Towards reusable real-time objects

Brian Nielsen [a] and Gul Agha [b]

[a] *Department of Computer Science, Aalborg University, Fredrik Bajersvej 7E, DK-9220 Aalborg, Denmark*
E-mail: bnielsen@cs.auc.dk

[b] *Open Systems Laboratory, University of Illinois at Urbana-Champaign, Department of Computer Science, 1304 W. Springfield Avenue, Urbana, IL 61801, USA*
E-mail: agha@cs.uiuc.edu

Large and complex real-time systems can benefit significantly from a component-based development approach where new systems are constructed by composing reusable, documented and previously tested concurrent objects. However, reusing objects which execute under real-time constraints is problematic because application specific time and synchronization constraints are often embedded in the internals of these objects. The tight coupling of functionality and real-time constraints makes objects interdependent, and as a result difficult to reuse in another system. We propose a model which facilitates separate and modular specification of real-time constraints, and show how separation of real-time constraints and functional behavior is possible. We present our ideas using the *Actor model* to represent untimed objects, and the *Real-time Synchronizers* language to express real-time and synchronization constraints. We discuss specific mechanisms by which *Real-time Synchronizers* can govern the interaction and execution of untimed objects. We treat our model formally, and succinctly define what effect real-time constraints have on a set of concurrent objects. We briefly discuss how a middleware scheduling and event-dispatching service can use the synchronizers to execute the system.

1. Motivation

Real-time systems remain among the most challenging systems to build, and often projects are late and products faulty. Developers are faced with ever more stringent requirements for building larger, more complex systems at a faster pace and without proportional resources. However, because current tools and techniques to deal with complexity do not scale linearly with the size of programs, development problems worsen. We believe that real-time systems can benefit significantly from a development approach based on components where new systems are constructed by composing reusable, documented and previously tested components. Unfortunately, current software development methods and tools do not properly support such construction.

Because real-time systems are safety critical and often unattended, they must operate under strict end-to-end time constraints and be dependable. Dependability requirements entail both correctness and tolerance to faults. Real-time systems can

be loosely defined as systems where timely response is equally important as correct response. Real-time systems typically monitor and regulate physical equipment. Some well-known examples include: manufacturing plant automatization, where the production steps must be supervised and coordinated; chemical processes which are automatically monitored and regulated through sensors and actuators; safety systems aboard trains and cars; financial applications where stock rates must be guaranteed up-to-date and where transactions must be completed within specific time bounds.

Historically, real-time systems were built using low level programming languages and executed on dedicated hardware and specialized operating systems: efficiency, high resource utilization, and integration with hardware were the primary concerns, software modularity and reuse were only secondary. In the light of more stringent development requirements, we believe the emphasis should now be on building modular and reusable components, which can be used in many applications. Middleware services (i.e., general purpose services located between platforms and applications [Bernstein 1996]) can then be used to help integrate the components.

However, reusing real-time components is often problematic because application specific time and synchronization constraints are embedded in the internals of these components. This kind of tight coupling makes components interdependent, and consequently unlikely to be reusable in other systems. Properly supported component-based software development will allow components to be developed individually and later be composed with other individually developed or existing components, making it possible to reuse components in different applications. Thus component-based software has emerged as an active area of research. Our work makes a contribution to this area.

2. Separation and reuse

In what follows we use collections of concurrent objects to represent components in a distributed real-time system. Typically these objects model real-world entities or act as proxies for them. The objects execute concurrently and communicate by exchanging messages containing computation results or information about their local states. Objects may be larger entities than data structures such as lists or trees, they need not be heavyweight processes.

Designing reusable objects is difficult and requires skilled engineers. Building reusable concurrent real-time objects is even more difficult, and necessitates particular restrictions:

1. Objects should not schedule themselves by setting their priorities or by specifying deadlines and delays on method invocations, e.g., use expressions such as `object.method(args) deadline 10`, or contain any other type of scheduling related information.

2. Objects should not manipulate timers for programming delays or timeouts. Timer manipulation includes requesting, cancelling, and handling timer signals.

3. Objects should not have hardwired synchronization constraints. In a concurrent system, certain restrictions on order of events must be enfored in order to ensure safety and liveness. This concerns both the order of invocations of a single object, and the interaction between invocations on a set of objects.

Priorities, real-time constraints, timer values, and synchronization constraints are all properties that are likely to differ between applications, and therefore objects that embed such behavior cannot be readily reused. In addition, most of these properties are *global* properties, not properties belonging internally to a single object. For example, a priority level only makes sense when compared to the priorities of other objects. Similarly, an object is usually part of a sequence of objects chained by method calls which together must obey an end-to-end deadline. A deadline on a method invocation only represents a single object's time budget along the call chain. New applications using the object will usually have different end-to-end deadlines and different call sequences. Therefore the objects would have to be modified, and consequently re-tested, to accommodate a new time budget.

Parameterizing objects with timing and scheduling information would solve these problems only to a very limited extent. This is partly because it is difficult to know which attributes should be parameterized, and partly because concurrency constraints among objects are difficult to capture through parameters. We argue that it is better to handle the composition by a *composition software agent*, and use design methods and programming languages/environments that explicitly provide notations and abstractions for this decoupling.

Another source of reuse is the constraints themselves. We expect that many instantiations of the same constraints will recur in different applications. It would therefore be advantageous to reuse them. However, a more important reason for reuse is that real-time and synchronization constraints can be extremely tricky to specify correctly. Constraints that work as desired should be reused rather than be replaced by new similar ones. An effective and modular language should enable the programmer to factor out common constraint instances as *constraint patterns* and support their composition.

We propose a model in which both real-time and synchronization constraints can be specified in an integrated manner, enabling a fairly general set of constraints to be specified. For example, a time constraint could specify that a controller object must receive sensor data from a sensor object every 20 milliseconds. A synchronization constraint temporarily disables some actions until others have taken place, for example, to prevent a producer from inserting in a full buffer. We refer to combined real-time and synchronization constraints as interaction constraints. Both types constrain dynamic interactions *among* objects.

