A Real-Time Specification Language

Fernando Náufel do Amaral
e-mail: fnaufel@inf.puc-rio.br

Edward Hermann Haeusler
e-mail: hermann@inf.puc-rio.br

Markus Endler
e-mail: endler@inf.puc-rio.br

PUC-RioInf.MCC22/02 August, 2002

Abstract: A specification language for real-time software systems is presented, along with a model-theoretic semantics. Notions from Category Theory are used to specify how the components of a system should interact. The potential role of the proposed language in the search for interoperability of specification formalisms is briefly discussed.

Keywords: real-time systems, concurrent systems, formal specification, category theory, interoperability.

Resumo: Apresenta-se uma linguagem de especificação para sistemas de *software* de tempo real, acompanhada de uma semântica em teoria dos modelos. Usam-se noções de Teoria das Categorias para especificar como os componentes de um sistema devem interagir. Discute-se brevemente o papel potencial da linguagem proposta na busca pela interoperabilidade de formalismos de especificação.

Palavras-chave: sistemas de tempo real, sistemas concorrentes, especificação formal, teoria das categorias, interoperabilidade.

A Real-Time Specification Language

Fernando Náufel do Amaral,
Edward Herman Haeusler,
Markus Endler
Dept. of Informatics, PUC-RJ,Brazil

September 27, 2002

Abstract

A specification language for real-time software systems is presented, along with a model-theoretic semantics. Notions from Category Theory are used to specify how the components of a system should interact. The potential role of the proposed language in the search for interoperability of specification formalisms is briefly discussed.

1 Introduction

The aim of this work is to present RT-Community, a specification language for real-time reactive systems. Using the language, one is able to specify the computations of the individual components of a real-time system as well as the way these components interact with each other. This makes RT-Community suitable as an architecture description language (ADL) in the sense of [2].

RT-Community is an extension of the specification language Community ([9]), from which it inherits its characteristics as a coordination language ([5]). Roughly speaking, this means that it supports the separation between computational concerns (what each component does) and coordination concerns (how components are put together and how they interact to present the behavior expected of the system as a whole).

The clear separation between computation and coordination may facilitate the integration of different formalisms in the specification of a system: the computations of the components may be specified at different levels of abstraction, using different languages (e.g., automata, temporal logics, programming languages etc.), whereas the signatures of the components, which indicate their functionality and the potential ways they can be coordinated with other components, are the only features of components that need to be taken into account in order to describe the architecture of the composite system.

The modeling of time in RT-Community merits some comments. Although the language and its semantics are essentially independent of the time domain used (any total order with a minimum element will do), we have adopted the set of nonnegative real numbers ($\mathbb{R}_{\geq 0}$), differently from other formalisms in the literature ([6], for instance), which represent time as a discrete total order. We believe a densely-ordered time domain is more adequate to model the behavior of physical systems working in continuous time; a discrete time domain would force us to choose a fixed "time quantum" *a priori*, a decision which may bring the specification of a system inconveniently close to implementation concerns. Among other formalisms which use dense time domains, we cite Alur and Dill's Timed Automata ([3]) and Wang's TCCS ([12]).

This paper presents the syntax and formal semantics of RT-Community in section 2. Section 3 shows how composition is done in RT-Community by way of some basic notions of Category Theory. Finally, section 4 offers some concluding remarks, especially about the potential role of RT-Community in the search for interoperability of specification formalisms.

2 RT-Community

An RT-Community component has the form shown in Figure 1, where

- V is a finite set of variables, partitioned into input variables, output variables and private variables. Input variables are read by the component from its environment; they cannot be modified by the component. Output variables can be modified by the component and read by the environment; they cannot be modified by the environment. Private variables can be modified by the component and cannot be seen (i.e. neither read nor modified) by the environment. We write $loc(V)$ for $prv(V) \cup out(V)$. We assume given but do not make explicit the specification of the data types over which the variables in V range.

- C is a finite set of clocks. Clocks are like private variables in that they cannot be seen by the environment; they can be consulted (i.e. their current values may be read) by the component; however, the only way a component can modify a clock variable is by resetting it to zero.

- F is a formula specifying the initial state of the component (i.e., conditions on the initial values of variables in $loc(V)$). The initial values of input variables are unconstrained, and clocks are always initialized to zero.

