Specification of Real-Time Interaction Constraints

Brian Nielsen
Aalborg University
Dpt. of Computer Science
Fredrik Bajersvej 7E
DK-9220 Aalborg, Denmark
Email: bnielsen@cs.auc.dk

Shangping Ren Gul Agha
University of Illinois at Urbana-Champaign
Dpt. of Computer Science
1304 W. Springfield Av.
Urbana, Illinois 61801, U.S.A
Email: { ren | agha }@cs.uiuc.edu

Abstract

We present a coordination language and its semantics for specification and implementation of object-oriented real-time systems. Real-time systems operate under real-time constraints, and our language supports expression thereof. In our language, a system is modeled by two separate but complementary descriptions: A collection of objects define the system's structure and functional behavior, and a set of interaction constraints define how these objects may interact. Our language thereby supports development of real-time systems by enabling objects build in isolation or reused from other systems to be composed via interaction constraints. We use the Actor model to describe objects and the concept of real-time synchronizers to describe interaction constraints.

Our model is accompanied by a formal semantics that precisely defines what real-time constraints means, and what constitutes a program's correct real-time behaviors. The semantics defines how the system may evolve in the real-time domain, and what progress guarantees the language makes. We briefly discuss implementation problems and potential solutions.

1. Introduction

Many applications like computer control systems that monitor and control physical equipment require operation under strict time constraints. Typical applications monitor the state of the equipment through sensors and react by affecting actuators in accordance with *time constraints*; e.g., before or after a given real-time time bound. We propose a concurrent object-oriented language for the specification and implementation of real-time systems.

An object-oriented real-time system is modeled by a collection of active objects executing concurrently. These objects interact by exchanging messages containing information about their local states. However, it is necessary to control the interaction between objects to ensure satisfaction of time and synchronization constraints. A time constraint could be that a controller object must receive sensor data from a sensor-object every 20 milli-seconds. Our model can be viewed as a coordination language that is concerned with controlling the dynamic interaction *among* objects.

A key point of our model is that the necessary synchronization and time constraints are specified separately from the basic functional behavior of a system. The benefits of this separation of concerns are twofold. First, objects with basic functional behavior can be developed individually and later be composed with other individually developed or existing objects. These objects are "glued" together by our interaction constraints. Second, when no interaction constraints are hardwired in the internals of objects it becomes easier to reuse objects in other applications where different interaction constraints apply. Thus, interaction constraints are installed "on top" of ordinary objects, and actively enforce programmer specified constraints, see Figure 1.

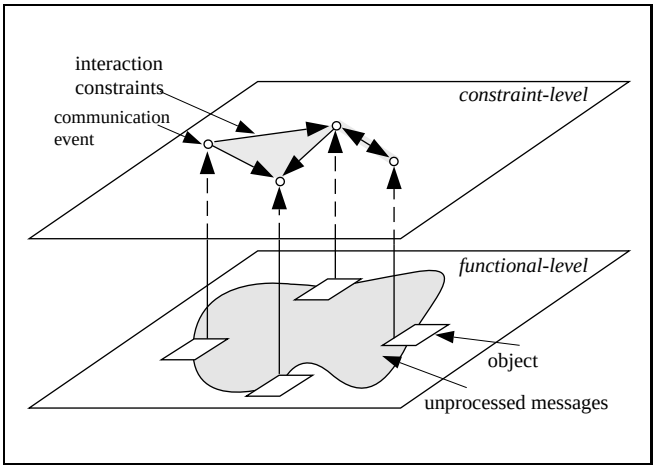

Figure 1: Separation of constraints and functionality

Interaction constraints are expressed in terms of enabling conditions on communication events occurring on the interface of objects. These events constitute the observable behavior of a system. What goes on inside an object is encapsulated, and cannot be constrained. Specifically, a collection of *synchronizer-entities* constrain by delaying or accelerating message invocations. We use the object-oriented *Actor*-model to describe objects.

The idea of separating functional behavior and interaction policies for Actors were first proposed by Frølund in [7], but only logical synchronization constraints could be specified. Later Ren in [13] made a first proposal for a dual language where real-time constraints could be expressed. Our work is a continuation of this line of research where we have emphasized a formal treatment of the model. We define a destilled language, RT-Synchronizers$^-$, and provide an operational semantics that defines the real-time behavior of a constrained actor-program. Related work that permit separate specification of real-time and synchronization constraints is [5] which proposes the composition filter model. Real-time input and output filters declared in an extended interface enable the specification of time bounds on method executions. Among many, one difference is that RT-Synchronizers$^-$ takes a global view of a collection of objects rather than of a single object. To our knowledge composition filters has not been defined formally.

Section 2 introduces and exemplifies our model. Section 3 provides the formal definition. Finally, in Section 4 we discuss our implementation ideas.

2. Specification of interaction constraints

In the Actor model [1, 2, 4] distributed computing entities (hardware or software) are modeled as abstract self contained objects called actors. An actor encapsulates a state, provides a set of public methods, and potentially invokes public methods in other objects by means of message passing. Unlike many object oriented languages, message passing is *non-blocking* and *buffered*. This means that when an actor sends a message it immediately resumes its computation without waiting for, or getting reply from, the receiver. Further, messages sent but not yet processed by the receiver are conceptually buffered in a mailbox at the receiver. Here the receiver picks them and processes them sequentially. In addition, actors execute *concurrently*. An actor system is illustrated in Figure 2.

