ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, La Jolla, California, June 1995.

RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems *

Shangping Ren and Gul A. Agha
Department of Computer Science
1304 W. Springfield Avenue
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

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

Abstract

We argue that the specification of an object's functional behavior and the timing constraints imposed on it may be separated. Specifically, we describe RTsynchronizer, a high-level programming language construct for specifying real-time constraints between objects in a distributed concurrent system. During program execution, RTsynchronizers affect the scheduling of distributed objects to enforce real-time relations between events. Objects in our system are defined in terms of the actor model extended with timing assumptions. Separation of the functional behaviors of actors and the timing constraints on patterns of actor invocation provides at least three important advantages. First, it simplifies code development by separating design concerns. Second, multiple timing constraints can be independently specified and composed. And finally, a specification of timing constraints can be reused even if the representation of the functional behavior of actors has changed, and conversely.

A number of examples are given to illustrate the use of RTsynchronizers. These examples illustrate how real-time constraints for periodic events, simultaneous events, exception handling, and producer-consumer may be specified.

*The research described has been made possible 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, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195). The authors would like to thank Mark Astley, Brian Nielsen, Masahiko Saitoh and Daniel Sturman for helpful discussions concerning the manuscript.

1 Introduction

Existing real-time programming languages typically intermix the programming of timing constraints with the programming of functional behaviors. This complicates the design of real-time systems as well as reasoning about them. One approach to solve this difficulty is to observe a programming discipline which confines specification of temporal constraints to certain isolated components in some standard pattern [23]. We take a more linguistic approach (similar in spirit to the work of Aksit et al. [8]): the goal of our research is to allow a separation of the two different design concerns in order to support configuration of components with real-time constraints.

We model components of a distributed real-time system, such as control processes and hardware devices, uniformly as actors. Our view is that the specification of a real-time system can be decomposed into the specifications of the functional behavior of components and that of the timing relations between them. We use a general purpose actor programming language to specify functional behaviors, and extend the language with a construct called RTsynchronizer's for specifying the timing relations.

We are currently working on the problem of composing and executing these two kinds of specifications. This work involves the development of a compiler which would translate a program written in a general purpose actor language, together with a collection of RTsynchronizer's over actors defined by the program, into executable code. Ideally, the compiler would be portable for concurrent computers belonging to a given class of architectures. We believe that such portability may be accomplished

by developing a model for a run-time system which represents resources abstractly. A particular concurrent computer would then be specified by quantifying its resources. Figure 1 summarizes how we envision the process of designing and developing real-time systems.

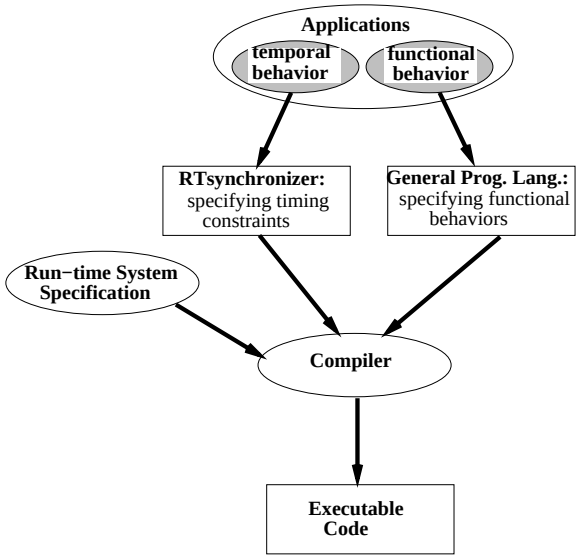

Figure 1: Real-Time System Development

The goal of this paper is to describe the research at the language level. Specifically, we describe RTsynchronizers. An RTsynchronizer is a collection of declarative constraints which restrict the temporal behavior of events over a group of actors. Essentially, RTsynchronizers abstract over common temporal coordination patterns, allowing the programmer to think in more natural terms about distributed timing constraints.

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. Our approach is particularly useful for open distributed systems where temporal coordination patterns evolve in a number of ways as components are added, removed, or updated. A key advantage is that if different real-time requirements are to be imposed on functionally identical actors or groups of actors, the behavior of such actors need not be redefined. In summary, RTsynchronizers allow us to:

- simplify programming by separating timing design correctness and functional design correctness.
- incrementally modify timing constraints and the representation of components.