- Γ is a finite set of action names. Actions may be either shared or private. Shared actions are available for synchronization with actions of other components, whereas the execution of private actions is entirely under control of the component.

- The body of the component consists of a set of instructions resembling guarded commands. For each action $g \in \Gamma$, we have the *time guard* $T(g)$, which is a boolean expression over atomic formulae of the form $x \sim n$ and $x - y \sim n$, where $x, y \in C, n \in \mathbb{R}_{\geq 0}$ and $\sim$ is one of $\{<, >, =\}$; the *data guard* $B(g)$, which is a boolean expression constructed from the operations and predicates present in the specification of the data types of V; the *reset clocks* $R(g) \subseteq C$, containing the clocks to be reset upon execution of g; and the *parallel assignment* of a term $F(g, v)$ to each variable v that can be modified by g. We denote by $D(g)$ the set of variables that can be modified by g. This set will be called the write frame (or domain) of g.

The execution of a component proceeds as follows: at each step, an action whose time and data guards are true (such an action is said to be *enabled*)

component P
  in `in` (V)
  out `out` (V)
  prv `prv` (V)
  clocks C
  init F
  do

$$\underset{g \in sh(\Gamma)}{[]} \quad g : [T(g), B(g) \rightarrow R(g), \ ||_{v \in D(g)} \ v := F(g, v)]$$

$$\underset{g \in prv(\Gamma)}{[]} \ \textbf{prv} \quad g : [T(g), B(g) \rightarrow R(g), \ ||_{v \in D(g)} \ v := F(g, v)]$$

end component

Figure 1: The form of an RT-Community component

may be executed. If more than one enabled action exists, one may be selected non-deterministically. If an action is selected, the corresponding assignments are effected and the specified clocks are reset. If no action is selected, time passes, the component idles and all clocks are updated accordingly (i.e., the semantics is based on global time). However, three conditions must be met: (1) a private action may not be enabled infinitely often without being infinitely often selected (the fairness condition), (2) time may not pass if there is an enabled private action (the urgency condition), and (3) a shared action may not be disabled solely by the passage of time (the persistency condition).

If a component has a non-empty set of input variables, it can only be "executed" in some environment which provides values to the component's input variables. Such a component is called *open*; otherwise it is called *closed*. It will be made clear below how open components may be connected in order to form a closed system. However, in the definition of the formal semantics of RT-Community, we will include in the behavior of an open component all possible behaviors of its environment.

A model-theoretic semantics of RT-Community based on Time-Labelled Transition Systems will be given below.

It should be remarked that one of the latest versions of (untimed) Community ([9]) includes mechanisms for underspecification, such as lower and upper bounds for action data guards (i.e., safety conditions and progress conditions, respectively), and the partial specification of the effect of an action g on its write frame $D(g)$, by means of a boolean expression (involving primed variables, as is customary in other formalisms) instead of a parallel assignment. Furthermore, Community components may be parameterized by sorts and operations. These mechanisms will not be needed for the purposes of this paper, but they can be easily incorporated into RT-Community.

2.1 A Simple Example

We present a component for the "snooze" feature of an alarm clock. Component *snooze* is activated when the timekeeping component of the alarm clock (not shown) reaches the preset time, as indicated by action *firstRing*. This action sets the output variable *ringing* to true, a change that may be detected by a "bell" component (not shown either). If the user presses the "off" button at this

```
component snooze
    in          float initialInterval, float minimum
    out         bool ringing
    prv         float interval
    clocks      c
    init        ¬ringing ∧ interval = −1
    do
    [] firstRing:        → ringing := true || interval := initialInterval
    [] snooze:           ringing ∧ interval > minimum →
                            reset(c) || ringing := false || interval := interval/2
    [] off:              ringing → ringing := false || interval := −1
    [] prv timeout:      c == interval → ringing := true
end component
```

Figure 2: The *snooze* component

point, the alarm and the snooze component are turned off, as indicated by the *off* action. However, if the user presses the "snooze" button (action *snooze*), the alarm stops ringing, only to ring again after a preset time interval. This second ringing of the alarm is activated by the *snooze* component upon detecting the timeout (private action *timeout*). Now, if the user presses the "snooze" button this time, he will be allowed to sleep for an additional period with half the duration of the initial interval. This pattern repeats, with the interval being halved each time the alarm rings and the user presses the "snooze" button, until either the user presses the "off" button or the interval reaches a certain minimum duration (in this last case, the alarm will go on ringing until the user presses the "off" button).

