages are singleton multiset configurations, so that more complex configurations are generated from them by multiset union. A sort *ObjConfiguration* denoting configurations without messages can be obtained by adding the subsort declaration

 $Object \leq ObjConfiguration \leq Configuration$

and the operator declaration

 $__: ObjConfiguration \ ObjConfiguration \ \rightarrow \ ObjConfiguration \\ [assoc \ comm \ id: \ null].$

Objects are terms (of sort *Object*) of the form

$$\langle O: C \mid att_1: val_1, \ldots, att_n: val_n \rangle$$

denoting an object named O, where O belongs to a set OId of *object identifiers*, of class C in a state having values val_1, \ldots, val_n for the attributes att_1, \ldots, att_n .

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

$$m(O, w) \langle O : C | att_1 : x, att_2 : y, att_3 : z \rangle \longrightarrow$$
$$\langle O : C | att_1 : x + w, att_2 : y, att_3 : z \rangle m'(y, x + w)$$

defines a (family of) transition(s) in which a message m having arguments Oand w is consumed by an object O of class C, with the effect of altering the attribute att_1 of the object and of generating a new message m'(y, x + w). By convention, attributes, such as att_3 in our example, whose values do not change and do not affect the next state of other attributes need not be mentioned in a rule. Attributes like att_2 whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, may be omitted from righthand sides of the rules. Thus the above rule could also be written

$$m(O, w) \langle O : C | att_1 : x, att_2 : y \rangle \longrightarrow \langle O : C | att_1 : x + w \rangle m'(y, x + w).$$

Real-time object-oriented systems can be specified by means of real-time rewrite theories by extending this setting with a sort System and an operator

$$\{_\}$$
: Configuration \rightarrow System.

Even though the tick rule will force the objects to synchronize in their time elapse, the system may still exhibit concurrency in its local transitions, which may occur between tick applications. We illustrate this style of specification of real-time object-oriented systems with a simple example.

3.5.1 Example: A single-thermostat system

A single-thermostat system consists of a thermostat object and zero or more "user" objects, defining the environment. The thermostat regulates the temperature by turning its heater on and off, and has to provide a temperature which is within 5 degrees of the user's desire, whenever this is possible. The temperature increases by 2 degrees per time unit when the heater is turned on, and decreases by 1 degree per time unit when the heater is turned off. The user may request a new desired temperature at any time by sending a message to the thermostat.

We assume that the specification includes a specification of Time, which satisfies the theory TIME, and a sort Temp denoting possible temperature values together with all the necessary operations. A sort OnOff contains the constants on and off, describing the state of the heater associated with the thermostat. A thermostat object has attributes $curr_temp$ and $desired_temp$ of sort Temp, denoting the current and desired temperatures, as well as an attribute *heater*, denoting the state of its heater. A user object is an object with an empty set of attributes.

In the following, let U and TS be variables of the sort *Oid* of object names, let x_r be a variable of sort *Time*, let y and z be variables of sort *Temp*, and let OC be a variable of the sort *ObjConfiguration* of messageless configurations.

At any time, a user may request a new desired temperature:

$$[new_temp]: \langle U: User \rangle \longrightarrow \langle U: User \rangle(set_temp(y)).$$

The thermostat object should treat such a message by recording the new

desired temperature (followed by the changing of the heater state if necessary):

 $[read_request]: (set_temp(y)) \langle TS : Thermostat | desired_temp : z \rangle \longrightarrow \\ \langle TS : Thermostat | desired_temp : y \rangle.$

The thermostat must turn off/on the heater, either when time has acted on a system such that the current temperature is exactly the desired temperature plus/minus 5 degrees, or when the system must change due to the adjustment of the desired temperature, in which case the current temperature may be more than 5 degrees off the desired temperature:

$$[on]: \langle TS: Thermostat | curr_temp: y, desired_temp: z, heater: off \rangle \longrightarrow \\ \langle TS: Thermostat | heater: on \rangle \text{ if } y \leq z - 5 \\ [off]: \langle TS: Thermostat | curr_temp: y, desired_temp: z, heater: on \rangle \longrightarrow \\ \langle TS: Thermostat | heater: on \rangle \text{ if } y \geq z + 5. \end{cases}$$

The following tick rules model the *effect* of time elapse on a system and ensure that:

- (1) time elapse can "stop" at any moment, reflecting the fact that the rule *new_temp* could be applied at any time,
- (2) time does not elapse past the moments the heater state should be changed, and
- (3) time does not elapse while there are any messages in the system (i.e., the requested temperature should be recorded at the time it is sent).

 $[tick_{on}]: \\ \{\langle TS : Thermostat | curr_temp : y, desired_temp : z, heater : on \rangle OC \} \xrightarrow{x_r} \\ \{\langle TS : Thermostat | curr_temp : y + x_r + x_r \rangle OC \} \text{ if } y + x_r + x_r \leq z + 5$

 $[tick_{off}]: \\ \{\langle TS : Thermostat | curr_temp : y, desired_temp : z, heater : off \rangle OC \} \xrightarrow{x_r} \\ \{\langle TS : Thermostat | curr_temp : y - x_r \rangle OC \} \text{ if } y - x_r \leq z - 5.$

The specification will work as expected, provided that the initial state contains exactly one thermostat object. A specification of a many-thermostat system is given in Section 3.7.2.