An actor is identified by a unique name, called a mail-address. This can be bound to state variables of type *actor reference*. To send a message an actor executes the **send** a, cv primitive. a contains the mail address of the target actor (possibly the actor itself), and cv is the value passed. In general cv encodes information about which method to be invoked along with its parameters. It is possible to communicate mail-addresses through messages thus allowing dynamic configuration of the communication topology.

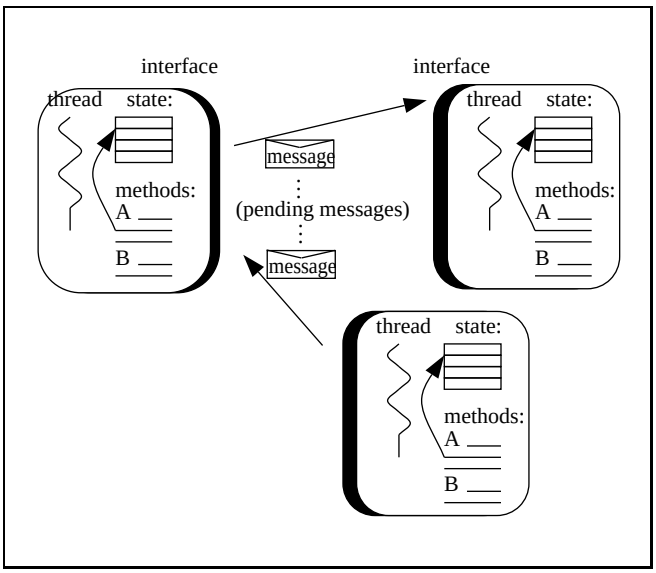

Figure 2: Illustration of an actor system

A *synchronizer* is an object that intercepts messages exchanged between actors and invokes these according to user specified constraints which expresses real-time or ordering constraints on pairs of message invocations. The messages of interest are captured by means of *patterns* which essentially are predicates over message contents and synchronizer state. The structure of a RT-Synchronizers$^-$ declaration is given in Figure 3. It consists of 4 parts: A set of instantiation parameters, declarations of local variables, a set of constraints, and a set of triggers.

$\textbf{synchronizer}\ (a_1, \ldots, a_n)\{$
$StateDeclaration$

$p_{11} \Rightarrow p_{21} \sim y_1$
$\vdots$
$p_{1n} \Rightarrow p_{2n} \sim y_n$

$p_1 : \overline{x} := \overline{exp}$
$\vdots$
$p_k : \overline{x} := \overline{exp}$
$\}$

Figure 3: Structure of RT-Synchronizers$^-$. $\sim \in \{\succ, \prec\}$

A constraint has one of the forms:

$p_1 \Rightarrow p_2 \prec y$**:** Here p_1 and p_2 are message patterns and y is a variable or constant with positive real value. Let $a_1(cv_1)$ and $a_2(cv_2)$ be message invocations matching

p_1 and p_2 respectively. This constraint then states that after an $a_1(cv_1)$ has occurred, an $a_2(cv_2)$ must follow before y time units. We say that $a_1(cv_1)$ is the firing event causing a demand for $a_2(cv_2)$.

$p_1 \Rightarrow p_2 \succ y$**:** This expresses that after $a_1(cv_1)$ occurs at least y time units must pass before $a_2(cv_2)$ is permitted.

In both cases there are no constraints on $a_2(cv_2)$ until $a_1(cv_1)$ fires. A pattern has the form $x_1(x_2)\mathbf{when}\ b$, where b is a boolean predicate (guard) over the message parameter x_2 and the state of the synchronizer. x_1 is a state variable containing an actor address. Intuitively, a message satisfies a pattern if it is targeted at x_1 and the boolen predicate evaluates to true. If a message satisfies a pattern, the invocation is affected by a constraint which must then be satisfied before the invocation may take place. When a constraint forbids the invocation of a message it is buffered until a later time when the constraint enables it. A disabled message may become enabled when a delay has expired or when the synchronizer changes state through a trigger operation.

The trigger section specifies how the synchronizer's state variables change when a message invocation satisfies a given pattern. Specifically, the assignments of the trigger $p : \overline{x} := \overline{exp}$ is executed when a message satisfying p is invoked. The synchronizers can thus adapt to the system's current state of affairs. To promote modularity of the interaction constraints these can be given as a *collection* of synchronizer objects conceptually executing concurrently.

2.1. Example 1: Steam boiler

The program in Figure 4 describes part of a simple boiler control system consisting of a pressure sensor, a controller, and a valve actuator. When requested the pressure sensor sends a message containing its pressure value back to its customer (the controller). Based on the pressure value, the controller computes an updated steam-valve position. The controller repeats this procedure periodically (every 20 time units plus/minus some tolerance) by sending itself a message. The controller must receive sensor data from the pressure sensor within 10 time units meassured from the start of the period, and it must update the steam valve position no later than 5 time units after receiving sensor data.