The duration of the initial interval and the minimum duration are provided by the environment of the *snooze* component, as indicated by input variables *initialInterval* and *minimum*. The specification of the *snooze* component is given in Figure 2. There, time guards and data guards that have the constant truth value **true** are omitted, and the resetting of a clock c is indicated by the **reset**(c) instruction.

2.2 Semantics

Definition 2.1 (TLTS associated to a component) *Let $\mathbb{R}_{\geq 0}$ denote the set of all nonnegative reals. To each component P of the form shown in Figure 1 there corresponds a Time Labeled Transition System (TLTS – see [1, 12]) $T_P =< S_P, \Gamma_P, s_0, \longrightarrow >$, where*

- $S_P = \{(\mathcal{V}, \mathcal{C}) \mid \mathcal{V}$ is a valuation of V and $\mathcal{C}$ is a mapping from C to $\mathbb{R}_{\geq 0}\} \cup s_0$

  Given the algebra U with set of sorts S, $\mathcal{V}$ is an S-indexed family of mappings $\mathcal{V}_s : V_s \rightarrow U_s$, where V_s denotes the set of variables of sort s used by the component P. The initial state s_0 is a "special" valuation, defined below, where all variables contain undefined values.

- $\Gamma_P = shr(\Gamma) \cup prv(\Gamma) \cup \{\epsilon(d) | d \in \mathbb{R}_{\geq 0}\} \cup \{env,\ init\}$

Each transition labeled by $\epsilon(d)$ represents the passage of d time units. *env* is a name representing actions of the environment of the component. The name *init* represents the initial actions of the component, taking it from the initial state s_0 to states where all variables have been initialized (compare the "entry transitions" of the Action Diagrams of [6]).

- $s_0 = (\mathcal{V}_0, \mathcal{C}_0)$, where $\mathcal{V}_0$ assigns to each variable the undefined value $\perp$, not in the algebra U; $\mathcal{C}_0$ assigns to each clock the value zero.

The transition relation $\longrightarrow$ is defined as follows:

- For all data valuations $\mathcal{V}$,

$$(\mathcal{V}_0, \mathcal{C}_0) \stackrel{init}{\longrightarrow} (\mathcal{V}, \mathcal{C}_0) \quad \text{iff} \quad \mathcal{V} \models F$$

  where F is the formula specifying the initial state of the component. Note that F can only constrain the values of local variables; there are no constraints on the values of input variables at $(\mathcal{V}, \mathcal{C}_0)$, in order to include all of their possible initializations by the environment.

- For all data valuations $\mathcal{V}$ and $\mathcal{V}'$, and for all clock valuations $\mathcal{C}$,

$$(\mathcal{V}, \mathcal{C}) \stackrel{env}{\longrightarrow} (\mathcal{V}', \mathcal{C}) \quad \text{iff} \quad \mathcal{V}(v) = \mathcal{V}'(v) \text{ for all } v \in loc(V)$$

  Actions of the environment cannot change the value of local variables of the component.

- For all $g \in shr(\Gamma) \cup prv(\Gamma)$, $(\mathcal{V}, \mathcal{C}), (\mathcal{V}', \mathcal{C}') \in S_P$,

$$\begin{array}{lll} (\mathcal{V}, \mathcal{C}) \stackrel{g}{\longrightarrow} (\mathcal{V}', \mathcal{C}') & \text{iff} & \mathcal{V} \models B(g) \quad \wedge \\ & & \mathcal{C} \models T(g) \quad \wedge \\ & & \mathcal{V}'(v_1) = [\![F(g, v_1)]\!]^{\mathcal{V}}, ..., \mathcal{V}'(v_m) = [\![F(g, v_m)]\!]^{\mathcal{V}} \\ & & \qquad \text{where } D(g) = \{v_1, ..., v_m\} \quad \wedge \\ & & \mathcal{V}'(v) = \mathcal{V}(v) \text{ for all } v \in \text{ loc}(V) - D(g) \quad \wedge \\ & & \mathcal{C}' = \mathcal{C}[0/c_1, ..., 0/c_n], \text{ where } R(g) = \{c_1, ..., c_n\} \end{array}$$

  That is, each shared or private action is enabled precisely at those states where its data guard and time guard hold, and the effects of the assignments and resetting of clocks by the action are materialized in the target state.

