A Modular Approach for Programming Distributed Real-Time Systems *

Shangping Ren Gul A. Agha
Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801, USA

Email: { ren | agha }@cs.uiuc.edu

Masahiko Saito †
Hitachi Research Laboratory
1-1, Omika-cho 7-chome, Hitachi-shi
Ibaraki-ken, 319-12 Japan

Email: miyabi@hrl.hitachi.co.jp

*The research described has been made possible in part by support from the Office of Naval Research (ONR contract numbers N00014-90-J-1899 and N00014-93-1-0273), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, by Hitachi, and by the National Science Foundation (NSF CCR 93-12495).

†The authors thank Brian Nielsen, Daniel Sturman, Mark Astley, Rajendra Panwar, James Waldby and other members of the Open Systems Laboratory who have provided helpful suggestions and discussion.

A Modular Approach for Prog. Distributed RT Systems

Contact author: Shangping Ren
1304 W. Springfield Ave.
Urbana, IL 61801
Phone: (217)- 333-4764
Email: ren@cs.uiuc.edu

Abstract

Conventional real-time programs associate real-time requirements with individual commands in a program. This approach has three weaknesses. First, it intermixes two different design concerns: functional correctness and temporal correctness. Second, mixing real-time requirements with program statements makes it harder, and in some cases infeasible, to specify constraints between objects. Third, it limits the ability to independently modify either the timing constraints or the representations of objects.

We describe a new approach that separates real-time constraints from functional aspects of an application; real-time constraints are described by synchronization code between the interfaces of objects. Objects in our system are defined using a real-time variant of the Actor model. We define a high-level programming language construct called *RTsynchronizer*, which specifies a collection of temporal constraints between actors. Thus, our approach separates what an object does from when it does it. Such separation also facilitates the ability to dynamically modify real-time constraints.

We illustrate the use of RTsynchronizers by a number of examples and then describe a meta-architecture that can be used to implement RTsynchronizers.

1 Introduction

Real-time programming languages typically intermix constructs for specifying timing constraints with those for specifying a computation. A number of researchers have proposed the use of an object-oriented paradigm for specifying of real-time constraints [25, 23, 32]; they propose to encapsulate timing constraints the same way objects hide the implementation of an abstract data type. We argue that encapsulating timing constraints into application code compromises the modularity of real-time programs: changing the implementation of an object requires that the timing constraints be re-specified and vice versa. Moreover, confining timing constraints within objects requires that a programmer explicitly translate constraints between events on different objects into constraints on execution within individual objects. Such explicit translation may not always be feasible. In particular, such low level specification requires that a programmer have a detailed model of scheduling, as well as a model of resources such as communication delay and bandwidth.

Many years ago, Nicklaus Wirth advocated the use of using a programming discipline that confines specification of temporal constraints to certain isolated components in some standard pattern and reuses the application code separately from its timing constraint specifications [33]. We take Wirth's proposal to its logical limit: our view is that a real-time system can be decomposed into modules which specify the functional behavior of components and modules which enforce timing constraints. We specify timing constraints in terms of the interfaces of a group of objects, thus maintaining the encapsulation of objects.

We model components of distributed real-time systems uniformly as *actors*. For example, control processes and hardware devices are modeled as actors. A general purpose actor programming language is used to specify functional behavior; it is augmented with a new construct, called *RTsynchronizer*, which describes timing relations. An RTsynchronizer is a collection of declarative constraints that restrict the temporal behavior of events over a group of actors. Essentially, RTsynchronizers are abstractions of common temporal coordination patterns, which allow a programmer to think in natural terms about distributed timing constraints. Using RTsynchronizers also allows a programmer to abstract over collections of similar constraints that may be applied to different groups of objects that have the same respective type interfaces (Figure 1).

A group of actors may be constrained by overlapping RTsynchronizers. Moreover, RTsynchronizers are dynamic — they can be added or removed in a running system. By supporting incremental modification of temporal coordination constraints, RTsynchronizers simplify the task of responding to changing requirements. However, although RTsynchronizers provide greater modularity, they do not simplify the problem of enforcing timing constraints in a given system.

The rest of this paper is structured as follows. Section 2 discusses related work

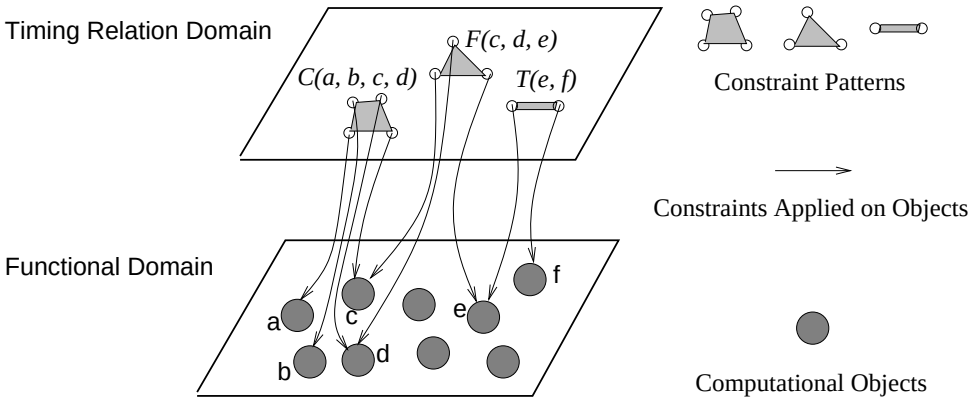

Figure 1: Decomposition of applications into timing relations and computational objects. Timing relations are specified as constraints between actors and actors implement the functional behavior

and motivates the design of our model and a new approach. Section 3 introduces the Actor model, and defines a real-time extension for it. Section 4 describes the language construct RTsynchronizers, their syntax and informal semantics. Section 5 presents examples that illustrate the use of RTsynchronizer. Section 6 discusses implementation of RTsynchronizer in detail. Section 7 restates our main results and outlines future work.