When the state of a system has a rich structure, it may be both natural and necessary to let a function denote the effect of time elapse on the whole state of a system, in contrast to, for example, the single-thermostat system in Section 3.5.1 where time elapse only affected one object in the system. The function δ denoting the action of time on a system has the form

 $\delta: State \ Time \rightarrow State$

involving the designated sorts *State* and *Time*. The action δ should be a *monoid action*, that is, it seems natural to require that it satisfies the axioms:

$$\delta(x, 0) = x$$

$$\delta(\delta(x, y_r), z_r) = \delta(x, y_r + z_r).$$

Tick rules should then be of the form

(†)
$$\{t\} \xrightarrow{r} \{\delta(t,r)\}$$
 if C .

Using the action δ to describe the effect of the passage of time on a dynamic evolution of a system is not without possible pitfalls. If done carelessly, it may allow "going back in time" to perform a rewrite. Suppose that $t = \delta(t', r)$ holds and that the "aged" term t' rewrites to t''. Then, there would also be an "aged" rewrite

$$\{t\} =_E \{\delta(t', r)\} \longrightarrow \{\delta(t'', r)\}.$$

For executable specification purposes it is important to require that the set E of equations in a rewrite theory is divided into a set E' of simplifying equations and a set Ax of structural axioms, in such a way that the equations in E' define a Church-Rosser and terminating set of equations modulo the set Ax, and such that the set of rules R is *coherent* [13,44,33] wrt. $E' \uplus Ax$. A rewrite theory is coherent if for every one-step sequential rewrite $t \longrightarrow t_1$ modulo the structural axioms Ax, there is also a rewrite $t!_{E'} \longrightarrow t'_1$ modulo Ax, for $t!_{E'}$ an E'-normal form of t modulo Ax, such that t_1 and t'_1 are E-equivalent. A coherent system does not allow "going back in time," since coherence would imply that that there is a "well-timed" rewrite $\{\delta(t', r)\} \longrightarrow \{\delta(t'', r)\}$ above, assuming that $\{\delta(t', r)\}$ reduces to $\{t\}$ when the equations are oriented.

A commonly occurring state structure for which we want the action of time to distribute over the different state components is a *multiset* distributed structure. For example, object-oriented systems and Petri nets have that structure. For multiset distributed systems we can give a general treatment of time actions that avoids coherence problems.

A simple solution to avoid coherence problems is to let each rule rewrite terms of sort *System* only, which would solve the coherence problem wrt. the symbol δ , since each rewrite would occur at the top. However, concurrency would be lost by this solution. Our idea is instead to use special tokens of the form '*' and let the extended state be a term in a supersort *ExtendedState* of the designated sort *State*, consisting of the multiset union of the original state and a multiset of tokens. The system operator $\{_\}$ should take arguments of the sort *ExtendedState*, while δ is left unchanged. If multiset union is denoted by juxtaposition, the tick rules would be of the form

$$(\ddagger) \quad [tick]: \{T t\} \xrightarrow{r} \{T \delta(t, r)\} \text{ if } C,$$

for T a variable of a sort *Tokens*, denoting multisets of tokens, and t a term of sort *State*. Each local rule should then have the form

$$* t \longrightarrow * \cdots * t'$$
 if C .

Since one token appears in the lefthand side of each local rule, the global state must contain at least n tokens for n local rewrites to fire concurrently. For object-oriented systems, the number of tokens in a configuration could suitably equal the number of objects in a configuration, since the number of rewrites firing concurrently is bounded by the number of objects present in the global state, under the assumption that at least one object appears in the lefthand side of each rule. Coherence wrt. the symbol δ is now trivially unproblematic, since every instance of a lefthand side of a local rule has least sort *ExtendedState*, and therefore cannot be an argument to δ .

To summarize, a monoid action δ denoting the effect of time elapse on the whole state may be useful for specifying real-time systems where the state of the system can have a rich distributed structure, but we must require coherence, since this ensures that δ does not cause counterintuitive rewrites resulting from "going back in time." The class of coherent real-time rewrite theories with a monoid action δ describing the effect of time elapse on a system and where the tick rules are of the form (†), or of the form (‡) for multiset distributed systems, will be denoted δ -RTRWTh.

3.7 Object-oriented δ -systems

The elapse of time affects one (functional) attribute in the single-thermostat system in Section 3.5.1. The tick rules could therefore be given by specifying directly the effect of time on that attribute. However, in more general object-oriented systems there can be an unbounded number of objects in a configuration which are affected by the elapse of time, in which case a finite number of tick rules cannot specify the elapse of time directly on the functional attributes. A simple solution is to use a function δ denoting the action of time on a configuration. The important class of real-time object-oriented systems where the objects have only functional attributes are multiset distributed systems. Therefore, we can use the techniques described in Section 3.6 to circumvent coherence problems wrt. δ without sacrificing concurrency. The following declarations should be added to the general framework for specifying object-oriented real-time systems given in Section 3.5, with $\{-\}$ redefined as stated below.

sorts	Tokens, *Configuration
$\mathbf{subsorts}$	$Tokens, Configuration \leq *Configuration$
ор	$*: \rightarrow Tokens$
ор	$__: Tokens \ Tokens \rightarrow Tokens \ [assoc \ comm \ id: null]$
ор	$__: * Configuration * Configuration \rightarrow * Configuration$
	$[assoc \ comm \ id: null]$
ор	$\{ _ \} : * Configuration \rightarrow System$
ор	δ : Configuration Time \rightarrow Configuration.