  Note that input variables may not be modified directly by the execution of g, but they may change under some action of the environment which is executed in parallel to g; therefore, their values are unconstrained in the target state. The clocks of a component, however, cannot be modified by its environment. The constraints on clock values indicate that the execution of an action is instantaneous.

- For all $(\mathcal{V}, \mathcal{C}), (\mathcal{V}', \mathcal{C}') \in S_P$,

$$(\mathcal{V}, \mathcal{C}) \stackrel{\epsilon(0)}{\longrightarrow} (\mathcal{V}', \mathcal{C}') \quad \text{iff} \quad \mathcal{V} = \mathcal{V}' \wedge \mathcal{C} = \mathcal{C}'$$

  Each state has an idling transition to itself, with no elapsed time.

- For all $d \in \mathbb{R}_{\geq 0} - \{0\}, (\mathcal{V}, \mathcal{C}) \in S_P$,

$$(\mathcal{V},\mathcal{C}) \xrightarrow{\epsilon(d)} (\mathcal{V},\mathcal{C}+d) \quad \text{iff} \quad \begin{array}{l} \text{there are no } g \in prv(\Gamma), s \in S_P \text{ such that} \\ (\mathcal{V},\mathcal{C}) \xrightarrow{g} s \\ \text{and} \\ \text{for all } g \in shr(\Gamma) \\ (g \text{ enabled at } (\mathcal{V},\mathcal{C}) \Rightarrow g \text{ enabled at } (\mathcal{V},\mathcal{C}+d)) \end{array}$$

  where $\mathcal{C}+d$ is the valuation that assigns to each clock c the value $\mathcal{C}(c)+d$. That is, the delay action $\epsilon(d)$ does not alter any data variables (local or input), but modifies clock values as expected.

  Two provisos on the passage of time: (1) private actions are considered *urgent* – when enabled, they must not be forced to wait (cf. a timed version of CCS in [12]); and (2) shared actions are considered *persistent* – when enabled, they may not be disabled by the passage of time (but they may be disabled when their data guards are falsified).

Definition 2.2 (Computations of a TLTS)
A computation of a TLTS $T =< S, \Gamma, s_0, \longrightarrow >$ *is an infinite sequence* $(s_i, g_i)_{i \in \mathbb{N}}$ *with* $s_i \in S$ *and* $g_i \in \Gamma$ *satisfying the following conditions:*

Initiality: The state s_0 of the computation is the initial state s_0 of T.

Consecution: For all $i \in \mathbb{N}$, $s_i \xrightarrow{g_i} s_{i+1}$ is a transition of T.

Progress of time: Let $elapsed(i) = \begin{cases} d & \text{if } g_i = \epsilon(d) \\ 0 & \text{otherwise} \end{cases}$

For all $d \in \mathbb{R}_{\geq 0}$ there exists $t \in \mathbb{N}$ such that $\sum_{i=0}^{t} elapsed(i) > d$.

Strong fairness for private actions: Any private action that is infinitely often enabled must be taken infinitely often.

Given these definitions, the behavior of an RT-Community component P is represented by all the computations of S_P.

3 Composing RT-Community Components

We use basic concepts from Category Theory ([8, 10, 14]) to define how composition is done in RT-Community. The use of Category Theory allows us to describe the interaction of components in an abstract fashion, thus establishing the essentials of their coordination in such a way that RT-Community components may be replaced by specifications in other formalisms (e.g. a temporal logic with time-bounded operators such as MTL – see [4]), a desirable feature when our ultimate goal is to promote the interoperability of formalisms.

3.1 The Categories of Signatures and Components

Formally, an RT-Community component is a pair (Σ, Δ) with Σ the signature and Δ the body of the component.