2.2. Example 2: Time Bounded Buffer

The example given in Figure 5 shows a bounded buffer (queue) where each element must be removed 20 time units after it has been inserted. In addition, the usual restrictions of not putting on a full buffer and not getting from an empty buffer are enforced. Note that the code uses a shorthand, $\mathbf{disable}\ p$, to temporarily prevent messages matching the pattern p from being invoked. $\mathbf{disable}\ p$ can be written as $e_0 \Rightarrow p \succ \infty$, where e_0 is a pattern assumed to be fired at system startup time.

```
actor pressureSensor () {
  real value;
  method read(actorRef customer) {
    send customer.reading(value);
  }
}
actor steamValve () { ... } // unspecified
actor controller (actorRef sensor,valve) {
  method loop() {
    send self.loop();
    send sensor.read(self);
  }
  method reading(real pressure) {
    newValvePos=computeValvePos(pressure);
    send valve.move(newValvePos);
  }
}

synchronizer boilerConstraints (actorRef:
                                  controller,valve) {
  //periodic loop:
  controller.loop ⇒ controller.loop ≺ 20+ϵ
  controller.loop ⇒ controller.loop ≻ 20-ϵ
  //deadline on reading:
  controller.loop ⇒ controller.reading ≺ 10
  //deadline on move:
  controller.reading ⇒ valve.move ≺ 5
}
```

Figure 4: Steam boiler

```
synchronizer bbConstraints (actorRef: q) {
  int n=0; // no of elements in queue
  q.put ⇒ q.get ≺ 20; // timebound on get
  disable q.get when n ≤ 0; //buf empty?
  disable q.put when n ≥ maxBufSz; //buf full?
  q.put: n++;
  q.get: n−−;
}
```

Figure 5: Bounded buffer with time constraints

2.3. Example 3: Rate Control

The example illustrates how rate control can be described, see Figure 6. At most 20 move operations can safely be performed on an actuator in any time window of 30 time units. The example uses an *event generator actor* to produce message invocations which the synchronizer use

to change state at certain time-points. An event generator actor does not add any functionality per se, but is necessary for the proper functioning of the synchronizer. This programming technique cancels the need for special internal event concept in RT-Synchronizers$^-$.

```
actor actuator { method move() { ... }}
actor eventGenerator {
   method timeOut() { send self.timeOut(); }
}
synchronizer rateControll (actorRef: actuator,
                                    eventGen) {
   int credit=20; // max no of events in window
   //timeOut 30 tu's after move:
   actuator.move ⇒ eventGen.timeOut ≺ 30;
   actuator.move ⇒ eventGen.timeOut ≻ 30;
   //event permitted?
   disable actuator.move when credit ≤ 0;
   //timeOut must be after move!
   disable eventGen.timeOut when credit ≥ 20;
   actuator.move: credit−−;
   eventGen.timeOut: credit++;
}
```

Figure 6: Rate control

3. Formal Definition

In the previous section we explained our model informally, and here we turn to its formal definition. The formal model defines the permissible behavior of a constrained actor program, and this is crucial in determining which executions on a physical machine will be considered correct. The separation of functionality and constraints is maintained in the formal definition, and this enables the semantics for Actors and RT-Synchronizers$^-$ to be given as independent transition systems. The meaning of a program composed of actors and synchronizers can then be given afterwards by putting the two transition systems in "parallel".

3.1. Semantics of Actors

We define a transition system κ for the actor-language. This defines how the state of the actor system changes when a primitive operation is performed, thus giving an abstract interpretation. The actor semantics presented here is inspired by the work of [4] where additional information can be found, but our is imperative in style, whereas [4] is applicative.

Our semantic model abstracts away the notion of methods. Instead, each actor has a single behavior—a sequence of statements—that it applies to every incoming message. When the actor has completed processing a message it executes the **ready**-statement[1] to indicate that it is ready to accept a new message.

The state of an actor system is represented by a configuration which can be thought of as an instantaneous snapshot of the system state made by a conceptual observer. It is modeled as a pair $\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle$ where α represents actor-states, and μ is the set of pending messages. The α mapping maintains the state of all actors in the system. An actor state holds the execution state of an actor: the values of its state-variables, and how far the actor has come in its computation. An actor state is written $[E \vdash b]_a$. a is the actor's address, E is an environment (mapping from identifiers to their values) tracking the values of the state-variables, and b is the remainder of the actor's behavior. In each computation step the actor reduces the behavior until it reaches a $\mathbf{ready}(x)$ statement. This signifies that the actor a is waiting for an incoming message whose contents should be bound to x. When a message arrives the actor continues its execution. A message is a pair $\langle a \Leftarrow cv \rangle$ consisting of a destination actor-address a, and a value to be communicated cv.

The semantics is given in Figure 7. The **fun** transition defines the effect on system state when an actor performs an internal computation step, a reduction of an expression. The transition system $\longrightarrow_\lambda$ defines the semantics of the sequential language used to express actor behaviors. Since we do not rely on a specific language, we have omitted its definition.

The interpretation of **send** is given by the **snd**-rule. The newly sent message is added to μ.