As already mentioned, the tick rules should be of the form

 $[tick]: \{T t\} \xrightarrow{r} \{T \delta(t, r)\}$ if C,

where T is a variable of sort Tokens, and t is a term of sort Configuration. Each instantaneous rule should have the form 6

$$[l]: *t \longrightarrow *\cdots *t',$$

for t, t' terms of sort *Configuration*, and where the number of tokens * in the righthand side should equal one plus the number of objects created by the rule, minus the number of objects deleted by the rule. The initial state of a

⁶ In systems where the number of objects created by a rule application depends on the state, the condition on the form of the rules could be relaxed so that $*\cdots *$ can be given by a term of sort *Tokens*, computing the number of tokens as a function of the state.

system should be of the form $\{* \cdots * t\}$, where the number of tokens equals the number of objects in the term t of sort *Configuration*.

3.7.1 Distribution over configurations

An operator δ acting on configurations provides, as we have just seen, a natural way of expressing the action of time on object systems where the number of objects in a configuration upon which time has an effect is unbounded. In these cases, δ should typically distribute over the elements, or over groups of elements, in the configuration. The former case can be modeled by the axioms

$$\delta(null, x_r) = null$$

$$\delta(C C', x_r) = \delta(C, x_r) \,\delta(C', x_r) \text{ if } C \neq null \text{ and } C' \neq null$$

(for C, C' variables of sort *Configuration*), to which the definition of the specific effect of time on single objects and on messages must be added to completely specify δ . The condition that C and C' be different from *null* ensures that the two equations above define a terminating rewrite system modulo associativity, commutativity, and identity (*null*) of the configuration union operator, when oriented from left to right.

In systems parameterized by $LTIME_{\infty}$ theories, a function *mte* giving the maximum time elapse of an object and message can be extended to configurations by the axioms

$$mte(null) = \infty$$

 $mte(C C') = min(mte(C), mte(C'))$ if $C \neq null$ and $C' \neq null$.

3.7.2 Example: A multi-thermostat system

A multi-thermostat system can have an arbitrary number of rooms, each equipped with a thermostat that works as in the single-thermostat system. Each user object is extended to contain a list of the thermostats to which it has access.

Let the specification be parameterized by the theory $LTIME_{\infty}$. Furthermore, let A, TS, and U be variables of sort OId, let S be a variable of a sort of sets of OIds, let C be a variable of sort Configuration, let T be a variable of sort Tokens, let x_r be a variable of sort Time, and let y and z be variables of sort Temp, modeling the temperature domain. Then, the function δ : Configuration Time \rightarrow Configuration denoting the action of time, and the function mte : Configuration \rightarrow Time_{∞} computing the maximum time elapse in a tick both distribute over configurations according to the equations described above, and are defined for singleton configurations as follows:

$$\begin{split} &\delta(set_temp(A, y), x_r) = set_temp(A, y) \\ &\delta(\langle U : User \rangle, x_r) = \langle U : User \rangle \\ &\delta(\langle TS : Thermostat | curr_temp : y, heater : on \rangle, x_r) = \\ &\langle TS : Thermostat | curr_temp : y + x_r + x_r, heater : on \rangle \\ &\delta(\langle TS : Thermostat | curr_temp : y, heater : off \rangle, x_r) = \\ &\langle TS : Thermostat | curr_temp : y - x_r, heater : off \rangle \\ &mte(set_temp(A, y)) = 0 \\ &mte(\langle U : User \rangle) = \infty \\ &mte(\langle TS : Thermostat | curr_temp : y, desired_temp : z, heater : on \rangle) = \\ &((z + 5) - y)/2 \\ &mte(\langle TS : Thermostat | curr_temp : y, desired_temp : z, heater : off \rangle) = \\ &((z + 5) - y)/2. \end{split}$$

The system's transition rules can then be given as follows:

$$[new_temp] : * \langle U : User | thermostats : A S \rangle \longrightarrow * \langle U : User \rangle (set_new_temp(A, y)) [set_temp] : * (set_temp(A, y)) \langle A : Thermostat | desired_temp : z \rangle \longrightarrow * \langle A : Thermostat | desired_temp : y \rangle [turn_on] : * \langle TS : Thermostat | curr_temp : y, desired_temp : z, heater : off \rangle \longrightarrow * \langle TS : Thermostat | curr_temp : y, desired_temp : z, heater : off \rangle \square * \langle TS : Thermostat | curr_temp : y, desired_temp : z, heater : on \rangle [turn_off] : * \langle TS : Thermostat | curr_temp : y, desired_temp : z, heater : on \rangle \square * \langle TS : Thermostat | curr_temp : y, desired_temp : z, heater : on \rangle \square * \langle TS : Thermostat | heater : off \rangle \\ if y \ge z + 5 [tick] : {T C} \xrightarrow{x_r} {T \delta(C, x_r)} \\ if x_r < mte(C).$$

3.8 Timed Petri nets

A Petri net [41] is usually presented as a set of *places* (each place representing a certain kind of *resource*), a disjoint set of *transitions*, and a relation of causality between them that associates to each transition the set of resources consumed and produced by its firing. Meseguer and Montanari recast this idea in an algebraic framework in [36], viewing the distributed states of the net, called *markings*, as multisets of places, and viewing the transitions as the arrows of an ordinary graph whose nodes are markings. In [32,35] it has been shown how Petri net computations can be expressed by rewriting of *markings*, that

is, of multisets over the set of places.

