©Copyright by Shangping Ren, 1997

AN ACTOR-BASED FRAMEWORK FOR REAL-TIME COORDINATION

BY

SHANGPING REN

B.Engr., Hefei Polytechnic University, 1985
M.Engr., Hefei Polytechnic University, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

Abstract

Programming language research should be driven by the needs of specific application domains, such as real-time embedded systems, multimedia, distributed data-base management applications, etc. For instance, most often large real-time applications involve distributed and concurrently but asynchronously operating devices. The correctness of the integrated systems depends not only on the correct operation of each individual device, but also on the correct coordination among these devices. Furthermore, the correctness of device operation and cooperation requires that both the underlying logical computation be correct and the computation satisfy some notion of quantitative timing requirements.

This thesis is a case study for how a language should be designed to suit a specific application domain. Specifically, I have designed RTsynchronizers (Real Time Synchronizers), a high-level programming language abstraction for specifying real-time coordination constraints between objects in a distributed open system. RTsynchronizers enforce timing constraints on a group of computational objects. They are defined in terms of the interface of members in the group; thus they are independent of the representations of computational objects. During program execution, RTsynchronizers affect the scheduling of distributed objects to enforce real-time relations between events. With RTsynchronizers, we achieve the ability to separately specify and program functional components and the coordination constraints among these components, and the ability to reuse the computational code as well as coordination constraint code.

This research uses both theoretical and experimental methods. Specifically, an operational semantics based on RTsynchronizers and regular computational objects (modeled as actors) is defined. RTsynchronizers are implemented as C++ objects which are used by system schedulers in deciding the appropriate messages to be invoked at message destination computational objects. The implementation demonstrates the feasibility of making the existence of RTsynchronizers transparent to computational objects. Furthermore, an algorithm for detecting infeasible real-time coordination constraints at source code level is developed.

To my parents for their love and their endless support in all my endeavors.

Acknowledgements

I wish to thank my parents, Dingmou and Lisan. Without their continuous love and their endless spiritual support and encouragement, this thesis would not have been possible. I would also like to thank my elder brother whose life experiences have shown me how to make a bright future with strong will and persistent effort. My two American host families Eva and Charlie Higgins, Elisabeth and Zoltan Ujhelyi, your continuous love, support and care helped me overcome the cultural barrier to reach today. No words can express my gratitude to you all. I would also like to thank my friend Professor Gary Riskowski whose friendship made my life here at the University of Illinois much more enjoyable. Last, but not least, I would like to thank my friend and husband Jie. He endured and shared a lot of my frustration over these years, but his understanding, his support and his love are always there for me.

My doctoral committee — Professor Gul Agha, Professor Jane Liu, and Professor Geneva Belford — have been invaluable through their advice and comments. I would especially like to thank my advisor, Gul Agha, for his mentorship and support, for always being there whenever I needed suggestions, guidance and support in research and in life. I would also like to thank Dr. Carolyn Talcott at Stanford University for her feed-back, comments and advice.

I have benefited greatly from discussions with members of the Open Systems Lab. In particular I would like to thank Svend Frølund, Rajendra Panwar, Daniel Sturman, Mark Astley, Nalini Venkatasubramanian, James Waldby, Brian Nielsen, Miyabi Saito, WooYoung Kim, and other group members for their input and contributions.

The research was 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 and NSF CCR 95-23253).

Table of Contents

Chapter

1 Introduction 1
- 1.1 Separating Timing Constraints from Functional Requirements 2
- 1.2 Actors 3
- 1.3 A New Abstraction for Timing Constraints 4
- 1.4 Overview 4

2 Actors and RTsynchronizers 5
- 2.1 Actor Model 5
- 2.2 Global Time and Timing Constraints in Real-Time Applications 9
  - 2.2.1 Global Time $\mathcal{T}$ 9
  - 2.2.2 Timing Constraints in Real-Time Applications 10
- 2.3 RTsynchronizers 11
  - 2.3.1 Restrictions on RTsynchronizers 12
  - 2.3.2 Design Issues 14
- 2.4 Language Abstraction for RTsynchronizers 15
  - 2.4.1 Specifying Constraints 16
  - 2.4.2 Satisfying Constraints 17
  - 2.4.3 Updating RTsynchronizer States 18
- 2.5 Real-Time Programming Languages 19

3 Language Description — Syntax and Semantics 24
- 3.1 Language Syntax and Examples 25
  - 3.1.1 Examples 25
  - 3.1.2 Syntax 28
- 3.2 Semantics of ART System 33
  - 3.2.1 Basic Actor Semantics 33
  - 3.2.2 ART Systems and Its Transition Rules 35
- 3.3 Conclusion 41