Our interaction constraints are conceptually installed "above" ordinary objects, and they actively enforce the developers' constraints, see figure 1. The enforcing agent is the scheduler (or schedulers) which bases its decisions on the supplied constraints.

Interaction constraints are expressed in terms of enabling conditions on communication events occurring on the interface of objects. These events constitute the

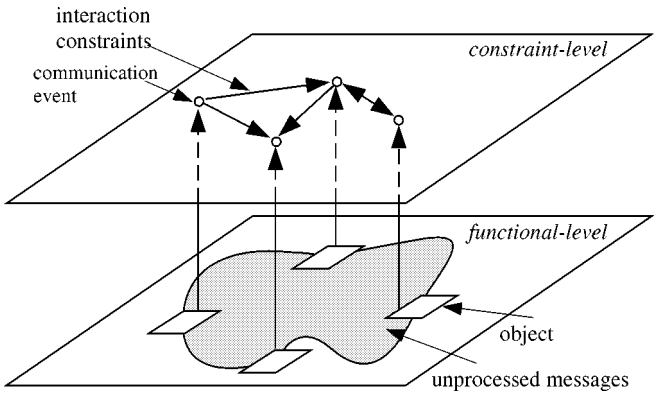

Figure 1. Separation of constraints and objects.

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. Each synchronizer implements a constraint pattern. We use the object-oriented *Actor* model to describe objects.

Section 3 introduces and exemplifies our model. Since we are interested in providing a clean and sound model, it is accompanied by a description of its semantics. Our goal is to succinctly define constraints and their effects on the objects they constrain. Section 4 provides formal definitions. Finally, in section 5 we discuss implementation.

3. Specification of interaction constraints

We use the object-oriented Actor model [Agha 1986,1990; Agha *et al.* 1997] to describe distributed computing entities (hardware or software). 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 continues its computation without waiting for, or getting a reply from, the receiver. Further, messages sent but not yet processed by the receiver are conceptually buffered in a mailbox at the receiver. The receiver receives and processes messages one at a time. In addition, actors execute *concurrently* with other actors. An actor system is illustrated in figure 2.

Each actor is identified by a unique name, called its *mail address*. A mail address can be bound to state variables of type *actor reference*. To send a message an actor executes the **send** $a.m(pv)$ primitive, where a is an actor reference variable containing the mail address of the target actor (possibly the actor itself), m is the method to be invoked, and pv is the value(s) passed. It is possible to communicate mail-addresses through messages thus allowing dynamic configuration of the communication topology.

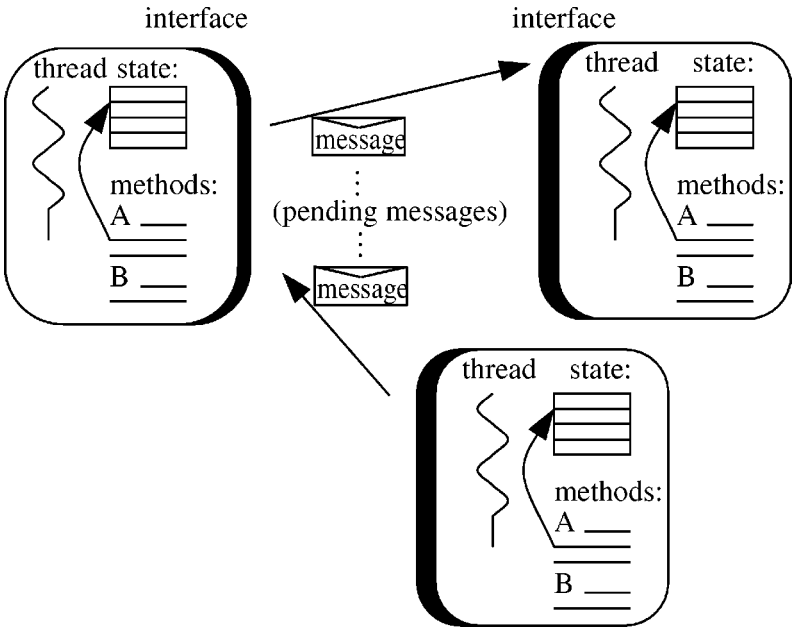

Figure 2. Illustration of an actor system.

```
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);
  }
}
```

Figure 3. Steam boiler.

The example actor program in figure 3 describes part of a simple boiler control system consisting of a pressure sensor, a controller, and a valve actuator. These entities are modelled as actors. The goal is to maintain a pre-specified pressure level in the boiler. The controller is the heart of the system. It repeatedly executes a method which sends a request to the pressure sensor for the current boiler pressure. The iteration is implemented by having the controller send itself a loop message which causes requests to be sent to the pressure sensor. The parameter of this message specifies

which actor the result must be sent to, in this case the controller itself. Upon request, the pressure sensor sends a reply containing its current pressure reading back to the initiator of the request (i.e., the controller). Based on that value, the controller computes an updated steam valve position, and sends a message to invoke the move method on the valve.

The RT-Synchronizers$^-$ language that we define in this paper to express constraints is purposely distilled: it does not include syntactic sugar for convenient description of common constraint patterns. This allows us to focus on the central ideas, and makes it easier to define a complete semantics. A *synchronizer* is an object that enforces user specified constraints on messages sent by actors. Such constraints express real-time or ordering constraints on pairs of message invocations. The messages of interest are captured by means of *patterns* that represent predicates over message contents and synchronizer state. The structure of an RT-Synchronizers$^-$ declaration is given in figure 4. It consists of 4 parts: a set of instantiation parameters, declarations of local variables, a set of constraints, and a set of triggers.

A constraint has one of the following 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 elapse. We say that event $a_1(cv_1)$ *fires* the constraint, and causes a *demand* for $a_2(cv_2)$.

$p_1 \Rightarrow p_2 \succ y$**:** 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 after $a_1(cv_1)$ fires. A pattern has the form $x_1(x_2)$ **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 boolean predicate evaluates to true. If a message satisfies a pattern, the invocation is

synchronizer $(a_1, \ldots, a_n)\{$
State Declaration

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

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

Figure 4. Structure of RT-Synchronizers$^-$. $\sim \in \{\succ, \prec\}$.

affected by a constraint which must then be satisfied before the invocation can 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, thus invalidating the constraint.