Petri nets have been extended to model real-time systems in different ways (see e. g. [1,37,21]). Three of the most frequently used time extensions are the following, from which other timed versions of Petri nets can be obtained either as special cases or by combining the extensions:

- (1) Each transition α has an associated time interval $[l_{\alpha}, u_{\alpha}]$. A transition fires as soon as it can, but the resulting resources (also called *tokens*) are delayed, that is, when a transition α fires, the resulting resources are not visible in the system until after some time $r \in [l_{\alpha}, u_{\alpha}]$.
- (2) Each place p has a duration r_p . A resource of kind p cannot participate in a transition until it has been at place p for at least time r_p .
- (3) 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 and phase transition systems.

Some of the timed extensions of Petri nets, such as *interval timed colored Petri nets*, appear in the context of *colored* Petri nets, where instead of having *atomic* places one has structured data. In this exposition, we abstract away from the colors of the tokens to concentrate on real-time features (see e.g., [43] for a treatment in rewriting logic of an important class of colored nets, namely, *algebraic* Petri nets).

The translation into rewriting logic of these first two cases is based on the rewriting logic representation of untimed Petri nets given in [35,32,34], where the state of a Petri net is represented by a multiset of places called a *marking* – where if place p has multiplicity n we interpret this as the presence of n tokens in that place – and where the transitions correspond to rewrite rules on the corresponding multisets of pre- and post-places.

There are two kinds of "tokens" in our translation of timed Petri nets: A term consisting of the place p represents a "visible" occurrence of a token at place p. A token that will be visible at place p in time r is represented by the term

Clearly, we want the equation dly(p,0) = p. A state, or *marking*, of a timed Petri net is a multiset of these two forms of places, where multiset union is represented by juxtaposition.

The number of delayed tokens in a marking, upon which time acts, is not

known in advance. Timed Petri nets are therefore best modeled by δ -realtime theories using the techniques for specifying multiset distributed δ -systems given in Section 3.6. The action δ : Marking Time \rightarrow Marking which models the effect of time elapse on a marking (without any occurrence of the special symbol '*'), distributes over the elements in a marking and is defined as follows:

$$\begin{split} \delta(p, x_r) &= p\\ \delta(dly(p, x_r), y_r) &= dly(p, x_r \div y_r)\\ \delta(null, x_r) &= null\\ \delta(M \ M', x_r) &= \delta(M, x_r) \ \delta(M', x_r) \ \mathbf{if} \ M \neq null \ \mathbf{and} \ M' \neq null. \end{split}$$

Transitions are represented by rewrite rules whose lefthand side requires an extra token of the form '*' and with new extra such tokens added on the righthand side according to the increase in cardinality from the preset to the postset. In version (1), 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 at each of the 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:

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

In version (2), each place p has an associated duration r_p , and a token must have been at a place p for at least time r_p before it can be used in any transition. This is equivalent to saying that the produced token cannot be visible before time r_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 by the rule

eager
$$[\alpha]$$
 : $* a \ a \ b \longrightarrow * \ dly(c, r_c) \ dly(d, r_d).$

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 function *mte* is used. It takes as argument a marking (without *), and returns the least amount of time until one or more non-available tokens become available:

$$mte(p) = \infty$$

$$mte(dly(p, x_r)) = x_r \text{ if } x_r \neq 0$$

$$mte(null) = \infty$$

$$mte(M \ M') = \min(mte(M), mte(M')) \text{ if } M \neq null \text{ and } M' \neq null.$$

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

lazy [tick]:
$$\{T M\} \xrightarrow{x_r} \{T \delta(M, x_r)\}$$
 if $x_r \leq mte(M)$,

where T is a variable of sort *Tokens* of multisets of '*', and M is a variable of sort *Marking*. Note that for simulation purposes, a tick rule

lazy [tick]:
$$\{T M\} \xrightarrow{mte(M)} \{T \delta(M, mte(M))\}$$
 if $mte(M) \neq \infty$

would be simpler, since no (non-tick) transition which is not currently enabled will be enabled before time has elapsed at least mte(M). The reason for the nondeterministic tick rule is to allow every moment in the time domain to be visited.

Transitions are supposed to fire as soon as possible in both versions of timed Petri nets. This is accomplished by the strategy described in Section 2.5.3 that triggers all eager instantaneous rules until none of these can be applied, followed by one application of the tick rule.

The correspondence between a version-(1) timed net and its rewriting logic translation can be given as follows. For any *net* markings (i.e., markings without "delayed" tokens or *s) M_0 and M_1 , the marking M_1 can be reached in time r from the initial marking M_0 in the net if and only if there is a rewrite

$$\alpha: \{*\cdots * M_0\} \longrightarrow \{*\cdots * M_1'\}$$

in the rewriting logic translation with $\tau(\alpha) = r$, with the number of occurrences of '*' in the lefthand side system greater or equal to the number of (non-*) tokens in M_0 , and with M_1 obtained from M'_1 by removing each delayed token from M'_1 .