2 Related Work

Perhaps, the most commonly used language for real-time programming is Ada. Ada was designed for the U.S. Department of Defense and is intended for critical real-time software design [19]. The predefined library package `CALENDAR` in Ada provides programmers with wall-clock time. The only time related language construct in Ada is `delay`. The delayed time is *relative* to the time when the `delay` statement is executed. However, Ada semantics does not enforce non-preemptive execution of statements, and hence `delay` can only guarantee a minimal task suspension. Precise periodic task execution cannot even be specified in an Ada program [31, 2]. Later versions of Ada, such as Ada95, have enhanced the `delay` construct and provided the `delay until` statement, which allows the programmer to specify an *absolute delay* time. Again, the semantics for both constructs only guarantee the lower bound of the delayed time. In addition, the only communication mechanism in Ada is *rendezvous*; asynchronous nonblocking inter-process communication, common in distributed systems, is not supported. Besides these disadvantages, time-related statements are

mixed with ordinary statements, which makes real-time program design, modification and reuse difficult.

Some researchers have proposed integrating timing constraints with sequential objects [20, 32, 25]. For instance, the real-time programming language Flex [25] extends C++ with a *control block* language construct, which may reside in the body of methods; timing constraints are specified in the control block. For our purpose, there are two weaknesses in this work. First, timing requirements are intermixed with application code within methods. Second, concurrency is not supported.

Concurrent object-oriented programming is an active area of research interest [8, 14, 5]. A number of real-time languages are based on concurrent object-oriented languages.

Ishikawa et al. proposed a concurrent object-oriented programming language called *RTC++* [23]. In RTC++, temporally constrained objects (called *real-time objects*) are distinguished from ordinary unconstrained objects. The distinction between real-time objects and ordinary objects means that objects with the same logical behavior but different temporal constraints require separate specifications. Timing constraints in RTC++ may be specified as parts of method declarations. However, the language also allows timing constraints to be specified in commands inside methods. The failure to impose a stricter discipline may lead to an intermixing of real-time constraint code and functional code. In addition, RTC++ uses a multi-threaded model to describe real-time applications: multiple operations can be invoked concurrently on a single object and affect its state. Thus, synchronization within individual objects is needed. Finally, RTC++, does not support inter-object timing constraints.

From a linguistic point of view, the cleanest approach thus far may be that of *real-time filters*, introduced in the work of Aksit et al. [10, 9]. Real-time filters extend an object's interface to include *input filters* and *output filters*. An out-going message must pass through a series of zero or more output filters before being sent to its target. Similarly, an incoming message passes through input filters before being received by the object. Timing constraints are encoded within filters as parameters; such filters may access an object within each message which represents a temporal constraint on the local execution. Temporal constraints are specified relative to message acceptance. The filter mechanism helps make complicated programs more manageable. However, a potential difficulty with the filters approach is an inability to abstractly specify timing constraints for a group of distributed objects.

The concurrent programming language Erlang [12] is based on parallel process model and intended for soft real-time systems. Erlang programs essentially obey actor semantics: e.g., there is no shared memory and all interaction between processes is by asynchronous message passing. The timing construct provided by Erlang is *after* which implements the *timeout* of waiting for specific messages and priority messages by specifying zero as timeout limit. However, the *after* constraint is hard coded with

the basic *receive* primitive inside each module in Erlang, which makes it impossible to modify timing constraints independently with other functional code.

Our work builds on the earlier work of Frølund and Agha [17, 18] who developed *synchronizers* to provide a declarative specification of coordination constraints on groups of actors. However, the semantics of RTsynchronizers is quite different. Synchronizers can enforce *qualitative* temporal ordering and indivisible (atomic) scheduling of multiple invocations at a group of objects. On the other hand, RTsynchronizers specify *quantitative* constraints. Moreover, the two sorts of constraints require very different implementation. Synchronizers are conservative, they can merely suspend invocations until constraints are met [18]. On the other hand, real-time constraints may require progress based on predicted behavior. While a formal semantics for RTsynchronizers remains to be defined, some recent work has made progress in that direction [27].

3 Real-Time Actor Model

3.1 Actor Model

The Actor model, developed by Hewitt [22] and Agha [3], provides a uniform abstract representation of distributed systems. Actors are encapsulated concurrent objects that interact with each other by sending buffered, asynchronous messages [3, 6, 7]. Memory chips, control devices, actuators, subprograms, and entire computers may be thought of as actors. In other words, all components of distributed real-time systems can be uniformly abstracted as actors; the model is independent of the hardware platform used to implement it.