Message reception is described by the **rcv** transition. When an actor executes a $\mathbf{ready}(x)$-statement it becomes ready to accept a new message in an environment with the updated state variables left by the previous processing. Also, the formal argument x is bound to the actual carried by the message. Finally, the message is removed from μ. It is exactly these receive transitions that are constrained by RT-Synchronizers$^-$. Other transitions are only affected indirectly.

By inspecting the semantics it should be clear that one cannot make any timing assertions about the execution of an actor program, as is required by real-time systems. To make this observation explicit we here briefly introduce time into the actor semantics.

Time can be added to transition systems by introducing a special set of *delay actions* written $\varepsilon(d)$ where d is a fi-

[1] In the classic Actor literature the **become** primitive is used instead of **ready**. When an actor executed a **become** it created a new anonymous actor to carry out the rest of its computation, and prepared itself to receive a new message. Thus, in the classic model, actors were multi-threaded, and tended to be extremely fine-grained. In recent literature [3], the simpler **ready** has replaced **become**, with essentially no loss of expressiveness. In addition we have, due to brevity, omitted the semantic definition of dynamic actor creation.

$$\langle \textbf{fun} : a \rangle \quad \frac{E \vdash b \ \longrightarrow_\lambda \ E' \vdash b'}{\langle\!\langle\, \alpha\,,\, [E \vdash b]_a \mid \mu \,\rangle\!\rangle \longrightarrow_\kappa \langle\!\langle\, \alpha\,,\, [E' \vdash b']_a \mid \mu \,\rangle\!\rangle}$$

$$\langle \textbf{snd} : a, \langle a' \Leftarrow cv \rangle \rangle \quad \langle\!\langle\, \alpha\,,\, [E \vdash \mathbf{send}(a', cv); b]_a \mid \mu \,\rangle\!\rangle \longrightarrow_\kappa \langle\!\langle\, \alpha\,,\, [E \vdash b]_a \mid \mu\,,\, \langle a' \Leftarrow cv \rangle \,\rangle\!\rangle$$

$$\langle \textbf{rcv} : a, \langle a \Leftarrow cv \rangle \rangle \quad \langle\!\langle\, \alpha\,,\, [E \vdash \mathbf{ready}(x); b]_a \mid \mu\,,\, \langle a \Leftarrow cv \rangle \,\rangle\!\rangle \longrightarrow_\kappa \langle\!\langle\, \alpha\,,\, [E[x \mapsto cv] \vdash b]_a \mid \mu \,\rangle\!\rangle$$

Figure 7: Configuration transitions $\longrightarrow_\kappa$

nite positive real-valued number representing the passage of d time-units. The idea is that system execution can be observed by alternatingly observing a set instaneous transitions and observing a delay. In [11] this idea was termed the *two-phase functioning principle*: System state evolves alternatingly by performing a sequence of instantaneous actions and by letting time pass.

By adding the rule: $\langle\!\langle\, \alpha \mid \mu \,\rangle\!\rangle \xrightarrow{\varepsilon(d)}_\kappa \langle\!\langle\, \alpha \mid \mu \,\rangle\!\rangle$, we extend $\longrightarrow_\kappa$ transition relation with the ability of letting time pass. The rule states that any actor configuration is always able to delay of some (finite) amount of time. The consequence is that one cannot tell how long time an actor program takes to finish; indeed the period between any pair of actions is indeterminate. This is a very reasonable model for untimed concurrent programs, where no assumptions on the relative ordern or timing of events should be made. However, a language with this semantics is unideal for real-time system because one can from the code only make assertions about eventually properties, not about bounded timing. A real-time programming language should make this possible, and its semantics should define when and by how much time can advance.

3.2. RT-Synchronizers$^-$ Semantics

We start by defining semantics for single constraints ($\longrightarrow_\gamma$ transition system), and thereafter proceed to a synchronizer object ($\longrightarrow_\sigma$ transition system), which essentially is a state plus a collection of constraints and triggers. The state variables of a synchronizer will be represented by an environment V mapping identifiers to their values. Constraints and patterns are evaluated in this environment.

Recall that a constraint has the form $p_1 \Rightarrow p_2 \sim y$. Whenever an invocation matches p_1 the constraint fires and thereby creates a new demand instance for an invocation matching p_2. Such a *demand* will semantically be represented by the triple $p_2 \sim d$, where d is a real number denoting the deadline or release time of p_2, depending on $\sim$. d is initialized with the value of state variable y,$V(y)$, when fired.

Since a constraint can fire many times successively, a constraint may induce many outstanding demand instances. The state of a single constraint is therefore represented as a *constraint configuration* $\langle \chi \mid \xi_\sim \rangle$ where $\xi_\sim$ stands for the (static description of a) constraint of the form $p_1 \Rightarrow p_2 \sim y$, and χ is a multi-set of demands instantiated from the static description $\xi_\sim$. The semantic rules are shown in Figure 8.