In version (2), we have the correspondence that, for any net marking M_0 , the net can reach a marking M_1 in time r from the initial marking M_0 if and only

if there is a rewrite

$$\beta: \{*\cdots * M_0'\} \longrightarrow \{*\cdots * M_1'\}$$

in the corresponding rewrite translation, where $\tau(\beta) = r$, where the number of occurrences of '*' in the lefthand side system greater or equal to the number of (non-*) tokens in M_0 , where M'_0 is obtained from M_0 by replacing each occurrence of a token at place p by a delayed token $dly(p, r_p)$, and where M_1 can be obtained from M'_1 by replacing each delayed token in M'_1 by the corresponding undelayed token.

Our rewriting logic specification of timed Petri nets illustrates the convenience of using eager and lazy rules, which allow a simple condition on the tick rule, which would otherwise have to take into account the enabledness of every transition in the system together with the *mte* part of the tick rule. Here, the tick rule only needs to compute the time until the next delayed token becomes "visible" and elapse time by that amount. After such a tick, the tick rule is again enabled but will, due to its being lazy, not be applied if the introduction of the new token enabled an (eager) transition (which in turn could trigger more transitions in zero-time).

3.9 The big picture

We have shown how some well-known models of real-time and hybrid systems can be naturally regarded as specializations of the real-time rewriting logic framework. Since we are interested in executable specifications, we have placed some computability restrictions on some models. The relationships between the models considered are summarized in Figure 1, where the arrows in the tree stand for specialization⁷, where the acronym OORTS stands for objectoriented real-time systems, δ -RTRWTh for real-time rewrite action theories, and δ -OORTS for real-time object-oriented action systems. Even though we have not presented an exhaustive discussion of real-time models, we think that the models we have chosen are significantly varied and well-known to suggest that rewriting logic is a good semantic framework for real-time and hybrid systems.

⁷ The timed automata model requires its computations to have unbounded total time elapse and to satisfy certain acceptance criteria, which is not the case for hybrid automata, explaining the missing arrow between these models.



Fig. 1. Specializations of real-time rewriting.

4 Relationship to timed rewriting logic

In this section we investigate the relationship between Kosiuczenko and Wirsing's timed rewriting logic (TRL) [24,25] and the framework we have presented for specifying real-time systems directly in rewriting logic. After briefly introducing TRL in Section 4.1, we propose in Section 4.2 a translation from TRL into rewriting logic. In this translation, the translation of any TRL-sequent derivable in a TRL theory is also derivable in the corresponding rewriting logic theory. The converse is in general not true. We explain the reasons for this discrepancy in Section 4.3. They are due to some conceptual differences between TRL and our method of specifying real-time systems in rewriting logic.

4.1 Timed rewriting logic

Rewriting logic has been extended by Kosiuczenko and Wirsing to handle realtime systems in their *timed rewriting logic* (TRL) [24,25]. TRL has been shown well-suited for giving object-oriented specifications of complex hybrid systems such as the steam-boiler [38] 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.

A TRL theory (Σ, E, L, TR) consists of an equational specification (Σ, E) satisfying the theory $TIME^8$, a set L of labels, and a set TR of timed rewrite rules of the form $[l]: t \xrightarrow{r} t'$, where r is a ground term of sort Time denoting the duration of the rewrite. A TRL sequent has the form $t \xrightarrow{r} t'$ and its intuitive meaning is that t evolves to t' in time r. More specifically, the set of sequents derivable from a TRL theory consists of all rules in the theory,

 $^{^{8}}$ They impose in some cases further requirements, such as TIME being an Archimedean monoid. This could of course be easily accommodated.

Timed transitivity:

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

Synchronous replacement:

$$\frac{t_0 \stackrel{r}{\longrightarrow} t'_0, \quad t_{i_1} \stackrel{r}{\longrightarrow} t'_{i_1}, \dots, t_{i_k} \stackrel{r}{\longrightarrow} t'_{i_k}}{t_0(t_1/x_1, \dots, t_n/x_n) \stackrel{r}{\longrightarrow} t'_0(t'_1/x_1, \dots, t'_n/x_n)}$$

where $\{x_{i_1},\ldots,x_{i_k}\} = \mathcal{V}(t_0) \cap \mathcal{V}(t'_0).$

Compatibility with equality:

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

Renaming of variables:

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

Fig. 2. Deduction rules in timed rewriting logic.

and all sequents which can be derived by equational reasoning and by using the deduction rules in Figure 2, 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 arbitrarily long period of time.
- 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 components of a system: a system rewrites in time r iff all its components do so.
- 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.

4.2 Timed rewriting logic in rewriting logic

In this section we define a mapping \mathcal{M} which takes any timed rewriting logic theory \mathcal{T} to a real-time rewrite theory $\mathcal{M}(\mathcal{T})$ such that $\mathcal{T} \vdash t \xrightarrow{r} t'$ implies that $\mathcal{M}(\mathcal{T}) \vdash \alpha : \{t\} \longrightarrow \{t'\}$, for some α with $\tau(\alpha) = r$, for all ground (\mathcal{T}) - terms t, t' of the designated sort *State*.

The idea is to introduce for each sort s an operator $\delta : s Time \to s$ corresponding to the effect of time elapse. Then, a TRL sequent $t \xrightarrow{r} t'$ ("t evolves in time r to t'") can be mapped 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'. Rewrite rules must be used to define δ , since the effect of time on a TRL state is not necessarily functional.