Each actor has its own thread of control and a unique mail address which may be used to send messages to it. Note that an actor's mail address may be included in messages sent to other actors, thus supporting dynamic reconfigurability. Actors may create new actors with their own unique mail addresses. Finally, actors may change their own local state. Each invocation (i.e. activation by a message) of an actor is atomic and uninterruptable. Any sequential programming language may be extended to an actor language by adding the following three operators:

- **send** specifies a destination actor and a message; in an object-oriented language, the latter may consist of a method to be invoked at the destination actor, and the values to be bound to parameters of the method. By default, communication is asynchronous; synchronous communication can be defined via abstraction [3, 4].
- **new** dynamically creates actors. The **new** operator takes an actor behavior as a parameter and returns the address of a newly created actors with the specified behavior.

- **ready** marks the end of processing the current message. After **ready** is executed, the actor may process the next message in its mail queue.

In the actor model, an *event* is defined as the invocation of a message at an actor; thus all events in the model are *arrival* events [15]. Events are instantaneous and obey: (1) *local arrival orders* and (2) *a causal order*. Each actor is modeled as a single threaded unit: it is invoked by one message at a time. Hence, the message arrival order at each individual actor is linear, or total. Because events at different actors may happen concurrently, the causal order is only partial. The transitive closure of local arrival orders and the causal order is called the *combined ordering*. Obviously, the combined order is also a partial order.

3.2 Actors in Real-Time

What is 'time'? Aristotle [1] considered philosophical aspects of time, such as: discreteness versus density, absolute (past, present, future) versus relative (before, concurrent with, after) orderings, and primitive entities of time (instants, intervals, events). These same concerns arise in computer science [11, 30] and engineering because of the need for synchronization among different computational units.

We assume that time is a dense domain; both absolute and relative orderings are defined; and instants are the primitive entities of time. Our reasons for making these choices are as follows:

- The execution of actors is asynchronous. Therefore events at different actors can happen arbitrarily close to each other in time. By defining the time domain as dense, we avoid the need to predetermine the minimal granularity of time. Discrete time steps would make modeling composition of systems problematic.

- Causal ordering in an actor system relates events at different actors and hence plays a vital role in a distributed computation. Therefore, using relative time is sometimes adequate. However, if absolute time can be provided, it simplifies description of quantitative timing relation in distributed systems. Hence, we assume that relative time relations and absolute time relations are both defined for our model.

- Events in the actor model are instantaneous; time instants correspond to the duration of events.

Intuitions about time and computation constrain what structures are possible for the combined ordering (defined earlier). Given these intuitions, global time is realizable, as shown by Clinger [15]:

(Strong) Axiom of Realizability *There exists a one-to-one mapping* g *from the events* E *into a set of real numbers without cluster points which preserves the combined ordering.*

Note that the absence of cluster points prevents an infinite number of events from happening in a finite amount of time.

We pick real time as a unique global reference time, and extend the actor model to include time dimension with the following assumptions. First, each actor a has its own local clock C_a with known bounded rate of drift $\rho \geq 0$ with respect to real-time. That is, for all a and all $t > t'$,

$$(1+\rho)^{-1} \leq \frac{C_a(t) - C_a(t')}{(t-t')} \leq (1+\rho)$$

where $C_a(t)$ is the reading of C_a at real-time t. Additionally, each local clock is *approximately synchronized* with every other, i.e., there exists an ϵ such that for all t and any actors a and b, $|C_a(t) - C_b(t)| \leq \epsilon$. Second, method execution on each actor is atomic and non-preemptive. Finally, method execution time is subsumed by communication and scheduling delay.

These assumptions are realistic and implementable (cf. [24, 13, 21]). With these assumptions, our real-time computational model is simplified in two ways. First, because local clocks are synchronized, a global clock in the system is obtained. Second, because actor invocations are scheduled atomically (non-preemptively), the message processing time is predictable as it only depends on an actor's local state and the contents of the message it is processing. Thus an upper bound on the worst-case execution time can be decided statically, assuming that each actor invocation may not result in an infinite loop. Hence, it is sufficient to use message invocation time (recall that message invocations correspond to events) as the *only* timing parameter to specify and constrain object temporal behavior.

To summarize, global clock time is one of the valid global time maps which preserves the partial combined ordering between events. A total *global precedence relation* among events may be defined as follows:

Global quantitative precedence relation ($\prec_d$) *For any two events e_1 and e_2 in a system with a global clock , if e_1 happens at time t_1, and e_2 happens at time t_2, with respect to the global clock, then $e_1 \prec_d e_2$ iff $t_2 = t_1 + d$, where t_1, $t_2 \in \Re^+$ and $d \in \Re^+ \cup \{0\}$.*

In distributed systems the global quantitative precedence relation may be approximated by using a common clock, provided the granularity of observation is sufficiently coarse.

Real-time distributed systems are systems in which the global quantitative precedence relation among events has to satisfy specified requirements.

In the next section, we provide a high-level language construct that allows programmers to specify timing constraints based on the events' global quantitative precedence relation, independently from the application's code.