The outline of this paper is as follows. In the next section, we briefly review some related work, and the motivation for the design of a new model and new language constructs. In section 3, we describe the actor model and time-related properties about the model. In section 4, we will discuss the separation between functional behavior design and temporal behavior design, and define *RTsynchronizers*. Section 5 provides a number of examples which illustrate the expressive power of our language constructs. We draw the conclusion in section 6.

2 Related Work

Ada is a programming language intended for critical software systems. However, it is difficult to implement full Ada in distributed systems — not only because 'it is already too complicated' [19], but also because ADA's tasking model is integrated into a stack and heap model. This integration poses difficult issues such as global variable or procedure placement; and the meaning associated when a task allocated to one processor receives an absolute address of another, for example, due to a pointer passed during rendezvous. In addition to these shortcomings of Ada as a general distributed language, Ada as a real-time programming language poses additional problems: in Ada, *rendezvous* is the only mechanism for communication among two tasks and communication is synchronous and unbuffered. However, the semantics of rendezvous can only guarantee the readiness of two communicating tasks. No specification of real time is involved. *Delay* is the only language construct in Ada which deals with real time, but *delay* does not provide the exact 'wakeup' time; it only guarantees the earliest wakeup time, which is insufficient for specification of real-time systems. Moreover, Ada does not provide separation between logical correctness and timing correctness. The lack of modularity makes it hard to reuse the program code and therefore makes it very difficult to design and reason about large systems.

Some researchers have proposed integrating timing constraints with sequential objects. In particular, the real-time programming language Flex [24] extends C++ with a language construct called a *control*

block, which can reside in the body of methods; timing constraints are specified in the control block. For our purposes, there are two weaknesses of this work. First, the timing requirements are intermixed with the code for the methods of individual objects. Second, Flex does not support concurrency.

Concurrent object-oriented programming is an active area of research interest (for example, see [4, 16, 2]). Specifically, there are a number of real-time languages based on concurrent object-oriented languages.

Ishikawa et al. proposed a concurrent object-oriented programming language called *RTC++* [13]. 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 behaviors but with different temporal constraints require separate specification. Timing constraints in RTC++ may be specified as part of method declarations. However, the language also allows timing constraints to be specified in commands within methods. The failure to impose a stricter discipline can lead to an intermixing of code for specifying real-time requirements and functional code. Finally, 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. By contrast, by using the actor model, we avoid the possibility of inconsistencies within the state of an actor due to concurrent accesses.

From a linguistic point of view, the cleanest approach thus far appears to be the work of Aksit et al. [8, 9]: they introduce a linguistic mechanism called a *real-time filter*. Real-time filters extend an object's interfaces to include *input filters* and *output filters*. An out-going message has to go through a series of zero or more output filters before it is sent to its target. Similarly, an incoming message passes through input filters before it is received by the object. Timing constraints are encoded inside the 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. Note that both filters and messages are first-class objects.

The filter mechanism makes the complexity of programs more manageable. A potential difficulty with their model for temporal constraints appears to be that they do not take into account the message transmission time; the timing constraints may only be specified on the methods after they are invoked. However, in distributed systems, the message transmission time (as opposed to the computation time) can be quite significant. More fundamentally, our approach differs from the filters approach in that we provide declarative mechanisms which specify an abstract timing behavior over a group of actors rather than procedural implementation in terms of individuals actors.

There are a number of real-time specification languages, such as timed Petri Nets, Modechart, i-o automata, and RTL [26, 25, 27, 28]. At the application level, specification languages can assist in real-time system design. Moreover, specification languages provide support for reasoning about real-time systems. However, the purpose of our research is to provide high-level programming language constructs for developing real-time applications. Thus, we require that our specifications be executable and integrate with standard object-based programming techniques.

The primitive Actor model is insufficient for real-time programming: in the Actor model, message transmission time is non-deterministic and there is no unique global clock. We extend the Actor model to allow each actor to have a local clock, and further assume that local clocks of a temporally constrained set of actors are sufficiently synchronized to allow a meaningful interpretation of the constraints.