A trigger command specifies how the synchronizer's state variables change when a message invocation satisfies a given pattern. Specifically, assignment of the trigger $p : \overline{x} := \overline{exp}$ is executed when a message satisfying p is invoked. Synchronizers can thus adapt to the system's current state.

To promote modularity of interaction constraints, the constraints can be specified as a *collection* of synchronizer objects executing concurrently.

3.1. Example 1: Steam boiler constraints

The synchronizer in figure 5 describes the real-time constraints for the simple boiler control system from figure 3. The controller should read the pressure periodically (every 20 time units plus or minus some tolerance). The controller must receive sensor data from the pressure sensor within 10 time units measured from the start of the period, and it must update the steam valve position no later than 5 time units after receiving sensor data. A message sequence chart illustrating the communication among the boiler objects and the associated timing constraints is shown in figure 6.

This example shows how real-time constraints can be expressed and imposed separately from the functionality. It also shows how periodic constraints can be expressed by combining deadlines and delays. To make the language easier to use, common constraint patterns such as those enforcing periodicity can be specified as macros.

3.2. Example 2: New boiler

In a new boiler application, the pressure sensor must be polled approximately every 100 time units for pressure readings, and the pressure valve must be moved accordingly no later than 20 time units after the appropriate reading. However, in

```
actor pressureSensor ( ) { ... };
actor steamValve ( ) { ... };
actor controller (actorRef sensor,valve) { ... };

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 5. Steam boiler constraints.

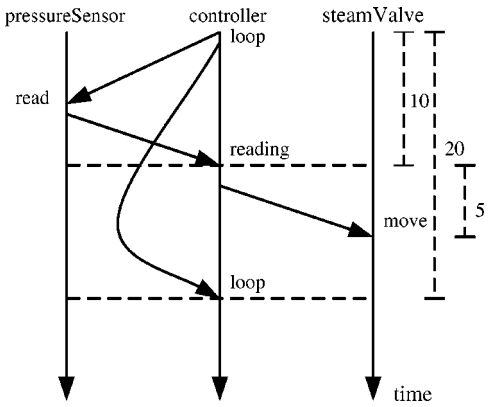

Figure 6. Message sequence chart with annotated timing constraints for the steam boiler example.

```
synchronizer newBoilerConstraints (actorRef: sensor, controller, valve){
  enum Mode {Normal,High} mode = Normal;
  //normal pressure periodic:
  sensor.read when mode==Normal ⇒ sensor.read ≺ 100+ϵ
  sensor.read when mode==Normal ⇒ sensor.read ≻ 100-ϵ
  //high pressure periodic:
  sensor.read when mode==High ⇒ sensor.read ≺ 50+ϵ
  sensor.read when mode==High ⇒ sensor.read ≻ 50-ϵ
  //deadline on move:
  sensor.read ⇒ valve.move ≺ 20
  //Trigger mode change
  controller.reading(pressure) when pressure ⩾ NormToHighThr: mode=High;
  controller.reading(pressure) when pressure ⩽ HighToNormThr: mode=Normal;
}
```

Figure 7. New steam boiler.

situations where the pressure in the boiler is high, the system must operate with a higher frequency. The pressure sensor must then be polled every 50 time units. Two threshold values, `NormToHighThr` and `HighToNormThr`, define which pressure values cause mode change.

The functional part is *reused* from example 1, i.e., the actors and their respective behaviors are unmodified, but they are now composed by the "newBoilerConstraints" synchronizer given in figure 7. The synchronizer maintains a mode state variable which tracks whether the system operates in high or normal pressure mode. The example also illustrates RT-Synchronizers$^-$'s ability to capture the dynamic changes that are common to many real-time systems through the use of a state variable, and to change time constraints accordingly.

3.3. Example 3: Time bounded buffer

This and the next example show two common real-time constraint patterns: a real-time producer-consumer relation, and rate control. These examples also