4 RTsynchronizers

An RTsynchronizer is like an ordinary actor in that it has a mutable state and it may be dynamically created. However, unlike ordinary actors, an RTsynchronizer does not send or receive messages. Rather, it provides a declarative specification of quantitative temporal ordering between certain message invocations at a group of actors.

4.1 Syntax

An RTsynchronizer is created by instantiating a *template* using the mail addresses of specific actors and values for other variables. Thus:

new MyRTS (**actor**:a1, a2, a3; **int**: d1, d2)

creates a new RTsynchronizer whose behavior is given by `MyRTS` which constrains invocations of messages on the actors `a1`, `a2`, and `a3`; `d1` and `d2` are parameters for data values. The behaviors of RTsynchronizers (such as the one bound to the name `MyRTS` above) are specified by a template as follows:

```
RTsynchronizer MyRTS( actor:actor1, actor2, actor3;  int: v1, v2) {
  Declare
   \* declare and initialize
     local variables; *\
  Constrain
   constraint_1;
   ...;
   constraint_n
 }
```

Constraints may make temporal assertions about events at a group of actors that are constrained by an RTsynchronizer; they may also trigger changes of local states in an RTsynchronizer. Moreover, constraints may be nested within constraints. The syntax for constraints is defined in Figure 2.

Patterns are used to abstract over similar messages. A `pattern` is a quadruple:

$$\textit{pattern} ::= (\textit{sender}, i, \textit{target.method}(var_1, \cdots, var_n), \textit{boolean-expression})$$

```
constraint
  ::= assert {timeConstraint} [ exception constraint]
  |  {{pattern}*} trigger {{variable := expression;}*}

timeConstraint
  ::= time relation_op time
  | timeConstraint {&&, ||} timeConstraint
  | ¬ timeConstraint

relation_op
  ::= == | < | > | <= | >=

time ::= iTime(pattern) [{+, −} d]
  | wall-clock-time

iTime : Msgs → ℜ+

d, wall-clock-time  ∈ ℜ+ ∪ {0}
```

Figure 2: Syntax for Constraints

such that a message `m` matches the above `pattern` if:

1. `m`'s sender is the actor specified in pattern-quadruple as *sender*;
2. `m` is the ith message sent by actor *sender* (we say its *sequence number* is i);
3. `m`'s destination is the actor specified in the pattern as *target*;
4. `m` invokes the method specified in the pattern as *method*; and,
5. the values transmitted in `m` satisfy the boolean expression.

Note that: (1) the sender *sender* and its sequence number i are automatically tagged to each message by the system. This is transparent to an application programmer: actors implementing the functional behavior will not see the extra system level information. (2) The `boolean-expression` may reference an RTsynchronizer's local variables as well as the values transmitted by the message. (3) a message may be uniquely represented by a pattern. Pattern may be more general if they use wildcards, specifically for the sender and the sequence number.

Function `iTime(pattern)` defines the invocation time of a message which satisfies the given pattern `pattern` and has yet to be processed. Therefore it is always either the current time or some time in the future.

4.2 Informal Semantics

The functionality of RTsynchronizers is to dynamically enforce constraints based on the principles of *safe progress* and *unsafe wait*. That is, if a message captured by the constraint patterns in an RTsynchronizer can be invoked with a guarantee that it does not cause a violation of any constraints in the future, the RTsynchronizer will release the message to a scheduler and the system progresses. Otherwise, the message will be retained by the RTsynchronizer, which will wait for more information until it can determine that releasing the message will not violate constraints.

Assert `timeConstraint` **exception** `constraint` states that the temporal relation specified by `timeConstraint` should be enforced by the system. Specifically, scheduling of messages whose invocation times are related by the expression `timeConstraint` must be such that it satisfies the given relation; otherwise, an exception handler will be invoked. For example, an assertion:

assert { `iTime(p1)` $\leq$ `iTime(p2) + d` } **exception** `constraint`

where `p1` and `p2` are patterns, indicates that if a message `m1` satisfying pattern `p1` arrives first, it can be invoked immediately without violating the constraints. On the other hand, a message `m2` of pattern `p2` will be delayed unless and until a message `m1` of pattern `p1` is in the system or may be predicted to occur, possibly as a result of processing `m2`. In any case, the RTsynchronizer queries a scheduler if the two messages are schedulable with respect to the constraint. If the scheduler gives an affirmative response to the query from the RTsynchronizer, the RTsynchronizer releases `m2` and tells the scheduler to constrain the schedule so that `m1` is scheduled within `d` time units after `m2`'s invocation. If the scheduler gives a negative response to the query regarding the schedulbility of the messages, an exception is signaled. The exception handling strategy is specified in `constraint`, the constraint may be a relaxed assertion, or a trigger command that changes RTsynchronizer's local states and hence enables other actions.

Note that there are potentially satisfiable constraints that may not be scheduled in a given system, either because the arrival of a constraining message cannot be accurately predicted, or because a given scheduler is unable to find a way to satisfy the specified constraint in a realistic amount of time.

The difference between the conjunction of two timing constraints in one **assert** commands rather than in separate two different **assert**s is that RTsynchronizer requires that in the first case, both timing constraints must be satisfied before any of the messages are scheduled, while in the second case, the messages in the two constraints may be independently scheduled. For instance, with **assert**{`p1 <= p2 + t1 && p3 <= p4 + t2`}, an RTsynchronizer will determine a schedule for four messages, each matching one pattern. However, with **assert**{`p1 <= p2 + t1`}; **assert**{`p3 <= p4`

`+ t2}`, the schedulbility of messages satisfying `p1` and `p2`, or messages satisfying `p3` and `p4` will be determined independently.