Sort information is used to separate terms containing the symbol δ from terms of the original signature, and a tick rule is added to the rules defining δ such that for ground \mathcal{T} -terms t, t' of sort $State, \mathcal{M}(\mathcal{T}) \vdash \alpha : \{t\} \longrightarrow \{t'\}$ holds for some α with $\tau(\alpha) = r$ if and only if $\mathcal{M}(\mathcal{T}) \vdash \delta(t, r) \longrightarrow t'$, which in turn holds whenever $\mathcal{T} \vdash t \xrightarrow{r} t'$ holds. The resulting real-time rewrite theory $\mathcal{M}(\mathcal{T})$ is not easily executable, since the tick rule introduces two variables in its righthand side. This reflects the fact that in TRL it is in general undecidable whether a term rewrites in time r (r > 0), and, even if it is known that trewrites in time r, it is also in general undecidable whether t rewrites to a given term t' in time r.

We assume that the time domain is functional, that is, that no rewrites of the form $t \xrightarrow{r} t'$, with $t \neq t'$ terms of sort *Time*, can be inferred from the TRL theory \mathcal{T} , and restrict our treatment to TRL theories where no extra variables are introduced in the righthand side of a rule. The reason for the latter 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 2 time units 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 2 time units.

4.2.1 The mapping from TRL to real-time rewrite theories

The mapping \mathcal{M} sends an order-sorted TRL theory $\mathcal{T} = (\Sigma, E, L, TR)$ to a real-time rewrite theory $\mathcal{M}(\mathcal{T}) = ((\mathcal{M}(\Sigma), \mathcal{M}(E), \mathcal{M}(L), \mathcal{M}(TR)), \phi(\mathcal{T}), \tau(\mathcal{T}))$ and sends a \mathcal{T} -sequent $t \xrightarrow{r} t'$ to an $\mathcal{M}(\mathcal{T})$ -sequent $\mathcal{M}(t \xrightarrow{r} t')$. It is defined as follows:

- The signature morphism $\phi(\mathcal{T})$ in $\mathcal{M}(\mathcal{T})$ takes *Time* to the sort *Time* in \mathcal{T} denoting the time domain, and takes the functions in *TIME* to the corresponding functions in \mathcal{T} .
- The set of sorts in $\mathcal{M}(\Sigma)$ consists of all the sorts in Σ , plus a new sort s^{δ} for each sort s in Σ , as well as a new sort *System*. For each sort s in Σ , $s \leq s^{\delta}$, and if $s \leq s'$ in Σ , then $s^{\delta} \leq s'^{\delta}$ in $\mathcal{M}(\Sigma)$.

- $\mathcal{M}(\Sigma)$ contains function declarations $f: s_1 \ldots s_n \to s$ and $f: s_1^{\delta} \ldots s_n^{\delta} \to s^{\delta}$ for each function $f: s_1 \ldots s_n \to s$ in Σ , a constructor $\{-\}: State \to System$ for the designated sort *State* in Σ , and a function $\delta: s^{\delta}$ *Time* $\to s^{\delta}$ for each sort s in Σ .
- $\mathcal{M}(E)$ contains an axiom e^{δ} for each axion e in E, where e^{δ} is the axiom e where each variable x : s is replaced by $x : s^{\delta}$. The set $\mathcal{M}(E)$ must also define δ to be a monoid action, that is, it contains the axioms

$$\delta(x_{s^{\delta}}, 0_{\phi}) = x_{s^{\delta}}$$

 $\delta(\delta(x_{s^{\delta}}, y_r), z_r) = \delta(x_{s^{\delta}}, y_r +_{\phi} z_r)$

for each sort s^{δ} in $\mathcal{M}(\Sigma)$ and variable $x_{s^{\delta}}$ of sort s^{δ} , and variables y_r, z_r of sort *Time*.

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

$$\mathcal{M}(t(x_1:s_1,\ldots,x_n:s_n) \xrightarrow{r} t'(x_1:s_1,\ldots,x_n:s_n)) = \\ \delta(t(x_1:s_1^{\delta},\ldots,x_n:s_n^{\delta}),r) \longrightarrow t'(\delta(x_1:s_1^{\delta},r)/x_1,\ldots,\delta(x_n:s_n^{\delta},r)/x_n)$$

where the free variables in t are x_1, \ldots, x_n and contain those of t'.

The set of rules $\mathcal{M}(TR)$ consists of a rule $[l] : \mathcal{M}(t \xrightarrow{r} t')$ if C^{δ} for each timed rule $[l] : t \xrightarrow{r} t$ if C in TR, and a tick rule

$$[tick]: \{x\} \xrightarrow{y_r} \{x'\}$$
if $\delta(x, y_r) \longrightarrow x'$

for variables x, x' of sort *State* and y_r of sort *Time*.

The theorem below shows that \mathcal{M} can be naturally understood as a map of logics. Specifically, as a map $\mathcal{M} : TRL \longrightarrow RWL$ from the entailment system [31] of TRL to that of rewriting logic.