```
actor q {
  method put(Item item) { //store item };
  method get(actorRef customer) { send customer.processItem(item);}
}
actor consumer( ) {                          actor producer( ) {
  method consume( ) {                          method produce ( ) {
    send q.get(self);                            send q.put(item);
    send self.consume( );                        send self.produce( );
  }                                            }
  method processItem(Item item) { ... }      }
}
synchronizer bbConstraints (actorRef: producer, consumer, q) {
  int n=0; // no of elements in queue
  q.put ⇒ q.get ≺ 20; // time bound on get
  disable consumer.consume when n ⩽ 0; //buf empty?
  disable producer.produce when n ⩾ maxBufSz; //buf full?
  producer.produce:  n++;
  consumer.consume:  n−−;
}
```

Figure 8. Bounded buffer with time constraints.

show how RT-Synchronizers$^-$ can express synchronization (event ordering) constraints.

Figure 8 shows a time bounded buffer 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, **disable** p, to temporarily prevent messages matching the pattern p from being invoked. **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. This synchronizer could be used, for example, in a multimedia system where the queue is an actor capable of decompressing a compressed video stream: the actor has a fixed storage capacity for (uncompressed) frames and until a compressed frame is decompressed, it occupies a buffer slot. The actor accepts messages containing a compressed frame and messages removing an uncompressed frame. The frames may only stay in the actor for a bounded amount of time. The buffer space must then be freed up for processing of new, fresh frames.

3.4. Example 4: Rate control

The example shown in figure 9 illustrates how rate control can be described. At most 20 move operations can safely be performed on an actuator in any time window of 30 time units.

We use an *event generator actor* to produce message invocations so that the synchronizer changes 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 obviates the need for a 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 9. Rate control.

$$\frac{\text{Actors: } \longrightarrow_{\kappa} \qquad \text{Single constraints: } \longrightarrow_{\gamma}}{\text{Constrained System: } \longrightarrow_{\kappa\sigma}} \qquad \frac{\text{Synchronizers: } \longrightarrow_{\sigma}}{\text{Constrained System: } \longrightarrow_{\kappa\sigma}}$$

Figure 10. Dependencies of the transition systems to be defined.

4. Formal definition

In this section we provide a formal definition of our model. The formal model defines the permissible behavior of a constrained actor program, which is crucial for 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 is then given by putting the two transition systems in "parallel." Figure 10 gives an overview of the the transition systems to be defined. A numerator denominator-pair should be read as $\frac{\text{Premise}}{\text{Conclusion}}$, where the premise is the condition that must hold in order for the conclusion to hold. The κ transitions define semantics for Actors, the γ transitions for individual constraints, and the σ transitions for synchronizer objects. Finally, $\kappa\sigma$ transitions define the behavior of a constrained actor-system.

4.1. Semantics of actors

We first define a transition system κ for an actor language. This defines how the state of an 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 [Agha *et al.* 1997], where a well-developed theory of actors can be found. However, note that we present actor semantics in imperative style rather than the applicative style used in previous work. Our semantic model abstracts away the notion of methods.

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

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

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

Figure 11. Configuration transitions $\longrightarrow_\kappa$.

Instead, each actor has a single behavior – a sequence of statements – which it applies to every incoming message.

When an actor has completed processing a message it executes the **ready** command to indicate its readiness to accept a new message. As an aside, readers familiar with the classic Actor literature will note that the original **become** primitive has been replaced with **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 [Agha 1996], 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.

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 $\langle \alpha \mid \mu \rangle$, 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 the commands that remain to be executed. An actor state is written as $[E \vdash b]_a$, where 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 **ready**(x) statement. This juncture 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. In general cv encodes information about which method to invoke along with the values of the method's parameters.

The semantics of actors is given in figure 11. The **fun** transition defines the effect on system state when an actor performs an internal computation step, i.e., 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 **ready**(x) command, it becomes ready to accept a new message in an environment with the updated state variables left by the previous processing. Also, the actual value carried by the message is bound to the formal argument x. Finally, the message is removed from μ. It is exactly these receive transitions that will be constrained by RT-Synchronizers$^-$. Other transitions are only affected indirectly.

From this semantics one can make no assertions about the execution time of an actor program; how, then, can we meet real-time requirements? To make this point clear, we temporarily introduce time into the Actor semantics.

Time can be added to transition systems by introducing a special set of *delay actions* written as $\varepsilon(d)$, where d is a finite positive real-valued number representing the passage of d time units. The idea is that system execution can be observed by alternately observing a set of instantaneous transitions and observing a delay. In [Nicollin *et al.* 1992] this idea was termed the *two-phase functioning principle*: system state evolves alternately by performing a sequence of instantaneous actions and by letting time pass.

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

4.2. 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); the latter is essentially 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 thereby creating 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 con-*

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

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

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

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

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

Figure 12. Semantics for single constraints $\longrightarrow_\gamma$ where $\sim \in \{\succ, \prec\}$.

figuration $\langle\!|\xi_\sim \mid \chi|\!\rangle$, 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 12.

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-deterministically, giving the implementation maximal freedom to choose the demand it finds the most appropriate, e.g., the one with the tightest deadline.

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 as $\chi \ominus e$. Thus for $p \succ d$, d is the amount of time that *must* pass before p is enabled. In particular, 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 is less than 0. However the $\langle$**Delay**$_{\prec}$: $e\rangle$ rule 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: something that must be enforced. Without this strict definition, our constraints would degenerate to mere assertions and not convey their intended meaning. Note that an actual language implementation may not always be able to give this guarantee – either statically or dynamically – for two reasons. First, because physical resources may not exist to realize them, and second, because finding feasible schedules for general constraints is computationally very complex.