Trigger and RTsynchronizer Local State Change

The values of RTsynchronizer's local variables may be reassigned; **Trigger** is used to change the state of an RTsynchronizer. A message which satisfies a pattern in the **trigger** command is invoked at the current time, while the RTsynchronizer atomically changes its local states as specified in the command. For example, `{(*,*, B.mthd, true)}` **trigger**`{msgCount++;}` counts the number of messages invoked at receiver `B` on method `mthd`. If there are several messages satisfying the patterns, we assume per message mutual exclusion to guarantee that the RTsynchronizer has a consistent local state. Thus, while the invocation order of such messages may be arbitrary, it must be serializable. Similarly, if there is a message that satisfies several **trigger** commands, only the first one in the RTsynchronizer is chosen and executed.

When a message satisfies both a pattern inside an **assert** command and a pattern inside a **trigger** command, it cannot be invoked until the constraints specified by the **assert** are satisfied, i.e., the state of the RTsynchronizer cannot change before the constraint has been satisfied.

Exceptions

The syntax given in Figure 2 only allows constraints to be nested in the **exception** part. Exception constraints are not visible to messages and hence are not enforced until the corresponding assertion fails. For example, in

assert`{iTime(p1)` $\geq$ `iTime(p2) - t}` **exception** **assert**`{iTime(p1)` $\leq$ `T}`

the nested constraint **assert**`{iTime(p1)` $\leq$ `T}` is usually not visible. However, if at the current time and with the current RTsynchronizer local state, the constraint **assert**`{iTime(p1)` $\geq$ `iTime(p2) - t}` is not satisfiable, either because it is infeasible or because a schedule cannot be found, the nested constraint is exposed to messages yet to be processed.

Multiple RTsynchronizers

Multiple constraints that are conjunctively enforced inside one RTsynchronizer also work conjunctively with multiple RTsynchronizers. However, in general, each RTsynchronizer constrains a group of patterns that are related to each other only inside

the RTsynchronizer, but independent of patterns inside other RTsynchronizer. When RTsynchronizers enforce their constraints on non-intersecting groups of actors, they may act autonomously. Moreover, RTsynchronizers may be distributed at different locations.

5 Examples

We show a few representative examples using RTsynchronizer. With RTsynchronizer, functional behaviors and quantitative temporal behaviors are specified separately; moreover, the temporal behavior depends only on the interface, not the representation of actors involved. These examples illustrate that we can reuse generic objects in real-time applications; reuse the correct timing constraints on different groups of objects as long as the objects have the required interface. In other words, with RTsynchronizers, modular specification of timing constraints becomes possible.

Example 1: Real-Time Synchronization

In distributed systems, we may require that two or more different actions conducted by different objects have to happen *'simultaneously'*, i.e., within some negligible time difference. For example, a robot pass a glass from one hand to the other. The release of the glass from one hand has to happen simultaneously with the grasping of the glass by the other hand. The timing constraint for the two robot hands may be specified as follows:

```
RTsynchronizer RTSyn( actor: lHand, rHand) {
 Constrain
  assert {(*, *, lHand.release, true) == (*, *, rHand.grasp, true)}
}
```

Since we do not care which object issues the command to release (grasp) the glass, we merely require that any release action be paired with a grasp action and that the two actions take place simultaneously.

Example 2: Periodic Events

Periodic events are a common paradigm in the real-time world. In the Actor model, a recurrent event may be modeled by an actor sending itself a message in response to a message of the same pattern. An RTsynchronizer may fix the recurrence interval and therefore make the invocation periodic.

```
RTsynchronizer PeriodicE( actor: O;  real: P) {
        /* P is the period. */
  Constrain
   assert{(O, *, O.m(x1,...,xn), b_exp1) == (O, *, O.m(x1,...,xn), b_exp2) + P }
  }
```

Example 3: Time Bounded Buffer

Consider an extension to the traditional producer/consumer with an unbounded buffer problem, where each product cannot be held inside the buffer longer than time t. We define an RTsynchronizer to specify the timing requirement without modifying the representation of generic unbounded buffers.

```
RTsynchronizer TBSyn( actor: TBbuffer;  real: t) {
       /* t is the time limit. */
  Constrain
   assert {(*, *, TBbuffer.get, true) <= (*, *, TBbuffer.put, true) + t}
 }
```

Another extension to the buffer problem is the following: there are two types of products produced by two different producers. Different types of products have different time limits for containment inside the buffer before a consumer consumes them. We can declare an RTsynchronizer `ITBSyn` to specify this requirement.

```
RTsynchronizer ITBSyn( actor: TBbuffer, Prdc1, Prdc2;  real: t1,t2) {
   /* t1 and t2 are the time limits for different products. */
  Constrain
   assert {(*, *, TBbuffer.get, true) <= (Prdc1, *, TBbuffer.put, true) + t1};
   assert {(*, *, TBbuffer.get, true) <= (Prdc2, *, TBbuffer.put, true) + t2}
 }
```

By using a product manager actor as a parameter, one can generalize the RT-synchronizer to a dynamic set of time limits, where the times (`t1, t2, ...`) in the constraint are replaced by queries to the product manager.