Theorem 10 Let \mathcal{T} be a TRL specification and let \mathcal{M} be defined as above. Then, for all terms $t, t', r \in \mathcal{T}_{\Sigma}(X)$,

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

As a corollary to 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, which in turn gives a rewrite $\mathcal{M}(\mathcal{T}) \vdash \alpha : \{t\} \longrightarrow \{t'\}$ with $\tau(\alpha) = r$ when t and t' are of sort *State* by applying the tick rule. It is also easy to see that $\mathcal{M}(\mathcal{T}) \vdash \alpha : \{t\} \longrightarrow \{t'\}$ implies $\mathcal{M}(\mathcal{T}) \vdash \delta(t, \tau(\alpha)) \longrightarrow t'$ for ground \mathcal{T} -terms t, t' of sort *State*.

4.3 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 we discuss the differences between deduction in TRL and in its translation into rewriting logic.

4.3.1 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.

4.3.2 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

$$\{ \delta(f(x), 2) \longrightarrow g(\delta(x, 2), \delta(x, 2)), \ \delta(a, 2) \longrightarrow b, \ \delta(a, 2) \longrightarrow c,$$
$$\{y\} \xrightarrow{x_r} \{y'\} \text{ if } \delta(y, x_r) \longrightarrow y' \},$$

where y and y' are variables of the designated state sort and range over δ -free terms, there is a rewrite $\delta(f(a), 2) \longrightarrow g(\delta(a, 2), \delta(a, 2)) \longrightarrow g(b, c)$, and therefore also a rewrite $\alpha : \{f(a)\} \longrightarrow \{g(b, c)\}$ with $\tau(\alpha) = 2$.

The difference depends on how the fork of a process is modeled. 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^9 .

⁹ 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".

4.3.3 Problems related to synchrony 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 synchrony 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).

4.4 Aging in TRL

To overcome the strong requirements of synchrony in TRL, which caused the differences in Sections 4.3.2 and 4.3.3, the special symbol age is introduced in [24,25]. 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 2. In both deduction rules, $t \xrightarrow{r+r'} t'$ is assumed to be a timed rewrite *rule* in the specification.

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

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 4.3.2, and the strong synchrony is loosened, as illustrated by the fact that in Section 4.3.3, $f(a, b) \xrightarrow{6} k(d, c)$ is derivable, since $f(a, b) \xrightarrow{2} g(a, age(b, 2)), g(a, age(b, 2)) \xrightarrow{2} h(age(a, 2), c), \text{ and } h(age(a, 2), c)) \xrightarrow{2} k(d, c)$ are derivable.

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

Example 11 In a TRL theory $\{f(x) \xrightarrow{2} g(x), f(b) \xrightarrow{2} g(c), a \xrightarrow{2} b\}$, one would expect $f(a) \xrightarrow{2} g(c)$ not to be 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 Concluding Remarks

We have presented a general method for specifying real-time and hybrid systems in rewriting logic in an executable way, have shown how a wide range of real-time and hybrid system models can be naturally expressed in rewriting logic, and have illustrated the ideas with several examples. This work should be further extended in several directions.

The systems that we have considered can be distributed and can exhibit concurrent computations, in which several components of the state can change simultaneously and independently. However, time is still in some sense global, since time acts on the global state, even though its effects can be local and distributed—for example, by advancing the local clocks of different distributed objects. The situation is entirely similar to that in some real-time models for distributed systems such as Lynch's general timed automata [26], where time also acts uniformly on all the distributed components. In fact, although we have not discussed general timed automata in this paper, they can also be specified within our general framework. Although the current framework can already be used for specifying and reasoning about a range of distributed timebased systems, it would be worth investigating how the assumption of global time action could be relaxed to local or distributed time actions.

We have explored what we think is a representative range of real-time and hybrid system models. However, the general timed (I/O) automata model mentioned above, real-time dataflow models such as Lustre's [20], and a variety of other models should also be specified in detail in rewriting logic. The interest is not merely conceptual: by using a formal meta-tool such as Maude [16], one can turn the rewriting logic specification of a model into a tool for executing and analyzing formal specifications in that model. Since at present some formalisms lack execution and analysis environments, this offers a way of developing new formal tools with considerably less effort than what would be required for conventional implementations.

Execution of rewriting logic specifications for real-time and hybrid systems is also another area deserving further work. Since the specifications are rewrite theories, and we assume that the underlying data types are computable, they can of course be executed in a rewriting logic language. The point, however, is that the rewrite rules are often *nondeterministic*, with extra new variables appearing on the righthand side. Therefore, they should be executed with appropriate *strategies*, to guide both the application of the rules and the choice of instantiations for the extra variables in a match. Strategies of this kind can be defined without any problem in languages such as ELAN [10] and Maude [13], but the development of a good library of such strategies suitable for real-time and hybrid system applications—leading perhaps to a specialized execution and analysis tool for them—remains to be done.

Another important research issue is the integration of different proof and analysis methods. On the one hand, verification of property-oriented specifications should be supported. This can be done either by inductive methods, based on the initial model of the rewriting logic specification, or by temporal logic reasoning, which in important cases can be supported by abstraction and model checking techniques. On the other hand, once we have an executable specification we can subject it to other forms of analysis, ranging from execution with a default strategy, to exploration of different computation paths with more sophisticated strategies, and to full symbolic simulation with techniques such as narrowing. Studying how all these different methods and their tools can best be combined to make system analysis and verification easier seems a promising research direction. Examples and case studies can help very much in this task.