Conflicting constraints that have no solutions should be detected as part of the compiler's static program check. Ren has shown how RT-Synchronizers$^-$ constraints can be mapped to linear inequality systems for which polynomial time algorithms exist for detecting solvability [Ren and Agha 1998; Ren 1997].

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

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

Given that the behavior of each individual constraint is well defined, it is easy to define the behavior of a collection of constraints as found within a synchronizer. Essentially the individual constraints are conjoined, i.e., we require 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* **[**$\bar{\gamma}|V$**]**, where $\bar{\gamma}$ is a *set* of constraint configurations (ranged over by γ). As previously stated, V represents the state variables of a synchronizer and is a mapping from identifiers to their values. The necessary definition is shown in figure 13. A synchronizer can engage in message reception $a(cv)$ or delay $\varepsilon(e)$ only when it is permitted by every constraint.

We have omitted the rather simple definition of the effect of triggers: V' is V simultaneously updated with the specified assignments in the matched triggers.

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

Preceding sections defined 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 (denoted by $\|$) that preserves the meaning of

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

Figure 13. Semantics for a synchronizer $\longrightarrow_{\sigma}$.

Unaffected Actions

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

Receive

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

Delay

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

Figure 14. Combined behavior $\longrightarrow_{\kappa\sigma}$.

constraints. We call a collection of synchronizers an *interaction constraint system configuration* which is written as $(\sigma_1, \ldots, \sigma_n)$ where σ ranges over synchronizer configurations. The composition $\parallel$ of an actor configuration and an interaction constraint system configuration is defined in figure 14.

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 idea 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,

a program consisting of actors and RT-Synchronizers$^-$ can be viewed as a specification for the set of possible systems.

Observe that not all transition sequences defined by $\longrightarrow^*_{\kappa\sigma}$ are realizable on a physical machine. The problem is related to the progress of time and our intuition about causal ordering. Suppose event e_1 is a method invocation resulting in the sending of a message which eventually causes a method invocation, event e_2, then we surely would expect that time has progressed between these events. That is, in terms of a fictitious global clock C, it should hold that $C(e_1) < C(e_2)$. However, in our semantics, time is not *required* to pass between causally related events, but only permitted to.

There are two related problems, time locks and cluster points. A time lock occurs when no time progress is possible, i.e., the delay transition is eternally disabled. In our model this occurs as a consequence of an unsatisfiable deadline constraint. A cluster point is a bounded interval of time in which an infinite number of events occur. It is possible to write such a specification in RT-Synchronizers$^-$. However, it will not be implementable on a (finitely fast) computer! Since our goal is to define the permissible implementations, and since time locks and cluster points are only required when explicitly specified, we have taken no measure to prohibit such behavior. A compiler should, however, warn developers about such unsatisfiable constraints.

5. Middleware scheduling of RT-Synchronizers$^-$

The examples in figures 5–9 illustrated how our language can be used as a specification or modeling language that defines the structure and permissible behavior of a computer system consisting of hardware and system software executing an application.

An attractive approach to implementing a language that supports separation of objects and time constraints is to use a *middleware scheduling/event dispatching* service. Such a service is depicted in figure 15. An application consists of two parts, objects and time constraints. A set of potentially reusable objects are composed by middleware services for communication and scheduling. Communication typically includes request-reply communication, point-to-point real-time communication, and group communication. The scheduler(s) are responsible for event dispatching and resource (typically processor) allocation, based on information that is specified by the application separately from the objects. Thus, objects are being controlled by the middleware, rather than controlling themselves or each other.

Specifically, given a set of synchronizers as input, this service should, preferably without further programmer involvement, schedule message invocations in accordance with the specified real-time and synchronization constraints. The remainder of this section is devoted to uncovering what such a service must do to execute the specification directly.

Implementing our full model is not an easy task, but the difficulty is mostly related to the generality of the constraints that can be expressed, rather than to the separation of functionality and time constraints. We have identified three main tasks a compiler and scheduling service should address:

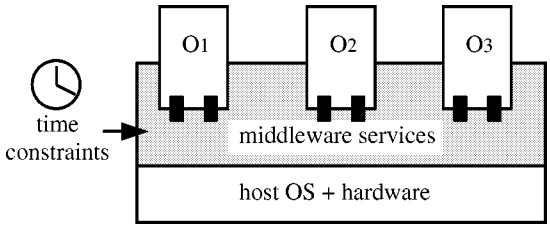

Figure 15. Middleware integrates pre-built objects.

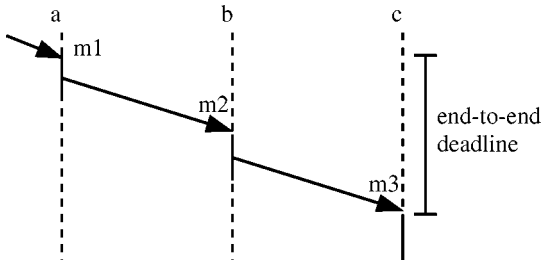

Figure 16. End-to-end deadlines require computation intermediate deadlines along the call chain.

Scheduling: One challenge is to find a scheduling strategy that satisfies the deadline constraints 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 intermediate constraints on all events along the call chain. 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_{m3} denote the reception events of these messages. Then a typical interaction constraint would be $a_{m1} \Rightarrow c_{m3} \prec 10$. This scenario is depicted in figure 16. Consequently, there is an implicit constraint on event b_{m2} which is to happen (well) before c_{m3}. Ideally, the compiler/runtime system should be able to perform constraint propagation along the call chain, and derive the intermediate deadlines.

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 increases fault tolerance, but by increasing the cost of maintaining consistency.

Our implementation idea seems practical for soft real-time systems only: we provide no procedure, whether automatic or manual, for establishing the guarantees of satisfaction of time constraints as required by hard real-time systems, and for the unrestricted type of real-time and synchronization constraints that we permit in our

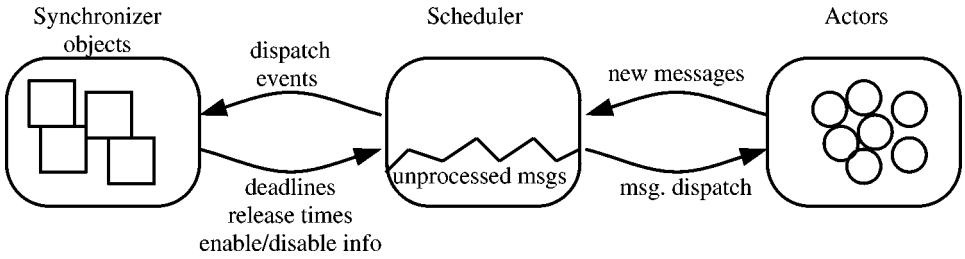

Figure 17. Implementation architecture with constraint directed scheduling.

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 [Sha *et al.* 1994].