Example 4: Air Traffic Control System

Consider an air traffic control system where there are two control towers, a primary and a back-up. A pilot gets information from the control towers frequently. If the pilot does not get the requested information from the primary control tower within $t1$ time units, he or she has to obtain the information from the back-up control tower. After trying the alternate tower, if within $t2$ time units information still has not arrived, the pilot must perform some emergency procedure within $t3$ time units after

getInfo failed, since unbounded waiting may cause a fatal accident. The problem may be abstracted in the following way.

Two actors Ctrl1 and Ctrl2 model the two control towers. The two control actors have sending information methods sendInfo. Actor pilot models the pilot. It has a method of getting information from the control towers, getInfo, with period p, and a method to do emergency actions emgAction. We declare ATCS RTsynchronizer to program the timing constraints for the air traffic control system. For safety reasons, we assume that the pilot is prepared for an emergency action once the flight takes off. In other words, when the ATCS RTsynchronizer is created, the timing exception handling message is sent.

```
RTsynchronizer ATCS( actor: Pilot, Ctrl1, Ctrl2;  real: t1, t2, t3, p) {
   /* t1, t2 are the time limits to get info from Ctrl1, and Ctrl2, respectively;
   t3 is the limit to do emergency action; pilot gets info at frequency 1/p. */
 Declare
  Boolean: danger := false
  Real: last
 Constrain
  assert{(Pilot, *, Pilot.getInfo, true)==(Pilot, *, Pilot.getInfo, true) + p};
    /* periodically get information from control tower */
  assert {(*, *, Ctrl1.sendInfo, true) <= (*, *, Pilot.getInfo, true) + t1}
     exception
      assert {(*, *, Ctrl2.sendInfo, true) <= (*, *, Pilot.getInfo,true) + t2}
         exception {(*, *, Pilot.getInfo, true)}  trigger{
                                danger := true;
                                last = iTime((*, *, Pilot.getInfo, true))};
       /* limit time for getting the info, or trigger the emergent action. */
  assert {(*, *, Pilot.emgAction, danger == false) <= 0};
    /* not in danger, disable the emgAction. */
  assert {(*, *, Pilot.emgAction, danger == true) <= last + t3}
    /* constrain the emergent action time. */
}
```

Note that, because RTsynchronizers are purely declarative and may only affect message scheduling, they cannot cause a side effect by actually sending an exception handling message. The above example also illustrates how the specification of timing exceptions presents an interesting problem. Other kinds of timing constraints affect only the order of message execution (i.e. the scheduling behavior) and thus do not involve the occurrence of any event that could not possibly have happened otherwise. In other words, they simply reduce the nondeterminism in the system. On the other hand, exception handling for timing faults may involve adding a behavioral response that does not already occur in the actor. In the above example, a system without real-time requirements is unlikely to have a pending exception message emgAction.

6 Implementation

We use real-time synchronization (see Example 1 above) as an example to illustrate how timing constraints on synchronization between objects may be implemented. Although we focus on real-time synchronization, our methodology is applicable to other synchronization patterns with timing constraints. Notice that our focus is on the issues related to the interaction between different schedulers (residing in an actor or a group of actors) to generate schedules satisfying the quantitative synchronization constraints [29].

Our approach is to divide synchronization into two phases: a negotiation phase and a local scheduling phase. In the negotiation phase, all participating objects communicate with one another to determine the time at which they are going to resume their message invocation. We also call this phase *global scheduling phase*. In the *local scheduling phase*, tasks are scheduled to execute simultaneously on local processors given the constraints imposed by global scheduling. The real-time synchronization example may be generalized as *real-time barrier synchronization*.

Real-time barrier synchronization is a barrier synchronization constrained by two timing values: *earliest release time* and *release time skew*. Earliest release time is the delay from the time the last object participating in barrier synchronization issues a barrier synchronization operation, until one of the participating objects resumes its method execution. Release time skew is the delay from when the first object resumes its method until all the objects resume their methods (Figure 3). Earliest release time mainly depends on communication delay and reliability of communication paths, while release time skew depends on local clock skew and local scheduling policy.

One of the software architectures suitable for implementing a system of actors (and by implication, any systems of distributed objects) with real-time scheduling, fault-tolerance and complex communication such as broadcast and multi-cast, is computational reflection [26, 4, 5]. Reflection is the model where an object can manipulate a causally connected description (i.e. meta-object) of itself. A change of the meta-object results in a change of implementation or behavior of the object.

In the Actor Model, a *mail queue* is thought of as a meta-actor (Figure 4). By changing data in the mail queue, for example, we can reverse the execution of two method invocations or can invoke a special method such as an emergency method. The instantiation of local scheduling policies is *local scheduling meta-actors*. We assume that a scheduling meta-actor exists on a computer and has the role of negotiating with the other scheduling meta-actors, determining the resumption time, and allocating resources such as CPU and memory segments on actors.