4 Language Implementation 43
- 4.1 Pre-Processing — Generating Time Constraint Graphs and Detecting Infeasible Time Constraints 44
  - 4.1.1 Mapping Constraints into Linear Inequalities 45
  - 4.1.2 Transforming MCNF of Constraint Assertions into MDNF 46
  - 4.1.3 Generating Time Constraint Graphs and the Bellman-Ford Algorithm . . 46

4.1.4 Significance of Conducting Infeasibility Detection 48
4.1.5 Complexity of the Infeasibility Detecting Algorithm 49
4.1.6 Related Work 50
4.2 Implementation Issues and Resolving Strategies 50
4.2.1 RTsynchronizer Data Structures 51
4.2.2 Synchronized Clocks 52
4.2.3 Distribution of RTsynchronizer State Information 53
4.2.4 Terminating the Operation of an RTsynchronizer 56
4.2.5 Scheduling Strategy 56
4.3 Conclusion 58

5 Conclusion 59

Bibliography 61

Vita 68

List of Figures

1.1 Decomposing Real-Time Systems in Two Domains . . . 3

2.1 An Actor System . . . 6
2.2 Three Primitive Operations . . . 7
2.3 Events, Orderings and Pending Events . . . 8
2.4 Two Possible (Nondeterministic) Transitions . . . 8
2.5 Producer/Consumer with Time-Bounded Buffer . . . 12
2.6 Relationship between Actors and RTsynchronizers . . . 13
2.7 Events, Orderings and Pending Events . . . 13
2.8 Constraint Subsumes Nondeterministic Transitions . . . 14
2.9 RTsynchronizer Template Example . . . 15
2.10 A Heating System . . . 18
2.11 Composition Filter . . . 22

3.1 Synchronized Robot Hands . . . 25
3.2 Periodic Events . . . 26
3.3 Time Bounded Buffer . . . 26
3.4 Dyeing Machine . . . 27
3.5 Starting and Terminating RTsynchronizer . . . 28
3.6 Safety 1 RTsynchronizer . . . 29
3.7 Periodic RTsynchronizer . . . 29
3.8 React RTsynchronizer . . . 29
3.9 Safety 2 RTsynchronizer . . . 30
3.10 Syntax for Values and Expressions in `RTCL` . . . 31
3.11 Syntax for RTsynchronizers, Constraints and Patterns . . . 31
3.12 The RTCL Program for the Time-Bounded Buffer Example . . . 32
3.13 Three Possible Actor States . . . 33
3.14 Transition Rules in Basic Actor Systems . . . 34
3.15 Two Possible RTsynchronizer States . . . 36
3.16 ART Systems Transition Rules — Instantaneous Transitions . . . 40
3.17 ART Systems Transition Rule — Failure Transitions . . . 41
3.18 ART Systems Transition Rules — Quiescent Transitions . . . 41

4.1 Real-Time Application Design Flow Graph under Separation Paradigm . . . 43
4.2 Time Constraint Graph . . . 47
4.3 Time Bounded Buffer . . . 48
4.4 Infeasible Timing Constraints . . . 48

4.5 RTsynchronizer's Structure 52
4.6 Centralized & Decentralized RTsynchronizers 53
4.7 Hybridizing Centralized and Decentralized Approaches 54
4.8 The Detailed Steps of Delivering a Message at a Time-Bounded Buffer 55
4.9 Adaptive Scheduling 56
4.10 Locating the Next Message for Execution 57

Chapter 1

Introduction

The computing industry is fast moving into real life activities: from applications as simple as home heating control systems, to ones as complicated as full-time patient monitoring systems, nuclear control systems, air traffic control systems, and space control systems. Many real world applications require distributed processing and operate under strict time constraints. For example, with patient monitoring systems, the state of the patient is typically gathered by sensors and input into computer systems; the computer systems 'compute' the necessary intervention, and then actuators carry out such treatment on the patient. Needless to say the sensor information must get to the computer *on time* and the actuator must perform the treatment with respect to the time constraint requirements. Thus, the correctness of such systems not only depends on the correctness of the underlying computation (in the monitoring example, the correct treatment), but also depends on meeting time constraints.

There are three major motivations for applying distributed computing in real world applications, specially in real-time applications. First, large embedded control systems are *distributed*, *concurrent* and *asynchronous*. For instance, in air traffic control systems, control towers at ground and control panels on planes operate concurrently and these two types of components are distributed physically. Second, distributed computing provides information and resource sharing among distributed users. Third, distributed computing provides higher reliability and availability [83]. A reliable system prevents loss of information even in the event of component failures. The multiplicity of storage devices and processors in a distributed computing system allows the maintenance of multiple copies of critical information within the system and the execution of important computations redundantly to protect them against catastrophic failures, where in centralized systems, system availability depends solely on the single central unit. In addition, distributed computing systems are capable of incremental growth. That is, it is possible to gradually extend the power and functionality of a distributed computing system by simply adding additional resources to the system when the need arises. Distributed computing also provides scalability — performance can be scaled up in the presence of increased resources, or can be maintained in the presence of increased workload by adding more resources [81].