Our work builds on the earlier work of Frølund and Agha [17] who developed *synchronizers* to provide declarative specification of coordination constraints on groups of actors. Synchronizers may be used to enforce *qualitative* temporal ordering and indivisible (atomic) scheduling of multiple invocations at a group of objects. On the other hand, RT-synchronizers allow the specification of *quantitative* constraints.

RTsynchronizers are designed to be an executable programming language construct which work with the interface (and not the representation) of actors. RTsynchronizer use relation operation (such as $>=$, $>$, etc.) to relate events to time and logical operation (such as $\vee, \wedge$, etc.) to specify the temporal behaviors of a group of actors. As shown in Figure 1, a compiler for real-time compiler must take this time-constraint information, as well as information about the functional behavior of actors transform it into executable code for a given concurrent computer. The transformation involves scheduling and possibly placement of actors; it does not affect their functional behavior in response to a particular message.

3 The Real-Time Actor Model

Actors are concurrent objects which interact by sending buffered, asynchronous messages [1, 3]. We will call the processing of a message by an actor an *invocation* of the actor by the message. Each actor has a unique mail addresses which may be used to send it messages. Note that a 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 address. Finally, actors may change their local behavior. Such change of state affects only the next message processed. Thus each invocation of an actor is atomic and uninterruptable. One way to think of actors is as a coordination model where the local behavior of an actor may be specified in an arbitrary programming language with actors providing concurrency and object-style encapsulation.

The actor model has a number of advantages in modeling distributed real-time systems. First, the lack of interruptability of invocations guarantees that the computation time will only depend on the behavior of the invoked actor and the content of the invoking message. Thus, computation time is a deterministic function of incoming messages. Second, the Actor model only requires the specification of the logical order of events; it thus allows scheduling to be separately specified.

However, there are some extensions which need to be made to actors in order to provide a satisfactory model for real-time systems. We extend the Actor model to provide each actor with a local clock and further assume that these local clocks are sufficiently synchronized. Note that this synchronization assumption requires that the scale in which timing constraints are expressed is sufficiently coarse to allow us to ignore, firstly, *relativistic* effects, and secondly, potential sources of clock drift. We further assume that a suitable meta-architecture supports time scheduling to satisfy feasible timing constraints. Note that such scheduling needs to take into account communication delays. Clock synchronization, scheduling and compilation mechanisms are beyond the scope of this paper.

We assume that each message in the system is unique (for example, because it is tagged by sender and its local clock time). Corresponding to each message is a time when the receiving actor (called the *target*) processes the message. We call this the *invocation time* for the message. We will assume that the function `iTime(msg)` returns the value of Time when `msg` invokes the target actor. Timing constraints are specified by asserting relations between the projection of the function *iTime* on a message and other timing variables. Messages are sent by actors in response to messages they receive. We assume that the function `cause(msg)` returns the message `msg1` which caused `msg` to be sent. Note that a single message may be the cause of several messages but that each message has a unique cause.

4 *RTsynchronizer*

We define the language construct, RTsynchronizer, to abstract the timing relations between actors. RTsynchronizers express timing constraints between multiple objects. An RTsynchronizer is like an ordinary actor in that it has a mutable state and may be dynamically created. However, unlike ordinary actors, an RTsynchronizer does not send or receive messages, rather it provides a declarative specification of the quantitative temporal ordering between certain message invocations at a group of actors. Note that the implementation of an RTsynchronizer need not be centralized.

An RTsynchronizer is created by instantiating a *template* using the mail addresses of specific actors. Thus:

new my-RT (**actor**:a1, a2, a3; **int**: d1, d2)

creates a new RTsynchronizer whose behavior is given by `my-RT` and which observes and possibly constrains invocations of 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 `my-RT` above) are specified by giving a template as follows:

RTsynchronizer $MySyn(var_1, \cdots, var_n)$ {
Declaration
`\* declare and initialize local variables; *\`
Rules
$rule_1$;
$\cdots$;
$rule_n$
}

Rules

Rules apply to groups of messages satisfying given patterns. Processing a group of messages excludes

the concurrent processing of another group of messages where such processing may cause an RTsynchronizer's state to be inconsistent. Moreover, a single rule is instantiated exclusively for a group of messages that has yet to be processed — i.e., the same message cannot be used to satisfy the same rule for two different groups of messages. Different rules inside one RTsynchronizer and within different RTsynchronizers are conjunctively applied. Note that programmers may specify inconsistent timing constraints; in this case, the constraints will not be satisfiable. However, by isolating timing constraints, RTsynchronizers raise the level of abstraction at which one needs to reason about timing constraints