The implementation of real-time barrier synchronization is as follows (Figure 4): barrier messages, the messages that invoke the barrier synchronization, are sent to the mail queue meta-actors. One task of mail queue meta-actors is the instantiation of real-time barrier synchronization. These meta-actors do pattern matching

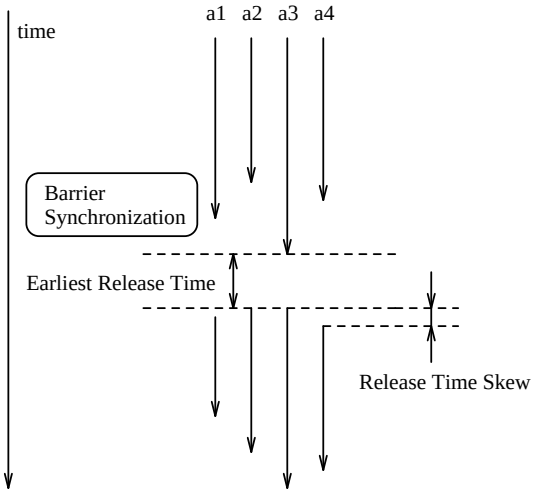

Figure 3: Time Values Related to Barrier Synchronization.

between received messages and specified patterns that require special treatment. If the received messages match the specified patterns of barrier synchronization, they inform the scheduling meta-actors of the other participants of barrier synchronization. Upon receiving this information, the scheduling meta-actor will negotiate with other scheduling meta-actors and allocate resources on actors based on the timing constraints imposed by the negotiation phase. If necessary, the scheduling meta-actor lets the mail queue meta-actor change the order of messages in the queue.

In the negotiation phase, the major work is to decide the time when participating objects resume their operations. The estimated release time can be derived as follows:

$$EstimatedReleaseTime = T + D_{comm} + (N_{msg} - 1) * D_{rep} + T_{rep} + D_{sched} + T_{skew}$$

where

T:	*present time*,
D_{comm}:	*worst case communication delay*,
N_{msg}:	*number of negotiation messages*,
D_{rep}:	*delay between negotiation messages*,
T_{rep}:	*negotiation message handling time*,
D_{sched}:	*scheduling delay*, and,
T_{skew}:	*maximum local clock skew.*

Each participating object sends the calculated estimated release time to all participating objects. Upon receiving the calculated estimated release time from all other participants, each participant chooses the maximum estimated release time as its re-

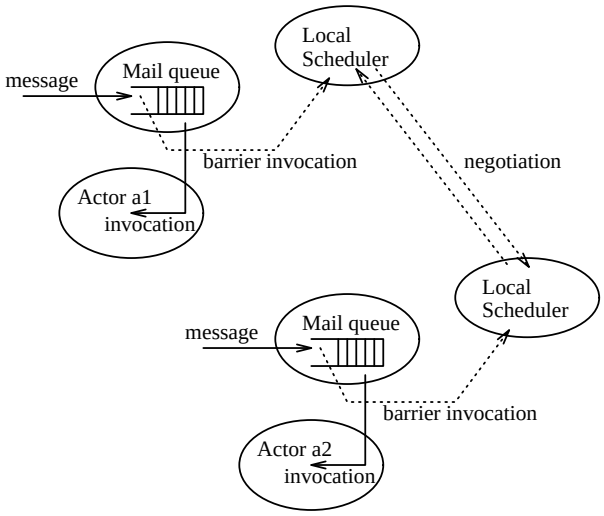

Figure 4: Software Architecture of Actor-Based Real-Time System.

lease time. This release time, which is agreed by all participating objects, is imposed on the local scheduling phase.

In the scheduling phase, we use the *synchronized sporadic server* approach to make scheduling more flexible. Synchronized sporadic servers are an extension of the sporadic server policy [16] which provides good responsiveness for aperiodic tasks. Since we are assuming that local clocks are synchronized, the sporadic servers will be activated simultaneously on each node if they are scheduled to execute at the same wall clock time. We regard actors invoked by barrier messages as aperiodic tasks. By redefining release time using the earliest activation time of the synchronized sporadic servers, it is possible to invoke a set of actors simultaneously. Alternatively, instead of redefining the release time, we could incorporate the activation delay of synchronized sporadic servers in the scheduling delay (D_{sched}) when calculating estimated release time.

7 Conclusions

We have argued that appropriate linguistic mechanisms for distributed real-time systems can simplify their design, implementation and reasoning. Specifically, such mechanisms need to separate a real-time program's functional behavior from its timing requirements. We have proposed an abstraction mechanism, RTsynchronizer, for the declarative specification of timing constraints among groups of actors. An RTsynchronizer abstracts the timing constraints from individual actors and isolates those

constraints into timing-specific modules. Our approach increases code reusability: one need not re-implement operational code when objects exhibit the same logical behavior. Likewise, we do not need to repeatedly specify timing constraints when the same timing constraints can apply to different groups of actors. Moreover, RTsynchronizers allow incremental modification of timing constraints. Because timing constraints are not intermixed with the application code, programs with RTsynchronizers are also easier to analyze.

We have shown that we may use meta-architecture's reflection mechanism and a two-phase scheduling procedure to implement RTsynchronizers. Although we have given only one implementation example, the approach is applicable to other synchronization patterns with timing constraints.