In distributed computing, individual components (objects) must exchange information and coordinate with each other to achieve integrated system goals. More specifically, many system have real-time requirements; in such systems, individual objects must not only cooperate with each other, but they must also cooperate with respect to quantitative timing constraints. As pointed out by Hoogeboom and Halang [42], complex coordination among individual asynchronous components exists in real-time systems and the coordination problem in its full gen-

erality cannot be solved by scheduling theory [60]. Hence real-time applications introduce issues that are non-existent in centralized systems. It is essential that distributed real-time programming languages allow programmers to handle resulting issues such as the quantitative timing constraints on object synchronization. However, most embedded systems are currently designed using low level programming languages, such as Assembly and C, in which the timing constraints are intermixed with the functional requirements; then compiler and scheduling technologies are used to find architecture dependent execution. Using low level programming languages and methods which intermix the functional requirements and timing requirements not only compromises object oriented design methodology, but also limits the re-usability of functional as well as the timing constraint specification code. This thesis provides a new programming paradigm that enables system developers to program real-time applications so that timing requirements are specified separately from high-level object-oriented implementation of functional requirements. The separation of functional and timing requirements facilitates the reusability of both functional code and timing requirement code. Furthermore, it enables methods for preliminary testing of design at the source code, architecture-independent level to check if the timing requirements are inconsistent.

This chapter gives an abstract description of our new language paradigm. We use the Actor model to model distributed systems, and extend it by providing a new language abstraction for modular and functional-independent specification of timing requirements for real-time systems. Finally, we outline the following chapters of the thesis.

1.1 Separating Timing Constraints from Functional Requirements

Let's consider an extension to the producer/consumer problem, namely a *time-bounded* producer/consumer problem. Assume that there is one `Producer` and one `Consumer`. The producer and the consumer have a common shared buffer. The `Producer puts` products into the buffer, while the `Consumer gets` products from the buffer. The common buffer is bounded. In addition, each product must be kept inside the buffer for a fixed interval of time `[d1, d2]` before the consumer is allowed to consume the product. This simple example illustrates two concerns, typical of any real-time systems, namely, the correctness of underlying computation for both the producer and the consumer, and the correctness of their temporal behavior. It is desirable that this extension should not involve internal modification of either the producer or the consumer's representation; in other words, the original producer/consumer program should be reused in the extended real-time environment. However, if the code intermixes the two concerns, it would be difficult to reuse either the functional code, or the timing constraint specification.

Our approach separates two orthogonal concerns — functional correctness and temporal correctness. Timing requirements are group-wise and are independent of the computational objects' behaviors. Figure 1.1 illustrates the separation paradigm. By keeping timing features separated from computational features, not only do we increase the program's modularity and reusability, but we also provide a foundation for level inconsistent timing constraints detection at source code level, which will be covered in detail in Section 4.1.

In the computational domain, complex real-time systems can be further decomposed into distributed concurrent components. We may model such components as computational objects. Similarly, in the time relation domain, a complex timing relationship among different compu-

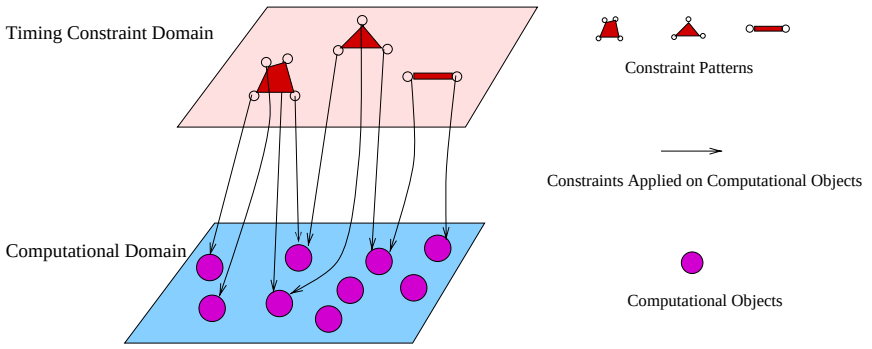


Figure 1.1: Decomposing Real-Time Systems in Two Domains

tational objects can be decomposed into smaller non-computational entities which abstract simpler timing requirements. However, these non-computational entities should not interfere with the behaviors of underlying computational objects, but only constraint when certain computational behavior should occur.

1.2 Actors

We use the Actor model as our underlying computational framework for distributed real-time computing. In other words, individual computational components in a distributed real-time system are modeled as actors.

An actor is modeled as a collection of linearly ordered events and is intuitively interpreted as a point in an abstract computational space. A unique actor name called an `aid` is associated with the space. Events at different actors may be related by a causal order. This partial causal order is interpreted as asynchronous messages passing between different spaces.