A rule is specified as follows:

rule ::= **with** { *match*$^+$ }
{{ *constraint;*}$^+$ }

match ::= (id: *pattern*)

A message satisfying a pattern in a given *match* is bound to the `id` specified by the corresponding match within the scope of the constraint which follows. Each rule is instantiated for every group of messages matching its respective pattern. Messages are separated into non-intersecting sets which satisfy match patterns and a rule is instantiated for each such set. We explain patterns and constraints below.

Patterns

A pattern is defined over message contents and target as follows:

$$pattern ::= actor.method(var_1, \cdots, var_n) \textbf{ sat } boolean\text{-}expression$$

where a message matches a given pattern if:

1. its target is the actor specified in the pattern;
2. it invokes the method specified in the pattern; and,
3. the values transmitted in the message satisfy the boolean expression.

Constraints

Constraints may make temporal assertions on the group of actors constrained by an RTsynchronizer and may trigger a change of the local state of an RTsynchronizer. Moreover, rules may be nested within an RTsynchronizer. The syntax for constraints is defined as in Figure 2.

```
constraint
  ::=  assert timeConstraint
  |  trigger {{variable := expression;}*}
  |  if exp  then constraint
           [ else constraint]
  |  rule

timeConstraint
  ::= time relation_op time
  | timeConstraint {∧, ∨} timeConstraint
  | ¬ timeConstraint

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

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

d, wall-clock-time  ∈ ℜ+ (including 0)
```

Figure 2: Syntax for Constraints

Assert `timeConstraint` states that the temporal relation specified by `timeConstraint` is to be enforced by the system. Specifically, the scheduling of messages whose invocation times are related by the expression `timeConstraint` must be such that it satisfies the given relation. **Trigger** is used to change the state of an RTsynchronizer: the values of its local variables, declared in the declaration section, may be reassigned.

5 Examples

In this section, we use examples to demonstrate the expressive power of *RTsynchronizers*. We focus on how quantitative timing constraints are enforced on events (message invocations).

Periodic Events

The periodic event is a very common phenomenon in the real-time world. For example, a sensor has to get data every P time units. A recurrence event in Actor model may be modeled by an actor sending itself a message in response to a message of the same pattern. An RTsynchronizer may fix the time interval between recurrence thus making the invocation periodic. periodicity.

```
RTsynchronizers PeriodicE( actor:O;  int:P) {
        P is the period
  Rules
   with { (msg1: O.m(x1, ···, xn)) }
     { if (cause(msg1)  sat O.m(x1, ···, xn))
        then  assert
        {iTime(msg1) == iTime(cause(msg1))
                        + P } }
  }
```

Simultaneous Events

In distributed systems, we sometimes need to describe a situation where two or more different actions conducted by different objects have to happen *'simultaneously'*, i.e., within some negligible time difference. For example, two robot hands 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. We may express this synchronization behavior as follows:

```
RTsynchronizer SimuE( actor:O1,O2) {
 Rules
  with {(rls: O1.release) (grsp: O2.grasp)}
   { assert { iTime(rls) == iTime(get) } }
}
```

Timing Exceptions

Consider in an air traffic control system, where a pilot has to frequently get information from the airport control tower. If the pilot does not get the requested information within a certain time, he or she has to either request the information from another control tower, or do some emergency procedure. Unbounded waiting may cause a fatal accident. The problem may be abstracted as follows.

Assume `O2` is an exception handling actor. `O1.m1` has to happen before time `T`. If not, `O2` will be activated with a call to method `m2`. We assume that an RTsynchronizer is created and an exception message of the form `O2.m2` is sent. `O2.m2` will remain disabled as long as a message is received within time `T`, otherwise it will be invoked at time `delta` after time `T`. Note that, because *RTsynchronizers* are purely declarative and may only affect message scheduling, it cannot cause a side effect by actually sending an exception handling message.