The function c_s determines whether the pattern of a demand instance is satisfied, and if so, removes it from the demand instance set. If the pattern is not satisfied the demand is maintained. Similarly, the function c_f determines whether or not the constraint fires and therefore whether or not to add a new demand instance. Thus the **Sat**-rules ensure that whenever a constraint fires, a demand (c_f) is added to χ. Also, whenever a demand (c_s) is satisfied, it is removed from χ. Due to the possibility of a single message matching both p_1 and p_2 the **Sat**-rules are prepared to both satisfy and fire a demand. The demand instance to be removed is chosen non-determistically; this gives the implementation the greatest freedom to choose the demand it finds the most appropriate, e.g., the one with the tightest deadline.

The passage of time is controlled by the **Delay**-rule such that the elapsed amount of time (e) is subtracted from d_i in each demand $p_i \sim d_i$. This is written $\chi \ominus e$. Thus for $p \succ d$, d is the amount of time that *must* pass before p is enabled. p will be enabled when d is less than 0. This requirement is enforced by the c_s function of the $\langle \mathbf{Sat}_\succ : a(cv) \rangle$ rule. For $p \prec d$, d is the amount of time that *may* pass before p will be disabled. p would be disabled if d was is less than 0. However the $\langle \mathbf{Delay}_\prec : e \rangle$ prevents time from progressing that much.

In effect, the delay rule ensures that deadline constraints are *always* satisfied in the semantics. This corresponds to the declarative meaning one would expect from a constraint: one that must be enforced. Without this strict definition our constraints would degenerate to merely assertions, and not convey its intended meaning. An actual language im-

$$\langle \mathbf{Sat}_{\prec} : a(cv) \rangle \quad c_s = \begin{cases} \emptyset & if\, a(cv) \models p_2 \\ p_2 \prec d' & otherwize \end{cases} \quad c_f = \begin{cases} p_2 \prec V(y) & if\, a(cv) \models p_1) \\ \emptyset & otherwize \end{cases}$$
$$\langle \chi \uplus p_2 \prec d' \mid \xi_{\prec} \rangle \xrightarrow{a(cv)}_{\gamma} \langle \chi \uplus c_f \uplus c_s \mid \xi_{\prec} \rangle$$

$$\langle \mathbf{Sat}_{\succ} : a(cv) \rangle \quad c_s = \begin{cases} \emptyset & if\, a(cv) \models p_2 \wedge d' \leq 0 \\ p_2 \prec d' & otherwize \end{cases} \quad c_f = \begin{cases} p_2 \succ V(y) & if\, a(cv) \models p_1) \\ \emptyset & otherwize \end{cases}$$
$$\langle \chi \uplus p_2 \succ d' \mid \xi_{\succ} \rangle \xrightarrow{a(cv)}_{\gamma} \langle \chi \uplus c_f \uplus c_s \mid \xi_{\succ} \rangle$$

$$\langle \mathbf{Sat}_{\emptyset} : a(cv) \rangle \quad c_f = \begin{cases} p_2 \sim V(y) & if\, a(cv) \models p_1) \\ \emptyset & otherwize \end{cases}$$
$$\langle \emptyset \mid \xi_{\sim} \rangle \xrightarrow{a(cv)}_{\gamma} \langle \emptyset \uplus c_f \mid \xi_{\sim} \rangle$$

$$\langle \mathbf{Delay}_{\sim} : e \rangle \quad \forall p_2 \prec d_i \in (\chi \ominus e).d_i \geq 0$$
$$\langle \chi \mid \xi_{\sim} \rangle \xrightarrow{\varepsilon(e)}_{\gamma} \langle \chi \ominus e \mid \xi_{\sim} \rangle$$

$$a(cv) \models x_1(x_2) \mathbf{when}\, b =_{def} a = V(x_1) \wedge b(V[x_2 \mapsto cv])$$

Figure 8: Semantics for single constraints $\longrightarrow_{\gamma}$ where $\sim \in \{\succ, \prec\}$

plementation may not always be able to give this guarantee, neither statically or dynamically, due to the computational complexity of finding feasible schedules for general constraints. Conflicting constraints that have no solutions should be detected as part of the compilers static program check. Ren has in [12] showed how RT-Synchronizers$^-$-constraints can be mapped to linear inequality systems for which polynomial time algorithms exist for detecting solvability.

The following transition sequence illustrates application of the transition rules for a constraint:

$$\begin{array}{ll} \langle \emptyset \mid p_1 \Rightarrow p_2 \prec 7 \rangle & \xrightarrow{a_1(cv)}_{\gamma} \\ \langle p_2 \prec 7 \mid p_1 \Rightarrow p_2 \prec 7 \rangle & \xrightarrow{\varepsilon(3)}_{\gamma} \\ \langle p_2 \prec 4 \mid p_1 \Rightarrow p_2 \prec 7 \rangle & \xrightarrow{a_1(cv)}_{\gamma} \\ \langle p_2 \prec 4, p_2 \prec 7 \mid p_1 \Rightarrow p_2 \prec 7 \rangle & \xrightarrow{\varepsilon(4)}_{\gamma} \\ \langle p_2 \prec 0, p_2 \prec 3 \mid p_1 \Rightarrow p_2 \prec 7 \rangle & \xrightarrow{a_2(cv)}_{\gamma} \\ \langle p_2 \prec 3 \mid p_1 \rightarrow p_2 \prec 7 \rangle & \xrightarrow{a_2(cv)}_{\gamma} \\ \langle \emptyset \mid p_1 \rightarrow p_2 \prec 7 \rangle & \end{array}$$

Given that the behavior of each single constraint is well defined it is easy to define the behavior a collection of constraints as found within a synchronizer. Essentially the individual constraints are conjuncted, i.e., we demand that all constraints agree on a given invocation. Similarly, they must all agree on letting time pass.