Definition 3.1 (Signature of a component) *An RT-Community signature is a tuple* $\Sigma =< V, C, \Gamma, tv, ta, D, R >$, *where* V *is a finite set of variables,* C *is a finite set of clocks,* Γ *is a finite set of action names,* $tv : V \longrightarrow \{in, out, prv\}$ *is the typing function for variables,* $ta : \Gamma \longrightarrow \{shr, prv\}$ *is the typing function for action names,* $D : \Gamma \longrightarrow 2^{loc(V)}$ *is the write frame function for action names, and* $R : \Gamma \longrightarrow 2^C$ *is the reset clock function for action names.*

Definition 3.2 (Body of a component) *The body of an RT-Community component with signature* Σ *is a tuple* $\Delta =< T, B, F, I >$, *with* $T : \Gamma \longrightarrow \mathcal{PROP}(C)$ *the time guard function for action names,* $B : \Gamma \longrightarrow \mathcal{PROP}(V)$ *the data guard function for action names,* $F : \Gamma \longrightarrow (loc(V) \longrightarrow \mathcal{TERM}(V))$ *the assignment function for action names, and* I *a formula over* $loc(V)$ *specifying the initial state(s) of the component. Here,* $\mathcal{PROP}(C)$ *denotes the set of boolean propositions over clocks,* $\mathcal{PROP}(V)$ *denotes the set of boolean propositions over variables and* $\mathcal{TERM}(V)$ *is the set of terms of the term algebra of the data types involved. Function* F *must respect sorts when assigning terms to variables.*

We now define the category *Sign* of RT-Community signatures. A component may already be the result of the composition of smaller components. The morphisms in *Sign*, then, are defined so as to capture the relationship between (the signature of) a program P_1 and (the signature of) the system P_2 in which P_1 participates as a component. This embedding (or superposition) viewpoint is usual when modeling concurrent systems through Category Theory: Winskell [14] defines a similar category for transition systems, which differs from ours in the fact that his morphisms capture a relationship inverse to the one we define here: a morphism from a system S_1 to a system S_2 in his category means that S_1 is a composite system with S_2 as one of its components.

In what follows, then, it is helpful to keep in mind that Σ_1 is the signature of a component (referred to as "the component") that is embedded in a system (referred to as "the system") whose signature is Σ_2.

Definition 3.3 (Category of signatures) Sign *is the category that has signatures of RT-Community as objects; a morphism* $\sigma : \Sigma_1 \longrightarrow \Sigma_2$ *in* Sign *is a triple* $< \sigma_v, \sigma_c, \sigma_a >$ *defined as follows:*

$\sigma_v : V_1 \longrightarrow V_2$ is a total function such that

- For all $v \in V_1$, $sort_2(\sigma_v(v)) = sort_1(v)$. Variables of the component are mapped to variables of the system in such a way that sorts are preserved;
- For all $o, i, p \in V_1$, $o \in out(V_1) \Rightarrow \sigma_v(o) \in out(V_2)$, $i \in in(V_1) \Rightarrow \sigma_v(i) \in in(V_2) \cup out(V_2)$, $p \in prv(V_1) \Rightarrow \sigma_v(p) \in out(V_2)$. The nature of each variable is preserved, with the exception that input variables of the component may become output variables of the system. This is because, as will be seen below, an input variable of a component may be "connected" to an output variable of another component; when this happens, the resulting variable must be considered an output variable of the system: its value can be modified by the system, and it remains visible to the environment (which, however, cannot modify it).

$\sigma_c : C_1 \longrightarrow C_2$ is a total, injective function from the clocks of the component to the clocks of the system. In other words, all clocks of the component must retain their identity in the system.

$\sigma_a : \Gamma_2 \longrightarrow \Gamma_1$ is a partial function from the actions of the system to the actions of the component. σ_a is partial because an action of the system may or may not correspond to an action of the component; i.e., if the component does not participate in a given action g of the system, then $\sigma_a(g)$ is undefined. Furthermore, note the contravariant nature of σ_a compared to σ_v and σ_c: because each action of the system can involve at most one action of the component, and each action of the component may participate in more than one action of the system, the relation must be functional from the system to the component. Besides, σ_a must satisfy the following conditions ($D(v)$ for a variable v denotes the set of actions having v in their domain):

- For all $g \in \Gamma_2$ for which $\sigma_a(g)$ is defined, $g \in shr(\Gamma_2) \Rightarrow \sigma_a(g) \in shr(\Gamma_1)$ and $g \in prv(\Gamma_2) \Rightarrow \sigma_a(g) \in prv(\Gamma_1)$. An action of the system is of the same nature (shared or private) as the component action involved in it.
- For all $g \in \Gamma_2$ for which $\sigma_a(g)$ is defined, and for all $v \in loc(V_1)$, $v \in D_1(\sigma_a(g)) \Rightarrow \sigma_v(v) \in D_2(g)$ and $g \in D_2(\sigma_v(v)) \Rightarrow \sigma_a(g) \in D_1(v)$. If a component variable is modified by a component action, then the corresponding system variable is modified by the corresponding system action. Besides, system actions where the component does not participate cannot modify local variables of the component.