The example illustrates how the specification of timing exception 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 have otherwise possibly happened. 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.

```
RTsynchronizer ExceptSyn( actor: O1,O2;
                          real: T, delta) {
 Declaration
  Boolean: activate := false
 Rules
   with { (info: O1.m1) (sig: O2.m2)}
    { if ¬(iTime(info) <= T)
       then {
        trigger { activate := true };
        assert {iTime(sig) = T + delta} }
         exception handling must be invoked
   with { (sig: O2.m2) }
    { if (activate == false)
       then {
        assert { iTime(sig) < 0 } } }
         enforce msg1 not to be invoked
         when the exception handler actor
         O2 is not activated
}
```

Producer/Consumer with Time-Limited Buffer

Consider an extension to the traditional producer/consumer with an unbounded buffer problem, where each product can only be held for a fixed amount of time after it has been produced. We define an RTsynchronizer to specify the problem without changing the logical behaviors of the producer or consumer:

```
RTsynchronizer ProdCons( real: delta;
                         actor: Producer, Consumer) {
  Rules
   with {(put: Producer.put)
         (get: Consumer.get)}
    { assert
     {iTime(get) <= iTime(put) + delta }}
 }
```

Processes Control Example

Selic et al. [21] describe a dye control industrial process as a canonical process control problem.

The problem is as follows. There is a sensor set which includes three sensors: a liquid level sensor (`SensorL`), a temperature sensor (`SensorH`) and a pressure sensor (`SensorP`); the sensors responsible for reading the dyeing liquid level, temperature, and pressure, respectively. These sensors also interact with the devices which are relevant to their functionality. There is a dye valve (`Dye`) controlling the incoming dye liquid and drain valve (`Drain`) controlling the out-going dye liquid. Finally, there is a heater controller (`Heater`) and a pressure controller (`Pressure`) (Figure 3).

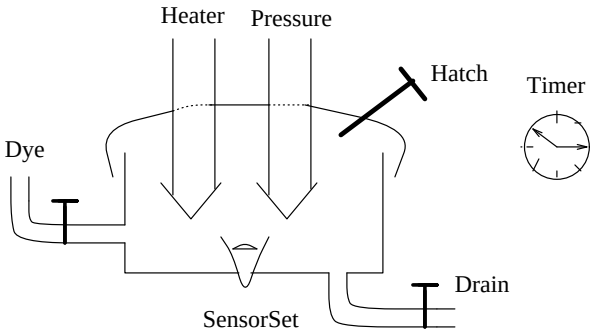

Figure 3: Dyeing Machine

The following constraints must be satisfied:

1. As soon as the dyeing liquid reaches level L, the temperature reaches H, and pressure reaches P, the Timer will start timing;
2. After T time units, the Timer will signal that the dyeing process is done;
3. The dyeing liquid has to be drained out through the drain valve within time D after the Timer sends out the finishing signal;
4. For safety reasons, the temperature and the pressure in the tank have to be reduced to room temperature and room pressure, respectively, before the dyeing water can be allowed to drain out.
5. Timer periodically (with period p) reports its time;
6. All three sensors periodically read their data with period $p1$;
7. When a sensor finds that the requirements are not satisfied, it signals the corresponding devices. The devices have to make the adjustment within time t.

We model Dye/Drain valve, Heater, Pressure, Hatch, Sensors and Timer as corresponding actors. The qualitative relationships between these actors, such as Drain/Dye cannot open when Heater/Pressure's `heat, cool/increasePressure, decreasePressue` method is in processing, may be simply expressed by synchronizers [17]. We focus on the quantitative time constraints represented by the RTsynchronizers.

Specifically, we use four RTsynchronizers to represent the timing requirements listed above: `SttSyn` is responsible for starting and terminating the dyeing process; `SafetySyn` enforces the safety properties; `PeriodSyn` enforces the timer, and the sensors repeat their functions periodically; `ReactSyn` enforces the requirement that devices react to messages within the applicable timing constraints Figure 4 to Figure 7).

These examples show that RTsynchronizer may provide a clean, simple and uniform construct for specifying timing constraints in the real world.

6 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 behaviors from its timing requirements. We have proposed an abstraction mechanism RTsynchronizer for the declarative specification of timing constraints over groups of actors. An RTsynchronizer abstracts the timing constraints over from individual actors and isolates those constraints into one module. Our approach increases code reusability: one need not repeat operation code when objects exhibit the same logical behavior. Moreover, we do not need to repeatedly specify timing constraints when the same timing constraints can apply to different groups of actors. Furthermore, because the timing constraints are not scattered all over the code, programs with *RTsynchronizers* are much easier to analyze.