A synchronizer is represented by a *synchronizer configuration* $\langle \bar{\gamma} | V \rangle$ where $\bar{\gamma}$ is a *set* of constraint configurations (ranged over by γ). As previously stated V represents the state variables of a synchronzier and is a mapping from identifiers to their values. The necessary definition is shown in Figure 9. A synchronizer can engage in a message reception $a(cv)$ or a delay $\varepsilon(e)$ only when this is permitted by every constraint. We have omitted the quite simple definition of the effect of triggers: V' is V simultaneously updated with the assignments in the matched triggers.

3.3. Combining Actors and RT-Synchronizers$^-$

The previous subsections gave the meaning of the actor and RT-Synchronizers$^-$ languages independently. The effect of constraining an actor program can now be defined here as a special form of parallel composition $\|$ that preserves the meaning of constraints. Call a collection of synchronizers for an interaction constraint system. An *interaction constraint system configuration* is written $((\sigma_1, \ldots, \sigma_n))$ where σ ranges over synchronizer configurations. The composition $\|$ of an actor configuration and an interaction constraint system configuration is defined in Figure 10.

$$\langle \textbf{Action} : \ell \rangle \quad \frac{\forall i \in [1..n].\gamma_i \xrightarrow{\ell}_{\gamma} \gamma_i'}{\langle \gamma_1, \ldots, \gamma_n | V \rangle \xrightarrow{\ell}_{\sigma} \langle \gamma_1', \ldots, \gamma_n' | V' \rangle}, \ell \in \{a(cv), \varepsilon(e)\}$$

Figure 9: Semantics for a synchronizer $\longrightarrow_{\sigma}$

Unaffected Actions

$$\frac{\langle\!\langle\, \alpha \mid \mu \,\rangle\!\rangle \xrightarrow{\ell}_{\kappa} \langle\!\langle\, \alpha' \mid \mu' \,\rangle\!\rangle \qquad \ell \in \{\langle \textbf{fun} : a \rangle, \langle \textbf{snd} : a, m \rangle, \langle \textbf{ready} : a \rangle\}}{\langle\!\langle\, \alpha \mid \mu \,\rangle\!\rangle \parallel ((\sigma_1, \ldots, \sigma_n)) \xrightarrow{\ell}_{\kappa\sigma} \langle\!\langle\, \alpha' \mid \mu' \,\rangle\!\rangle \parallel ((\sigma_1, \ldots, \sigma_n))}$$

Receive

$$\frac{\langle\!\langle\, \alpha \mid \mu \,\rangle\!\rangle \xrightarrow{\ell}_{\kappa} \langle\!\langle\, \alpha' \mid \mu' \,\rangle\!\rangle \qquad \bigwedge_{i \in [1..n]} \sigma_i \xrightarrow{a(cv)}_{\sigma} \sigma_i' \qquad \ell = \langle \textbf{rcv} : a, \langle a \Leftarrow cv \rangle \rangle}{\langle\!\langle\, \alpha \mid \mu \,\rangle\!\rangle \parallel ((\sigma_1, \ldots, \sigma_n)) \xrightarrow{\ell}_{\kappa\sigma} \langle\!\langle\, \alpha' \mid \mu' \,\rangle\!\rangle \parallel ((\sigma_1', \ldots, \sigma_n'))}$$

Delay

$$\frac{\bigwedge_{i \in [1..n]} \sigma_i \xrightarrow{\varepsilon(d)}_{\sigma} \sigma_i'}{\langle\!\langle\, \alpha \mid \mu \,\rangle\!\rangle \parallel ((\sigma_1, \ldots, \sigma_n)) \xrightarrow{\varepsilon(d)}_{\kappa\sigma} \langle\!\langle\, \alpha \mid \mu \,\rangle\!\rangle \parallel ((\sigma_1', \ldots, \sigma_n'))}$$

Figure 10: Combined behavior $\longrightarrow_{\kappa\sigma}$

Transitions unaffected by interaction constraints altogether are message sends and local computations. These only have effect on the actor configuration. Message invocations $\langle \textbf{rcv} : a, m \rangle$ are the interesting events affected by constraints. Note that the same invocation may be constrained by several synchronizers, and *all* must certify the invocation, i.e., synchronizers, like constraints, are composed conjunctively. The idea is that adding more synchronizers should further restrict the behavior of objects. A consequence of this is that the synchronizers also must agree on letting time pass.

The combined semantics define all correct transition sequences ($\longrightarrow^*_{\kappa\sigma}$). A transition sequence corresponds to one possible *schedule* of the implemented system (consisting of actors, constraints, operating system, runtime system, and hardware resources), and thus a primary task of the language implementation is to schedule events in the system such that the resulting schedule can be found in the program's semantics. Thus an actor||RT-Synchronizers$^-$ program can be viewed as a specification for the set possible systems.