  The following item states analogous conditions for clocks of the component:
- For all $g \in \Gamma_2$ for which $\sigma_a(g)$ is defined, and for all $c \in C_1$, $c \in R_1(\sigma_a(g)) \Rightarrow \sigma_c(c) \in R_2(g)$ and $g \in R_2(\sigma_c(c)) \Rightarrow \sigma_a(g) \in R_1(c)$.

We define the category of RT-Community components, with whole components as objects and special signature morphisms between them.

Definition 3.4 (Category of components) *Comp is the category that has components of RT-Community as objects; a morphism $\sigma : (\Sigma_1, \Delta_1) \longrightarrow (\Sigma_2, \Delta_2)$ in Comp is a signature morphism $\sigma : \Sigma_1 \longrightarrow \Sigma_2$ satisfying the following conditions:*

- For all actions g in Γ_2 with $\sigma_a(g)$ defined, we have $\Phi \models B_2(g) \rightarrow \bar{\sigma}(B_1(\sigma_a(g)))$ and $\Phi \models T_2(g) \rightarrow \bar{\sigma}(T_1(\sigma_a(g)))$, where Φ is a suitable axiomatization of the specification of the data types involved, and $\bar{\sigma}$ is the extension of σ_v to the language of terms and propositions. The behavior of the system (Σ_2, Δ_2) is such that an action g of the system cannot disrespect the data and time guards of the corresponding action $\sigma_a(g)$ of the component; i.e., the system can only strengthen said guards.
- $\Phi \models I_2 \rightarrow \bar{\sigma}(I_1)$, with Φ and $\bar{\sigma}$ as above. The initial state of the component must be implied by the initial state of the system.
- For all actions g in Γ_2 with $\sigma_a(g)$ defined, and for all local variables v in $D_1(\sigma_a(g))$, we have $F_2(g)(\sigma_v(v)) = \bar{\sigma}(F_1(\sigma_a(g))(v))$, where, as before, $\bar{\sigma}$ is the extension of σ_v to the language of terms and propositions. Recall that F is the function that assigns to each action g a mapping from the variables in the action's domain $D(g)$ to terms of the term algebra of the data types involved. This means that an action g of the system can only

assign to a local variable of the component the "translation" of the value that the corresponding action $\sigma_a(g)$ of the component does.

3.2 Channels and Configurations

In categorial terms, the composition of two components P_1 and P_2 that do not interact at all (either via input and output variables or via shared actions) is given by the coproduct $P_1 + P_2$ of the corresponding objects in the category $\underline{Comp}$. This is a relatively uninteresting way to compose two components: the set of variables of the resulting system is the disjoint union of the sets of variables of the components (an eventual name clash is seen as accidental, and resolved by qualifying variable names with, say, the component's name); likewise, the set of actions of the resulting system is the disjoint union of the sets of actions of the components.

In the more interesting case where there must be interaction between P_1 and P_2, the interaction is either in the form of the connection of variables or in the form of the synchronization of actions (or both). This interaction is specified using a third component, called a *channel*. Given components P_1 and P_2 and a channel P_c, the composition of P_1 and P_2 via P_c is represented by the diagram

$$P_1 \xleftarrow{\sigma_1} P_c \xrightarrow{\sigma_2} P_2$$

When certain conditions are met, such a diagram is called a *configuration*. In a configuration, morphisms σ_1 and σ_2 specify how P_1 and P_2 should interact, as discussed below:

Variables: The only variables that the channel P_c can contain are of the input kind, so they can be mapped to input or output variables of P_1 and P_2. Given an input variable v of P_c, we say that variables $\sigma_1(v)$ and $\sigma_2(v)$ of P_1 and P_2, respectively, are connected. If two input variables are connected, the result will be an input variable of the composite system. If an input variable and an output variable are connected, the result will be an output variable of the composite system. We do not allow two output variables to be connected (a diagram where this happens is not considered a well-formed configuration). Furthermore, private variables and clocks cannot be connected, so we do not allow the channel P_c to have private variables or clocks.