Acknowledgements

We cordially thank Christiano Braga, Manuel Clavel, Francisco Durán, Alexander Knapp, 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

- 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] R. Alur and D. L. 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] R. Alur and D. L. Dill. A theory of timed automata. Theoretical Computer Science, 126(2), 1994.
- [5] M. Barr and C. Wells. Electronic supplement to the second edition of "Category Theory for Computing Science". Available at http://www.cwru.edu/artsci/ math/wells/pub/papers.html.
- [6] M. J. Beeson. Foundations of Constructive Mathematics. Springer-Verlag, 1985.
- [7] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
- [8] J. A. Bergstra and J. V. Tucker. Algebraic specification of computable and semicomputable data types. *Theoretical Computer Science*, 50:137–181, 1987.
- [9] N. Bjørner, Z. Manna, H. B. Sipma, and T. E. Uribe. Deductive verification of real-time systems using STeP. In Proc. of ARTS'97, volume 1231 of Lecture Notes in Computer Science. Springer-Verlag, 1997.
- [10] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. http: //www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm.
- [11] M. Clavel. Reflection in general logics, rewriting logic, and Maude. PhD thesis, University of Navarre, 1998.
- [12] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science. Elsevier, 1998.
- [13] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International, Menlo Park, 1999. http: //maude.csl.sri.com.

- [14] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, Proc. Reflection'96. Xerox PARC, 1996.
- [15] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, Proc. 1st Intl. Workshop on Rewriting Logic and its Applications, volume 4 of Electronic Notes in Theoretical Computer Science. Elsevier, 1996.
- [16] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Maude as a formal meta-tool. SRI International, February 1999, http://maude.csl.sri. com.
- [17] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Hybrid Systems III, volume 1066 of Lecture Notes in Computer Science. Springer-Verlag, 1996.
- [18] K. Futatsugi and R. Diaconescu. CafeOBJ report. AMAST Series, World Scientific, 1998.
- [19] J. A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
- [20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9), September 1991.
- [21] H. M. Hanisch. Analysis of place/transition nets with timed arcs and its application to batch process control. In M. Ajmone Marsan, editor, Application and Theory of Petri Nets 1993, volume 691 of Lecture Notes in Computer Science. Springer-Verlag, 1993.
- [22] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HyTech. In TACAS'95, volume 1019 of Lecture Notes in Computer Science. Springer-Verlag, 1995.
- [23] Y. Kesten, Z. Manna, and A. Pnueli. Verifying clocked transition systems. In Proc. Hybrid Systems III, volume 1066 of Lecture Notes in Computer Science. Springer-Verlag, 1996.
- [24] P. Kosiuczenko and M. Wirsing. Timed rewriting logic for the specification of time-sensitive systems. In H. Schwichtenberg, editor, Proc. Intl. Summer School on Proof and Computation, Marktoberdorf, 1995. Springer-Verlag, 1997.
- [25] P. Kosiuczenko and M. Wirsing. Timed rewriting logic with an application to object-based specification. Science of Computer Programming, 28(2-3), 1997.
- [26] Nancy Lynch. Distributed Algorithms. Morgan Kaufmann, 1996.
- [27] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In Real-Time: Theory in Practice, volume 600 of Lecture Notes in Computer Science, 1991.

- [28] Z. Manna and A. Pnueli. Models for reactivity. Acta Informatica, 30, 1993.
- [29] Z. Manna and A. Pnueli. Clocked transition systems. Logic and Software Engineering, 1996. Also available as Stanford University CSD technical report STAN-CS-TR-96-1566.
- [30] Z. Manna and H. Sipma. Deductive verification of hybrid systems using STeP. In Hybrid Systems: Computation and Control, volume 1386 of Lecture Notes in Computer Science. Springer-Verlag, 1998.
- [31] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, Logic Colloquium'87. North-Holland, 1989.
- [32] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
- [33] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
- [34] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In Proc. Concur'96, volume 1119 of Lecture Notes in Computer Science. Springer-Verlag, 1996.
- [35] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997. Springer-Verlag, 1998.
- [36] J. Meseguer and U. Montanari. Petri nets are monoids. Information and Computation, 88, 1990.
- [37] S. Morasca, M. Pezzè, and M. Trubian. Timed high-level nets. The Journal of Real-Time Systems, 3:165-189, 1991.
- [38] 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, volume 1165 of Lecture Notes in Computer Science. Springer-Verlag, 1996.
- [39] P. C. Ölveczky and J. Meseguer. Specifying real-time systems in rewriting logic. In J. Meseguer, editor, Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, volume 4 of Electronic Notes in Theoretical Computer Science. Elsevier, 1996.
- [40] M. Rabin. Computable algebra: General theory and theory of computable fields. Transactions of the American Mathematical Society, 95:341–360, 1960.
- [41] W. Reisig. Petri Nets, volume 4 of EATCS monographs on Theoretical Computer Science. Springer-Verlag, 1985.

- [42] L. J. Steggles and P. Kosiuczenko. A timed rewriting logic semantics for SDL: A case study of the alternating bit protocol. In Proc. 2nd International Workshop on Rewriting Logic and its Applications, volume 15 of Electronic Notes in Theoretical Computer Science. Elsevier, 1998. At http://www.elsevier.nl/locate/entcs/volume15.html.
- [43] M.-O. Stehr. A rewriting semantics for algebraic Petri nets. Manuscript, SRI International and C.S. Dept., Univ. of Hamburg, 1998.
- [44] P. Viry. Rewriting: An effective model of concurrency. In *Proc. PARLE'94*, volume 817 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.