Constraint directed scheduling is an implementation technique that dynamically uses the information of the fired constraints in the synchronizers to assign deadlines and release times to messages (see figure 17). 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 call graph annotated with worst case execution time and message propagation delays, and include a copy of it at runtime [Ren 1997]. The runtime system then has the information necessary to propagate constraints automatically when this cannot be done statically by the compiler. Moreover, we expect that in many cases the compiler would be able to compile away synchronizers entirely. It can generate code (similar to remote-procedure-call stubs) which can be linked with the objects. This code implements the time constraints by manipulating timers, setting priorities and/or instructing the scheduler about method call deadlines, etc.

It is interesting to note that the operational semantics can assist in the implementation of a constraint directed scheduling system. An operational semantics can often be constructed such that it constitutes an abstract algorithm for the language implementation. However, because our semantics abstracts away any notion of resources and execution time, in our case, this algorithm can only be partial. In particular, it does not solve the constraint propagation problem mentioned earlier.

The following example demonstrates two potential benefits of the semantics. First, it shows how the semantics manipulates the synchronizer data structure by adding

and removing constraints, and second it indicates how release times and deadlines for messages can be deduced. Recall the boiler example in section 3.1. We show how the runtime system may execute that specification. We maintain two important data structures, the set of fired demands, and the pool of unprocessed messages. We reuse the notation for demands from the semantics: $\langle\!|\xi_\sim \mid \chi|\!\rangle$, where $\xi_\sim$ stands for the static description of a constraint and χ is the multi-set of instantiated demands. A message is written as $o.m[R,D]$, where o is the target object, m the method to be invoked, and R and D respectively the release time and deadline of the message. In the following, we measure time relative to a global clock **t**, and not using individual timers as was convenient in the semantics. Each row in figure 18 shows the global time at which a given event (i.e., message invocation) occurs, the resulting synchronizer state, and the set of unprocessed messages (including those produced by the event).

At time 0, the system is shown in the initial state in which the message pool contains an initialization message (*controller.loop*) and in which no synchronizer demands have been fired. Suppose the scheduler invokes the *controller.loop* message at time 1. This invocation matches three constraints and consequently causes the synchronizer to issue three new demands. The first two constitute the periodic constraint on a future loop message and the last one determines the deadline on the sensor reading. During processing of the loop message the controller sends out two new messages, the loop message to itself, and a read request to the pressure sensor.

The new loop message matches two demands, and according to the semantics these are applied conjunctively. The runtime system can therefore deduce the release time and the deadline (an ϵ interval around time 21) for the loop message from the demands. Deducing a deadline for *sensor.read* constitutes a more difficult case (labeled with a ‡ symbol in figure 18). There is no immediate matching demand on which to base the deadline. But it can be noted that there is a demand for which no matching message exists in the message pool. It is therefore likely that invocation of the unmatched *sensor.read* message will cause sending of the demanded message (as it indeed turns out to be the case in this example). Therefore the *sensor.read* message should be assigned a deadline before the demanded deadline (at time 11). The specific choice of deadline is in general a heuristic function of slack time and method computation time. Here time 6 is chosen.

The approach of assigning where unmatched messages are assigned deadlines based on the most urgent unmatched demand will generally constrain the system unnecessarily, but selecting precisely the right message to constrain is generally impossible without extra information about potential causal relations between messages. This information is exactly what needs to be generated by the compiler. Less ideally, the missing constraints could be resolved explicitly by the programmer by providing additional synchronizers. In less expressive real-time programming languages where end-to-end constraints cannot be expressed, the programmer would always be forced to do this.

Resuming the example at time 4 where *sensor.read* is invoked, the sensor responds with a *controller.reading*. Since this message matches a demand, it inherits the deadline

t	Event	Synchronizer State	Message Pool
0	(initial)	$\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\prec 20+\epsilon$ | $\emptyset$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\succ 20-\epsilon$ | $\emptyset$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.reading $\prec 10$ | $\emptyset$ $\rangle\!\rangle$ $\langle\!\langle$ c.reading $\Rightarrow$ v.move $\prec 5$ | $\emptyset$ $\rangle\!\rangle$	c.loop$[0,\infty]$
1	c.loop	$\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\prec 20+\epsilon$ | c.loop $\prec 1+20+\epsilon$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\succ 20-\epsilon$ | c.loop $\succ 1+20-\epsilon$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.reading $\prec 10$ | c.reading $\prec 1+10$ $\rangle\!\rangle$ $\langle\!\langle$ c.reading $\Rightarrow$ v.move $\prec 5$ | $\emptyset$ $\rangle\!\rangle$	c.loop$[21-\epsilon, 21+\epsilon]$ s.read$[0,6]^{\ddagger}$
4	s.read	$\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\prec 20+\epsilon$ | c.loop $\prec 1+20+\epsilon$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\succ 20-\epsilon$ | c.loop $\succ 1+20-\epsilon$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.reading $\prec 10$ | c.reading $\prec 1+10$ $\rangle\!\rangle$ $\langle\!\langle$ c.reading $\Rightarrow$ v.move $\prec 5$ | $\emptyset$ $\rangle\!\rangle$	c.loop$[21-\epsilon, 21+\epsilon]$ c.reading$[0,11]$
9	c.reading	$\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\prec 20+\epsilon$ | c.loop $\prec 1+20+\epsilon$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\succ 20-\epsilon$ | c.loop $\succ 1+20-\epsilon$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.reading $\prec 10$ | $\emptyset$ $\rangle\!\rangle$ $\langle\!\langle$ c.reading $\Rightarrow$ v.move $\prec 5$ | v.move $\prec 9+5$ $\rangle\!\rangle$	c.loop$[21-\epsilon, 21+\epsilon]$ c.move$[0,14]$
13	v.move	$\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\prec 20+\epsilon$ | c.loop $\prec 1+20+\epsilon$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\succ 20-\epsilon$ | c.loop $\succ 1+20-\epsilon$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.reading $\prec 10$ | $\emptyset$ $\rangle\!\rangle$ $\langle\!\langle$ c.reading $\Rightarrow$ v.move $\prec 5$ | $\emptyset$ $\rangle\!\rangle$	c.loop$[21-\epsilon, 21+\epsilon]$
21	c.loop	$\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\prec 20+\epsilon$ | c.loop $\prec 21+20+\epsilon$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.loop $\succ 20-\epsilon$ | c.loop $\succ 21+20-\epsilon$ $\rangle\!\rangle$ $\langle\!\langle$ c.loop $\Rightarrow$ c.reading $\prec 10$ | c.reading $\prec 21+10$ $\rangle\!\rangle$ $\langle\!\langle$ c.reading $\Rightarrow$ v.move $\prec 5$ | $\emptyset$ $\rangle\!\rangle$	c.loop$[41-\epsilon, 41+\epsilon]$ s.read$[0,26]$

Figure 18. Sample execution of the boiler specification.