Our approach to defining the semantics is inspired by the recent years research in formal specification languages for real-time systems, and the use of timed transition systems are borrowed from these languages. Often, the languages take the form of extended automata (Timed automata [6], Timed Graphs [6, 11]), or process algebras such as Timed CSP [16]. A different approach is to include a model of the underlying execution resources. This approach is taken in [15] and [19]. The resulting semantics includes an abstract model of the execution environment (number of CPU's, scheduler, execution time of assignments etc.). The process algebra Communicating Shared Resources (CSR) has been designed with the explicit purpose of modeling resources [8, 9]. A process always runs on some, possibly shared, resource. A set of processes can be mapped to different sets of resources, hence describing different implementations. Thus, these approaches model relative concrete system, rather than being specifications for a set of possible systems, as is our goal.

4. Implementation Strategy

Through the examples in Figures 4, 5, and 6, we have indicated that our language can be used as a specification or modeling language that defines the structure and permissible behavior of a computer system: hardware and system software that executes application software. However, we would also like to use our language as a high-level real-time programming language such that an RT-Synchronizers$^-$ specification can be compiled and executed. Our model is still to be implemented, and we shall here only discuss a

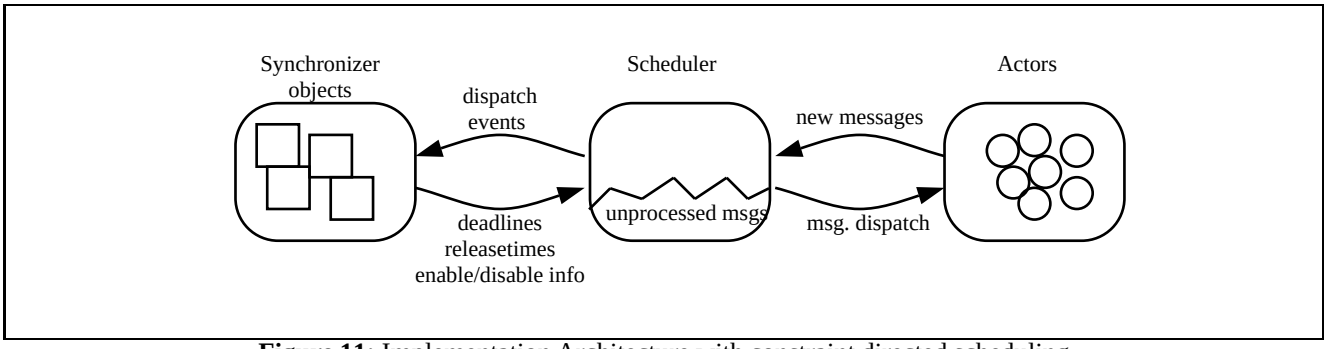

Figure 11: Implementation Architecture with constraint directed scheduling

possible implementation strategy.

Implementing our full model will be no easy task, but the difficulty is mostly related to the generality of the constraints that can be expressed, and less due to the separation of functionality and time constraints. We have identified three main tasks an implementation should address:

Scheduling: One challenge will be to find a scheduling strategy that satisfies as many deadline constraints as possible when the RT-Synchronizers$^-$-program is executed on a physical machine with limited resources. In addition, hard and firm real-time systems require an a priori guarantee (or at least a solid argument) that timing constraints will be satisfied on the chosen platform during runtime.

Constraint propagation: In RT-Synchronizers$^-$ the programmer need only specify end-to-end timing relations, not on all events along the causality path: Assume that actor a receives a message $m1$; a then responds with a message $m2$ to actor b which in turn sends a message $m3$ to actor c. Let a_{m1}, b_{m2} and c_{m4} denote the reception events of these messages. We say that $a_{m1}\ causes\ b_{m2}$ and $b_{m2}\ causes\ c_{m3}$. Then a typical interaction constraint would be $a_{m1} \Rightarrow c_{m3} \prec 10$. Consequently, there is an implicit constraint on event b_{m2} which is to happen (well) before c_{m3}. Thus compiler/runtime system should be able to perform constraint propagation along the causality path.

Synchronizer distribution: If the synchronizer entities are maintained as runtime objects, how should their state be distributed? Here there is a classic compromise between a centralized solution where consistent updates are easy versus a distributed solution that potentially reduces bottlenecks and potentially increases fault tolerance, but at the increased cost of maintaining consistency.

Our implementation strategy, *constraint directed scheduling*, seems practical for soft real-time systems only: We do not have a procedure, neither automatic nor manual, for establishing the guaranteed satisfaction of time constraints required by hard real-time systems, and for the unrestricted type of real-time and synchronization constraints that we permit in our language. Additionally, a full *verification* of the implemented system is rarely practical. To make schedulability analysis practical one often restricts the types of constraints to *periodic* constraints. Similar restrictions can be made to RT-Synchronizers$^-$. With simple dependencies between periodic tasks generalized rate-monotonic analysis can be utilized [18]. The Real-time Object-Oriented Modeling method (ROOM) [17], which has many notions in common with the actor-model, has recently been extended with notions for specifying real-time properties [14]: Message sequence charts with annotated timing information can be used to express activation periods of methods or end-to-end deadlines on sequences of message invocations. With these two kinds of constraints and a few design guidelines the authors show how scheduling theory can be applied to room-models.