While we are convinced of the advantages of a linguistic mechanism for abstracting real-time constraints, considerable research remains to be done to make our approach practical. This research involves understanding the formal semantics of *RTsynchronizers* and developing efficient implementation mechanisms. Current research in our group is focused in these areas.

```
RTsynchronizer SttSyn( actor: SensorL, SensorH, SensorP, Timer;  real: L, H, P, T, D){
     L, H, P are required parameters for level, temperature
    and pressure, respectively; T, D are time limits.
  Declaration
   Real: level, temp, pressure;
  Rules
   with {
     (reachL: SensorL.sig(level)  sat level == L),
     (reachH: SensorH.sig(temp)  sat temp == H),
     (reachP: SensorP.sig(pres)  sat pres == P),
     (start: Timer.start)  }
          { assert{ iTime(start) == max(iTime(reachL),iTime(reachH), iTime(reachP))};
            (1) as soon as the dyeing liquid volume, temperature,
             and the pressure reach the required level, the timer starts its timing

              with {(alarm: Timer.alarm)}
                 { assert{ iTime(start) + T == iTime(alarm) };
                      (2) after the elapsed time reaches T,the Timer alarms

                          with {(empty: SensorL.sig(level)  sat level == 0)}
                            { assert{ iTime(empty) <= iTime(alarm) + D ∧
                                      iTime(empty) >= iTime(alarm) }}
                              (3) the liquid has to be drained out with within D time after Timer alarms.
                 }}
 }
```

Figure 4: Starting and Terminating RTsynchronizer

```
RTsynchronizer SafetySyn( actor: SensorH, SensorP, Drain;  real: rmT, rmP){
   rmT is the room temperature; rmP is the room pressure
  Declaration
   Real: level, temp, pressure
  Rules
       with {(currentH: SensorH.sig(h))}
        { trigger { temp := h } }
       with {(currentP: SensorP.sig(p))}
        { trigger { pressure := p } }
       with {(openDrain: Drain.open  sat ((temp > rTemp) ∧ (pressure > rPre)))}
        { assert { iTime(openDrain) < 0 } }
          (4) safety requirement: Drain should not be opened
           if the liquid temperature or pressure is above the room value
 }
```

Figure 5: Safety RTsynchronizer

```
RTsynchronizer PeriodSyn( actor: Timer, SensorL, SensorT, SensorP;  real: p, p1){
 Rules
      with {(reportT: Timer.reportTime)}
       { if (cause(reportT)  sat Timer.reportTime)
           then  assert{iTime(cause(reportT)) + p == iTime(reportT)}}
                (5) every p time, the Timer reports the time.

      with {(reportL: SensorL.read)}
       { if (cause(reportL)  sat SensorL.read)
           then  assert {iTime(cause(reportL)) + p1 == iTime(reportL)}}
               (6) every p1, the SensorL reads the liquid level;similar control for SensorH, SensorP.
```

Figure 6: Periodic RTsynchronizer

```
RTsynchronizer ReactSyn( actor: SensorL, SensorH, SensorP;
      actor: Heater, Pressure, Drain, Dye;
      real: L, T, P, t1, rmT, rmP){
 Declare
  Real: level, temp, pre
 Rules
      with {(levelLow: SensorL.sig(level)  sat level <= L),
              (openDye: Dye.open)}
       { assert {iTime(openDye) <= iTime(levelLow) + t1 }}
         (7) when the sensor senses that the requirements
          are not satisfied, the corresponding device has to
          do the adjust within t1 time.
              ....
      with {(cool: Heater.cool(temp)  sat temp < rmT)}
      {  assert { iTime(cool) < 0 }}
         when the temperature is already below room temperature, discard the cool message
      ...
  }
```

Figure 7: React RTsynchronizer

References

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

[2] 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 **1** (1993), no. 2, 3–14.

[3] G. Agha, I. Mason, S. Smith, and C. Talcott, *Towards a Theory of Actor Computation*, Third International Conference on Concurrency Theory (CONCUR '92), Springer-Verlag, August 1992, LNCS, pp. 565–579.