from that (time 11). The result of invoking the reading message (at time 9) is the firing of a new demand on the valve movement and the sending of a *valve.move* message. Again, the runtime system is able to deduce the deadline on the move message from the move demand. Finally, at time 21, the loop message is invoked. This satisfies the remaining two demands, but at the same time fires two new demands, starting the next period.

6. Related work

Real-time CORBA (Common Object Request Broker Architecture) [Group 1996] is a highly visible research effort where practitioners are shifting towards component-based real-time systems. An object request broker can be viewed as middleware facilitating transparent client-server communication in a heterogeneous distributed system. It also contains other communication services to facilitate building distributed applications. However, according to [Schmidt *et al.* 1997], current ORBs are ill-suited for real-time systems for at least four reasons. They lack interfaces for specifying quality of service, quality of service enforcement, real-time programming facilities, and performance optimizations.

Current proposals for real-time CORBA [Cooper *et al.* 1997; Feng *et al.* 1997; Kalogeraki *et al.* 1997; Schmidt *et al.* 1997] use a quality of service metaphor for specifying real-time constraints. Typically, the interface definition language is extended with QoS-datatypes. In TAO ORB [Schmidt *et al.* 1997], these parameters, which are necessary for guaranteeing schedulability according to rate monotonic scheduling, include worst case execution time, period, and importance. In the NRad/URI's proposal [Cooper *et al.* 1997] for a dynamic CORBA, time constraints are specified in a structure containing importance, deadline and period, and the constraints specify time bounds on a client's method invocations on a server. The proposed runtime system uses this information to compute dynamic scheduling and queuing priorities. The Realize proposal [Kalogeraki *et al.* 1997] associates deadline, reliability, and importance attributes to application tasks, where a task is defined as a sequence of method invocations between an external input and the generation of an external result. That is, deadlines in Realize are true end-to-end deadlines.

We see a clear trend in specifying real-time requirements through interface definitions and letting middleware enforce them. Clients and servers are largely unaware of the imposed real-time requirements. However, we think that these approaches – although an improvement – are imperfect:

- The quality of service attributes seem to be derived from what current run-time systems can manage rather than forming a coherent set. We have opted for a clean language instead of a more or less arbitrary collection of attributes.
- The types of constraints that can be specified are restrictive, e.g., only periods or deadlines between request and reply events. In addition, the constraints are static; once assigned they cannot be modified to respond to dynamic changes in

the system's state of affairs. We allow for a fairly general set of constraints to be specified.

- Synchronization constraints are not considered. In our proposal, synchronization constraints are specified using the same mechanism as time constraints.

The concept of separating functional behavior and interaction policies for Actors was first proposed in [Frølund and Agha 1993] and a detailed description, operational semantics and implementation can be found in [Frølund 1996]. That work only considered constraints on the order of operations. Our work is a continuation of this line of research where we have extended it to apply to real-time systems and provided a formal treatment of the extended model. However, to what extent real-time and synchronization constraints can always be cleanly separated from functionality remains an open issue, and one which we think can be best resolved through larger case studies.

Another approach which permits separate specification of real-time and synchronization constraints for an object oriented language is the composition filter model [Akşit *et al.* 1994; Bergmans and Akşit 1995]. Real-time input and output filters declared in an extended interface enable the specification of time bounds on method executions. Among the differences between composition filters and RT-Synchronizers$^-$ is that RT-Synchronizers$^-$ takes a global view of a collection of objects whereas the composition filter model takes a single object view. No formal treatment of composition filters appears to be available in the literature.

The Real-time Object-Oriented Modeling method (ROOM) [Selic *et al.* 1994], which has many notions in common with the Actor model, has recently been extended with notions for specifying real-time properties [Saksena *et al.* 1997]: message sequence charts with annotated timing information can now 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.

Our approach to defining the semantics is inspired by recent research in formal specification languages for real-time systems, and the use of timed transition systems is borrowed from these languages. Languages often take the form of extended automata (Timed Automata [Alur *et al.* 1990], Timed Graphs [Alur *et al.* 1990; Nicollin *et al.* 1992]), or process algebras such as Timed CSP [Schneider 1995]. A different approach is to include a model of the underlying execution resources. This approach is taken in [Satoh and Tokoro 1994] and [Zhou and Hooman 1992]. 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 [Gerber and Lee 1989,1992]. 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 relatively concrete systems, rather than being specifications for a set of possible systems, as was our goal.

A recent implementation result is [Kirk *et al.* 1997], 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 constraints to scheduling information. We expect that our semantics can help in filling up this gap.

7. Discussion

Developers of modern real-time systems are required to construct increasingly large and complex systems, preferably at no extra cost. To satisfy this requirement, it is essential that developers can build real-time systems from existing components, and that newly developed components can be reused in several applications. We argued that in order to facilitate reuse of real-time objects, the real-time and synchronization constraints governing the object's interaction should be specified separately from the objects themselves. However, current development methods do not adequately support such separation.

We formulated our ideas in the context of Actors, and an associated specification language, RT-Synchronizers$^-$. Combined, they enable separate and modular specification of real-time systems: computing objects are glued together by synchronizer entities that express real-time and synchronization constraints. However, we believe that these ideas are applicable beyond these specific languages.

Our model is explained both conceptually and formally. Through a series of examples we indicated how separate specification is possible. Our operational semantics defines exactly what constraints are and what their effect on a given set of objects should be.

Our work on semantic modeling has clarified our understanding of the behavior of our model, and provides a succinct and detailed definition of synchronizers and constrained actor programs. In particular, we have gained new insight into three areas, which made the effort worthwhile:

- We defined the semantics in a modular fashion by composing a transition system for the untimed object-model with a transition system which interprets the time constraints. This composition explicitly points out which object transitions are affected and how: reception of messages and time-progress may only occur when permitted by the constraints. Other object transitions are only indirectly affected.
  The modularity opens the possibility of plugging in a different constraint specification language, i.e., the $\longrightarrow_\sigma$ transition could be replaced with the semantics for the new language. The composition will work when affected transitions remain as above, and when the semantics of the new language can be given as a timed transition system. Thus, our *constraining* concept is captured by the composition.
- Our semantics helped uncover some of the semantic subtleties of our constraint language, such as what happens when patterns and constraints overlap. For example, the same message may both fire a new demand as well as satisfy an existing

one. Moreover, we decided that overlapping constraints should be interpreted conjunctively, i.e., both must be satisfied. Finally, we decided that adding more synchronizers should further restrict the behavior of objects; i.e., synchronizers must be satisfied conjunctively.
It should also be noted that the rules defining the semantics of individual constraints appear complicated. This should give food for thought when revising the language or the semantics.

- The last major benefit is that our semantics suggests an implementation strategy suitable for soft real-time systems. The synchronizer entities can be maintained at runtime and can be used to extract information about release times and deadlines of messages. The semantics gives an abstract interpretation of the synchronizer objects and specifies how demands should be added or removed.

Building real-time components and architectures for integrating them is an area of active research. We believe that with additional research, component-based development will allow more complex real-time systems to be developed on schedule. However, additional work is needed, both on the models used for separate specification and on the middleware services necessary to implement them.

Acknowledgements

This work was made possible in part by support from the US National Science Foundation under contracts NSF CCR-9523253 and NSF CCR-9619522; by support from the US Air Force Office of Scientific Research, under contract AF DC 5-36128. The authors would like to thank other members of the Open Systems Laboratory for their comments and critical insights into the work related in this paper. In particular, we would like to thank Shangping Ren for her contribution to the definition of RTSynchronizers, and to Nadeem Jamali for his comments on a draft of this paper. A part of this research was done while the first author was a visitor to the University of Illinois Open Systems Laboratory under a fellowship from the Danish Technical Research Foundation and the Danish Research Academy.

References

Agha, G. (1986), *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Los Alamitos, CA.

Agha, G. (1990), "Concurrent Object-Oriented Programming," *Communications of the ACM 33*, 9, 125–141.

Agha, G. (1996), "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.

Agha, G., I.A. Mason, S.F. Smith, and C.L. Talcott (1997), "A Foundation for Actor Computation," *Journal of Functional Programming 7*, 1–72.

Akşit, M., J. Bosch, W. van der Sterren, and L. Bergmans (1994), "Real-Time Specification Inheritance Anomalies and Real-Time Filters," In *Proceedings ECOOP*, pp. 386–407.

Alur, R., C. Courcoubetis, and D. Dill (1990), "Model-Checking for Real-Time Systems," In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pp. 414–425.

Bergmans, L. and M. Akşit (1995) "Composing Synchronization and Real-Time Constraints," In *Proceedings of the Object Oriented Real-Time Systems (OORTS) Workshop*, San Antonio, TX. In conjunction with 7th IEEE Symposium on Parallel and Distributed Computing Systems.

Bernstein, P.A. (1996), "Middleware – A Model for Distributed System Services," *Communications of the ACM 39*, 2, 86–98.

Cooper, G., L.C. DiPippo, L. Esibov, R. Ginis, R. Johnston, P. Kortman, P. Krupp, J. Mauer, M. Squadrito, B. Thuraisingham, S. Wohlever, and V.F. Wolfe (1997), "Real-Time CORBA Development at MITRE, NRaD, Tri-Pacific and URI," In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, IEEE, San Francisco, CA, pp. 69–74.

Feng, W., U. Syyid, and J.W.-S. Liu (1997) "Providing for an Open, Real-Time CORBA," In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, IEEE, San Francisco, CA, pp. 75–80.

Frølund, S. (1996), *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*, MIT Press.

Frølund, S. and G. Agha (1993), "A Language Framework for Multi-Object Coordination," In *Proceedings of the European Conference on Object Oriented Programming (ECOOP) '93*, O. Nierstrasz, Ed., Lecture Notes in Computer Science, Vol. 707, Springer-Verlag, Kaiserslautern, Germany, pp. 346–360.

Gerber, R. and I. Lee (1989) "Communicating Shared Resources: A Model for Distributed Real-Time Systems," In *Proc. Real-Time Systems Symposium*, IEEE, Santa Monica, CA, pp. 68–78.

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

Group, O.M. (1996), "Realtime CORBA – A White Paper – Issue 1.0," Technical Report ORBOS/96-09-01, Object Management Group.

Kalogeraki V., P. Melliar-Smith, and L. Moser (1997), "Soft Real-Time Resource Management in CORBA Distributed Systems," In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, IEEE, San Francisco, CA, pp. 46–51.

Kiely, D. (1998), "Are Components the Future of Software," *IEEE Computer 32*, 2, 10–11.

Kirk, B., L. Nigro, and F. Pupo (1997), "Using Real Time Constraints for Modularisation," In *Joint Modular Language Conference*, Linz.

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

Ren, S. (1997), "An Actor-Based Framework for Real-Time Coordination," Ph.D. thesis, Department Computer Science, University of Illinois at Urbana-Champaign.

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

Ren, S. and G. Agha (1998), "A Modular Approach for Programming Embedded System," In *Lectures on Embedded Systems*, Lecture Notes in Computer Science, Vol. 1494, F. Vaandrager and G. Rozenberg, Eds., Springer-Verlag, pp. 170–207.

Ren, S., G. Agha, and M. Saito (1996), "A Modular Approach for Programming Distributed Real-Time Systems," *Journal of Parallel and Distributed Computing 36*, 1, 4–42.

Saksena, M., P. Freedman, and P. Rodziewicz (1997), "Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Control Software," In *18th IEEE Real-Time Systems Symposium*, IEEE, pp. 240–251.

Satoh, I. and M. Tokoro (1994), "Semantics for a Real-Time Object-Oriented Programming Language," In *Int. Conf. on Computer Languages*, IEEE, Toulouse, France, pp. 159–170.

Schmidt, D.C., R. Bector, and D.L. Levine (1997), "An ORB Endsystem Architecture for Statically Scheduled Real-Time Applications," In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, IEEE, San Francisco, CA, pp. 52–60.

Schmidt, D.C. and M.E. Fayad (1997a), "Lessons Learned Building Reusable OO Frameworks for Distributed Software" *Communications of the ACM 40*, 10, 85–87.

Schmidt, D.C. and M.E. Fayad (1997b), "Object-Oriented Application Frameworks," *Communications of the ACM 40*, 10, 32–38.

Schneider, S. "An Operational Semantics for Timed CSP," *Information and Computation 116*, 193–213.

Selic, B., G. Gullekson, and P.T. Ward (1994), *Real-Time Object-Oriented Modeling*, Wiley Professional Computing, Wiley, New York.

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

Zhou, P. and J. Hooman (1992), "A Proof Theory for Asynchronously Communicating Real-Time Systems," In *Proc. Real-Time Systems Symposium*, IEEE, Phoenix, AZ, pp. 177–186.