However, note that some timing constraints may be infeasible to implement, even given sufficient required resources. As in the real-time synchronization example, if only the execution of releasing a glass by one hand causes the execution of grasping the glass by another, it is impossible to realize simultaneity. In other words, RTsynchronizers can be specified independently of application code but such specifications depend on the structure of an application's functional behavior. A goal of our research is to develop methods to analyze the static information given by application code and detect infeasible constraints.

References

[1] *Aristotle — Physica — Laterza.* Bari, 1991.

[2] Ada95 6.0 reference manual. Technical report, Internet's Public Ada Library, 1995.

[3] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[4] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[5] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, (2):3–14, 1993.

[6] G. Agha, I. A. Mason, S. F. Smith, and C. .L. Talcott. Towards a theory of actor computation. In *The Third International Conference on Concurrency Theory (CONCUR'92)*, pages 565–579. Springer Verlag, August 1992. LNCS 630.

[7] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1996. to appear.

[8] G. Agha, P. Wegner, and A. Yonezawa, editors. *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[9] M. Aksit and L. M. J. Bergmans. Composing and reusing synchronization and real-time specification. In *The Object-Oriented Real-Time Systems (OORTS) Workshop*, pages 13–22, San Antonio, Texas, October 1995. to appear in OOPS Messeuger, ACM SIGPLAN.

[10] M. Aksit, J. Bosch, and W. Sterren. Real-time specification inheritance anomalies and real-time filters. In *Proceedings of ECOOP'94*, pages 386–407. Springer Verlag, July 1994. LNCS 821.

[11] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11), 1983.

[12] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1993.

[13] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, pages 55–94. ACM Press, 1993.

[14] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[15] W. Clinger. Foundation of actor semantics. Technical Report AI-TR-633, MIT, 81.

[16] J. Lehoczky et al. Enhancing aperiodic responsiveness in a hard real-time. In *Proc. IEEE Real-Time Systems Symp.*, 1987.

[17] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proceedings of ECOOP 1993*, volume 707 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[18] Svend Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.

[19] G. Goos and J. Hartmanis (ed.). *The Programming Language Ada Reference Manual*, February 1983.

[20] A. S. Grimshaw, A. Silberman, and J. .W. S. Liu. Real-time mentat at programming language and architecture. In *IEEE Globecom'89*, 1989.

[21] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, pages 97–138. ACM Press, 1993.

[22] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[23] Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer. Object-oriented real-time language design: Constructs for timing constraints. In *Proceedings OOPSLA/ECOOP '90*, pages 289–298, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[25] K. Lin and J. W. Liu. Flex: A language for real-time systems programming. Technical Report 1634, UIUC, 1990.

[26] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87*, pages 147–155, December 1987. Published as ACM SIGPLAN Notices, volume 22, number 12.

[27] B. Nielsen and G. Agha. Semantics for an actor-based real-time language. In *Fourth International Workshop on Parallel and Distributed Real-Time Systems*, Honolulu, April 1996. (to be published).

[28] O. Nierstrasz and M. Papathomas. Viewing objects as patterns of communicating agents. In *OOPSLA '90 Proceedings*, 1990.

[29] M. Saito and G. Agha. A modular approach to real-time synchronization. In *Object-Oriented Real-Time Systems Workshop*, pages 13–22, San Antonio, Texas, October 1995. to appear in OOPS Messeuger, ACM SIGPLAN.

[30] F. A. Schreiber. Is time a real time? an overview of time ontology in informatics. In W. A. Halang and A. D. Stoyenko, editors, *Real Time Computing*. Springer-Verlag, 1994.

[31] S. Tucker Taft. Ada 9x: From abstraction-oriented to object-oriented. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 127–143, October 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[32] Kazunori Takashio and Mario Tokoro. DROL: An object-oriented programming language for distributed real-time systems. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 276–297, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[33] N. Wirth. Towards a discipline of real-time programming. *Communications of the ACM*, 20:577–583, 1977.

Biographies

Shangping Ren is a doctoral candidate and research assistant in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign. Her research interests include programming languages and parallel distributed real-time computing. She received her BS and MS from Department of Computer Engineering at Hefei Polytechnic University, China.

Gul A. Agha is Director of the Open Systems Laboratory at the University of Illinois at Urbana-Champaign and Associate Professor in the Department of Computer Science. Dr. Agha is the Editor-in-Chief of *IEEE Parallel and Distributed Technology*, and Associate Editor of the journals *ACM Computing Surveys* and *Theory and Practice of Object Systems*, and an ACM International Lecturer. Dr. Agha past recipient of the Incentives for Excellence Award from Digital Equipment Corporation, and the Naval Young Investigator Award from the US Office of Naval Research. He was named a Fellow in the University of Illinois Center for Advanced Study in 1992. Dr. Agha has given over eighty invited lectures at universities, industrial laboratories, and international conferences in the past five years.

Masahiko Saito is a Researcher of the Hitachi Research Laboratory, Hitachi, Ltd. His research interests are distributed real-time systems, reliability and object- oriented techniques, and he is currently involved in fault-tolerant distributed control systems. He received his M.Sc. in computer science from the Kyoto University, Japan, in 1988. He was a Visiting Researcher in the Open Systems Laboratory at University of Illinois at Urbana-Champaign from 1994 to 1995.