Actions: The only actions that the channel P_c can contain are of the shared kind. Furthermore, these actions must be "empty" in the sense that their time and data guards are the constant values **true**, they do not reset any clocks and do not modify any variables (i.e. their write frame is empty). This means that the channel P_c is a "neutral" component with no behavior of its own, serving only as a kind of "wire" for joining the actions of P_1 and P_2.

Given an action g_1 of P_1 and an action g_2 of P_2 having $\sigma_1(g_1) = \sigma_2(g_2) = g$ for g an action of P_c, we say that g_1 and g_2 are synchronized. When two actions are synchronized, the result will be a joint action of the composite system. This joint action will be of the shared kind, its data and time guards being the conjunction of the corresponding guards of g_1 and g_2, its effect being to reset the union of the sets of clocks reset by g_1 and g_2 and to perform the assignments in the union of the sets of assignments dictated by g_1 and g_2.

Formally, the system that results from the composition of P_1 and P_2 through channel P_c is the pushout (in the category $\underline{Comp}$) of the configuration above,

component *timekeeping*

in	**Time** *alarmTime*, **Time** *currentTime*	
out	**float** *snoozeInterval*, **float** *minimum*	
prv	**int** *ticksPerSec*, **Time** *now*, **boolean** *alarmOn*	
clock	c	
init	$snoozeInterval = 10 \wedge minimum = 1 \wedge ticksPerSec = ... \wedge \neg\ alarmOn$	
do		
[] *setTime*:		$\rightarrow now := currentTime \parallel$ **reset**(c)
[] *setAlarm*:		$\rightarrow alarmOn :=$ **true**
[] *ring*:		$alarmOn \wedge now == alarmTime \rightarrow$ **skip**
[] *alarmOff*:		$\rightarrow alarmOn :=$ **false**
[] **prv** *keepTime*:		$c == ticksPerSec \rightarrow now := now + 1 \parallel$ **reset**(c)

end component

Figure 3: The *timekeeping* component

written $P_1 +_{P_c} P_2$. In general, for a configuration diagram involving any (finite) number of channels and components, the resulting system is given by the colimit of the diagram. A configuration is said to be well-formed when it satisfies the conditions described above. For well-formed configurations, the colimit will always exist.

As discussed in works about the original, untimed version of Community (e.g. [9]), the requirement that channels have no behavior of their own gives rise to a close relationship between configuration diagrams in the category $\underline{Comp}$ of components and the corresponding diagrams in the category $\underline{Sign}$ of signatures. This relationship is materialized in the properties of the forgetful functor *sig* mapping each component (Σ, Δ) to its signature Σ. These properties confirm that specifying the interactions between components in a complex system can be done solely on the basis of the information provided by the signatures of such components. In practice, this allows us to use signatures as channels for simplifying our diagrams.

3.3 A Simple Example

In order to illustrate composition in RT-Community, we present the *timekeeping* component of the alarm clock discussed in Section 2.1 and show how the *timekeeping* and *snooze* components can be put together. The *snooze* component was shown in Figure 2. The *timekeeping* component is shown in Figure 3. There, it is assumed that a data type *Time* is available, and that adding 1 to a variable of this data type corresponds to adding one second to the time value it contains. Besides, the fact that an action performs no assignments and resets no clocks (like the *ring* action) is indicated by the abbreviation **skip**.

When composing the *timekeeping* and *snooze* components, we want to identify variables *snoozeInterval* and *initialInterval*, as well as variables *minimum* (in *timekeeping*) and *minimum* (in *snooze*), so that the input variables in *snooze* will contain the constant values provided by *timekeeping*.

Furthermore, we want to synchronize actions *ring* and *firstRing* so that *snooze* will be activated exactly when *timekeeping* detects that the current time equals the time the alarm has been set to ring. We also want to synchronize the *alarmOff* and *off* actions, meaning that when the user presses the "off" button, both the snooze and the alarm mechanisms are turned off.

Notice that the resulting composite system is still open, in the sense that there are still unconnected input variables: *currentTime* and *alarmTime* receive values given by the user when he or she wants to set the time or the alarm, operations that are made available by the shared actions *setTime* and *setAlarm*, respectively.