[4] G. Agha, P. Wegner, and A. Yonezawa (eds.), *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, Cambridge, Massachussets, 1993.

[5] C. Hewitt, *Viewing Control Structures as Patterns of Passing Messages*, Journal of Artificial Intelligence Vol. 8, 1977, pp. 323–364.

[6] W. Clinger, *Foundation of Actor Semantics*, MIT technical report, AI-TR-633.

[7] A. Yonezawa, J. P. Briot and E. Shibayana, *Object-Oriented Concurrent Programming in ABCL/1*, September 1986, OOPSLA, pp. 258–268.

[8] M. Aksit, J. Bosch, and W. Sterren *Real-Time Specification Inheritance Anomalies and Real-Time Filters* Springer-Verlag, July 1994, LNCS, pp. 386–407.

[9] M. Aksit, K. Wakita, J. bosch, L. Bergmans, and A. Yonezawa, *Abstracting Object Interactions Using Composition Filters*, Springer-Verlag, July 1993, LNCS, pp. 152–1184.

[10] S. Matsuoka, and A. Yonezawa, *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in Research Directions in Object-Oriented Programming, MIT press, 1993

[11] B. Dasarathy, *Timing Constraints of Real-Time Systems: constructs for expressing them, methods for validating them*, IEEE Transactions of Software Engineering, January, 1985, pp. 80–86.

[12] Y. Ishika, H. Tokuda, and C.W. Mercer, *Object-Oriented Real-Time Language Design: Constructs for Timing Constraints*, ECOOP/OOPSLA proceedings, 1990, pp. 289-298.

[13] Y. Ishikawa, H, Tokuda, and C. W. Mercer, *An Object-Oriented Real-Time Programming Language*, IEEE Computer, October 1992, pp. 66–73.

[14] L. Y. Liu, and R. K. Shyamasundar, *RT-CDL: A real-time design language and its semantics*, Information Processing, 1989, pp. 21–26.

[15] N. Wirth, *Towards a Discipline of Real-Time Programming*, Communication of ACM, 1977, vol. 20 pp. 577–583.

[16] Denis Caromel, *Toward a Method of Object-Oriented Concurrent Programming*, Communications of the ACM **36** (1993), no. 9, 90–102.

[17] S. Frølund, and G. Agha, *A Language Framework for Multi-Object Coordination*, Springer Verlag, July 1993, LNCS 627.

[18] S. Frølund, *Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages*, ECOOP'92 European conference on Object-Oriented Programming (O. Lehrmann Madsen, ed.), Springer-Verlag, June 1992, LNCS 615, pp. 185–196.

[19] T. Baker, W. Halang, S. Natarajan, and O. Pazy, *Languages: ADA ? Object-Oriented ?* IFAC Real Time Programming, Georgis, USA, 1991.

[20] D. Kafura, and K. H. Lee, *ACT++: Building a Concurrent C++ with Actors*, JOOP, May/June 1990, pp. 25–37.

[21] B. Selic, G.Cullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling* WILEY, 1994

[22] M. Joseph, *Problems, promises and performance: some questions for real-time system specification*, Springer-Verlag, June 1991, LNCS 600, pp. 315–324.

[23] N. Wirth, *Toward a Discipline of Real-Time Programming*, Communication of ACM, Vol. 20, No. 8, Auguest 1977.

[24] K. Lin, J. W. S. Liu, *FLEX: A Language for Real-Time Systems Programming*, Technical Report No. 1634, UIUC

[25] F. Jahanian and A. K. Mok, *Modechart: A Specification Language for Real-Time System*, IEEE transactions on Software Engineering, 1988

[26] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze, *A Unified High-Level Petri Net Formalizm for Timed-Critical Systems*, IEEE Transactions on Software Engineering, Vol 17, No. 2 February 1991

[27] N. A. Lynch and M. R. Tuttle,*An Introduction to input/output atuomata*, CWI Quarterly, 2(3):219–246, September, 1989

[28] R. Alur and T. A. Henzigner, *A really temporal logic*, Proc. 30th Annual Symp. Foundations of Computer Science, IEEE Computer Science Press, PP. 164-169, 1989

[29] Z. Manna, A. Puneli, *The Temporal Logic of Reactive and Concurrent Systems — Specification*, Springer-Verlag, 1992