Constraint directed scheduling is an implementation technique that dynamically uses the information of the fired constraints in the synchronizers to assign deadline and release times to messages, see Figure 11. Synchronizer objects are thus maintained at run time as data-objects, whose state can be inspected by the scheduler.

Time based scheduling such as Earliest-deadline-first (EDF) can then be used to dispatch messages based on their deadlines. We propose to use EDF-scheduling because it is dynamic and optimal: if a feasible schedule exists EDF will produce one. Obviously, EDF does not in itself guarantee that a feasible schedule exists and constraint violations may therefore occur. An advantage of our strategy is that it does more than simply monitor the time constraints; it constructively applies information from the synchronizers to its scheduling decisions.

We propose to let the compiler compute a conservative version of the causality graph annotated with worst case ex-

ecution time and message propagation delays, and include a copy of it at runtime [12]. The runtime system then has the information necessary to propagate constraints automatically. The cost of this scheme is the space needed to store the causality graph and the time required to do the propagation.

An alternative to the causality graph would be to require the programmer to resolve all intermediate deadlines. In essence this requires the programmer to manually perform (deadline) constraint propagation and solving as part of the programming task. This fits with the philosophy that turning a specification into a program is a matter of refining specifications by gradually introducing more and more detail.

A recent result is [10] where certain aspects of RT-Synchronizers$^-$ are implemented in their DART framework where constraints are used to dynamically instruct the scheduler about delays and deadlines of messages. However the paper gives no systematic (automatic) translation of of constraints to scheduling information. We expect that our semantics can help in filling out this gap: Since it is operational it gives a direct algorithm for assigning deadlines to messages. That is, the operational semantics is an abstract algorithm for the behavior of the synchronizer objects.

5 Summary

We have presented a specification and programming language, RT-Synchronizers$^-$, that facilitate separate and modular specification of real-time systems: computing objects are glued together by synchronizer entities that expresses real-time and synchronization constrains. This permits a component based approach to the construction of real-time systems. Our language is explained both conceptually and formally. Our operational semantics give an abstract interpretation of the interaction constrains—this interpretation is used directly in our implementation proposal. Future work includes implementation work, both on the compiler and runtime system side.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Los Alamitos, California, 1986. ISBN 0-262-01092-5.

[2] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] G. Agha. Modeling Concurrent Systems: Actors, Nets, and the Problem of Abstraction and Composition. In *17th International Conference on Application and Theory of Petri Nets*, Osaka, Japan, June 1996.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, page 68pp, To be published.

[5] M. Akşit, J. Bosch, W. van der Sterren, and L. Bergmans. Real-Time Specification Inheritance Anomalies and Real-Time Filters. In *Proceedings ECOOP*, pages 386–407, 1994.

[6] R. Alur, C. Courcoubetis, and D. Dill. Model–checking for real–time systems. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990.

[7] S. Frølund. *Constraint-Based Synchronization of Distributed Activities*. PhD thesis, Department of Computer Science, University of Illinois at Urbana Champaign, September 1994.

[8] R. Gerber and I. Lee. Communicating Shared Resources: A Model for Distributed Real-Time Systems. In *Proc. Real-Time Systems Symposium*, pages 68–78, Santa Monica, CA, USA, 1989. IEEE.

[9] R. Gerber and I. Lee. A Layered Approach to Automating the Verification of Real-Time Systems. *IEEE Transactions on Software Engineering*, 18(9):768–784, September 1992.

[10] B. Kirk, L. Nigro, and F. Pupo. Using Real Time Constraints for Modularisation. In *Joint Modular Language Conference*, March 1997. Linz.

[11] X. Nicollin, J. Sifakis, and S. Yovine. Compiling Real-Time Specifications into Extended Automata. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.

[12] S. Ren. *An Actor-Based Framework for Real-Time Coordination*. PhD thesis, Department Computer Science. University of Illinois at Urbana-Champaign, 1997. PhD. Thesis.

[13] S. Ren and G. Agha. RT-Synchronizer: Language Support for Real-Time Specifications in Distributed Systems. *ACM Sigplan Notices*, 30(11), November 1995. Proceedings of the ACM Sigplan 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems.

[14] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Control Software. In *18th IEEE Real-Time Systems Symposium*, pages 240–251. IEEE, December 1997.

[15] I. Satoh and M. Tokoro. Semantics for a Real-Time Object-Oriented Programming Language. In *Int. Conf. on Computer Languages*, pages 159–170, Toulouse, France, 1994. IEEE.

[16] S. Schneider. An Operational Semantics for Timed CSP. *Information and Computation*, 116:193–213, 1995.

[17] B. Selic, G. Gullekson, and P. T. Ward. *Real-time Object-oriented Modeling*. Wiley Professional Computing. John Wiley & Sons, Inc., New York, 1994. ISBN 0-471-59917-4.

[18] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.

[19] P. Zhou and J. Hooman. A Proof Theory for Asynchronously Communicating Real-Time Systems. In *Proc. Real-Time Systems Symposium*, pages 177–186, Phoenix, AZ, USA, 1992. IEEE.