Further interaction with the environment is given by the following features:

- The *snooze* action of the *snooze* component remains as a shared action of the system; it must be synchronized with an action of the environment representing the pressing of the "snooze" button while the bell is ringing.
- The joint action (*alarmOff* | *off*) is a shared action of the system that must be synchronized with an action of the environment representing the pressing of the "off" button while the bell is ringing.
- The output variable *ringing* in the *snooze* component must be connected to an input variable of a component representing the actual bell mechanism.

4 Concluding Remarks

RT-Community, like untimed Community, lends itself to the specification of architectural connectors that express interaction of a more complex nature than in the examples presented here. Other features of the language include the capacity for underspecification, such as lower and upper bounds for action data guards (i.e., safety conditions and progress conditions, respectively), and the partial specification of the effect of an action g on its write frame $D(g)$, by means of a boolean expression (involving primed variables, as is customary in other formalisms) instead of parallel assignment.

When specifying the components of a real-time system, it may be appropriate to use formalisms of a higher level of abstraction than RT-Community. One example would be the use of a real-time temporal logic (e.g. MTL – [4]) to specify the behavior of a component. In fact, we expect that by using mappings between logics such as those described in [7], a wider range of formalisms may be employed in the specification of a single system, allowing for a situation of interoperability among logics and specification languages.

The adaptation of RT-Community to systems involving mobility and dynamic reconfiguration is the subject of current study. A similar goal is being pursued in relation to untimed Community ([13]), where graph rewriting is used to reflect runtime changes in the system. We are investigating an alternative approach, where channels may be passed between components to allow them to engage in new connections at execution time – a strategy similar to the one used in π-calculus ([11]).

RT-Community is currently being contemplated for the specification of hypermedia presentations, a domain where real-time constraints occur naturally.

References

[1] L. Aceto, A. Burgueño and K.G. Larsen, Modedl Checking via Reachability Testing for Timed Automata, BRICS Report Series RS 97–29 (1997).

[2] R. Allen and D. Garlan, A Formal Basis for Architectural connectors, ACM TOSEM, 6(3)(1997) 213–249.

[3] R. Alur and D.L. Dill, A Theory of Timed Automata, Theoretical Computer Science 126(1994), 183–235.

[4] E. Chang, Compositional Verification of Reactive and Real-Time Systems, PhD Thesis, Stanford University (1995).

[5] D. Gelernter and N. Carriero, Coordination Languages and their Significance, Comm. ACM 35, 2 (1992) 97–107.

[6] T.A. Henzinger, Z. Manna and A. Pnueli, Temporal Proof Methodologies for Timed Transition Systems, Information and Computation 112 (1994) 273–337.

[7] A. Martini, U. Wolter and E.H. Haeusler, Reasons and Ways to Cope with a Spectrum of Logics, in J. Abe and J.I. da S. Filho (eds.), Logic, Artificial Intelligence and Robotics (proc. LAPTEC 2001), series: Frontiers in Artificial Intelligence and Applications 71, IOS Press, Amsterdam (2001) 148–155.

[8] B. Peirce, Basic Category Theory for Computer Scientists, The MIT Press (1991).

[9] J.L. Fiadeiro and A. Lopes, Semantics of Architectural connectors, in M. Bidoit and M. Dauchet (eds), TAPSOFT'97, LNCS 1214, Springer-Verlag (1997) 505–519.

[10] P.B. Menezes and E.H. Haeusler, Teoria das Categorias para Ciência da Computação, Sagra Luzzato, UFRGS (2001).

[11] R. Milner, Communicating and Mobile Systems: the π-Calculus, Cambridge University Press (1999).

[12] Y. Wang, CCS + Time = An Interleaving Model for Real-Time Systems, in J.L. Albert and B. Monien and M. Rodríguez-Artalejo, editors, Automata, Languages and Programming, 18th International Colloquium, volume 510 of Lecture Notes in Computer Science, Madrid, Spain, 8-12 July 1991, Springer-Verlag, 217–228.

[13] M. Wermelinger and J.L. Fiadeiro, Algebraic Software Architecture Reconfiguration, in Software Engineering–ESEC/FSE'99, volume 1687 of LNCS, Springer-Verlag (1999) 393–409.

[14] G. Winskell and M. Nielsen, Categories in Concurrency, in A.M. Pitts and P. Dybjer (eds.), Semantics and Logics of Computation, Cambridge University Press (1997) 299–354.