The Actor model, developed by Hewitt [41] and refined by Agha [2], provides a uniform abstract representation of distributed systems. Actors are encapsulated concurrent objects that interact with each other by sending buffered, asynchronous messages [2, 6, 7, 8]. 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. The Actor model not only provides a uniform view of distributed computation units, but also provides scalability by allowing asynchronous message passing as the only means of inter-object communication means. In addition, the asynchronous communication model does not infringe upon the concurrency inherent in applications.

Each actor has its own thread of control and a unique name which may be used for send messages to it. An actor's name may be included in messages sent to other actors, thus supporting dynamic reconfigurability. An Actor also has a local states, which may only be manipulated by the actor itself upon processing incoming messages.

Actors are reactive entities — they only react in response to messages. However, an actor is a single threaded unit: it processes messages one at a time. Furthermore, each invocation of a message at the destination actor is atomic and uninterruptable.

Messages are the only means by which autonomous actors communicate with each other. Message passing is asynchronous in that the message sender does not wait for an acknowledgment from the message receiver before it proceeds with the rest of the computation. Messages may not necessarily be received in the order in which they are sent. However, they are ensured to be delivered at the destination actor sooner or later. More precisely, weak fairness is guaranteed in the Actor model.

1.3 A New Abstraction for Timing Constraints

As has been briefly explained before, real-time systems may be decomposed into two separate domains: a computational and a timing constraint. The complex timing requirements in real-time systems may be further decomposed into non-computational entities we call RTsynchronizers, which abstract simpler timing requirements.

RTsynchronizers are different from computational objects in that they do not directly interact with the underlying computational objects. Their functionality is to constrain *when* computational objects should react and *when* certain computational behavior should occur. On the other hand, by observing certain behavior happening at the computational objects, RTsynchronizers may change their own state and thus the restrictions on scheduling of messages at a group of computational objects.

1.4 Overview

Here is how the rest of this thesis is organized. Chapter 2 formally defines the Actor model and discusses the design of RTsynchronizers.

Chapter 3 provides a formal definition of RTsynchronizer and the semantics of Actor systems with real-time extensions. The state of a system is represented by a mathematical structure called a *configuration* and computation is represented by a transition between relations on configurations. System failure with respect to timing requirements is also defined in this chapter.

Chapter 4 is dedicated to developing an algorithm for source code level inconsistent (or infeasible) timing constraint specifications and to discussing issues and strategies for the distributed implementation of RTsynchronizers. It includes the data structure of RTsynchronizers, the distribution of RTsynchronizer state information and scheduling strategies.

In Chapter 5, we draw our conclusions on this research, the findings, the unsolved problems and future work.

Chapter 2

Actors and RTsynchronizers

We argued that distributed systems should be developed by composing concurrent asynchronous systems, represented as computational objects called *actors*. Computational object's encapsulate computational behavior and address questions of *what* they should do and *how* they should do, but not *when*. However, in order to achieve real-time application goals, these autonomous and asynchronous computational objects must synchronize and cooperate with each other to adhere to constraints about *when*. Such real-time coordination constraints among a group of objects cross an individual objects' encapsulated state and process boundary; rather they affect a group of objects. Thus we require an external environment that decides *when* each individual computational object should react. Clearly, coordination behavior is not only different from, but orthogonal to an object's traditional computational behavior. Therefore, it should be separately specified by a new abstraction. By specifying the coordination separately, we can promote modularity and reusability.

Separating real-time synchronization behavior from an individual object's computational behavior, not only allows software designers to orthogonally decompose complex systems and concentrate on one type of concern at a time, but also improves program modularity, reusability, and analyzability; and thereby increases complex systems' dependability.

We use an Actor language for specifying objects' computational behavior, but design our own language construct, RTsynchronizer, to express the real-time coordination behavior among a group of objects. Section 2.1 describes the actor model. Section 2.2 discusses our time model and timing constraints in real-time applications. RTsynchronizer is discussed in detail in Sections 2.3 and 2.4. Finally, in Section 2.5 we discuss some related work, summarize our findings and draw conclusions.

2.1 Actor Model

Actor model is an abstract, language independent model of distributed computing based on partially ordered events. For our purposes, an event is discrete and instantaneous. An *actor* is a conceptual point in space on which events are 'localized'; corresponding to each actor is a set of linearly ordered events. We call this linear order the *arrival order* of events on that actor. The arrival order has an origin called the *creation event*. Intuitively, that is the point at which the actor is created. An event causes two types of effects: an internal change in the actor's behavior (which may affect how it responds to future events) and events at other actors. The

latter case represents events that change the behavior of other existing actors and/or creation events (corresponding to the creation of new actors).

The Actor model, first proposed by Hewitt [41] and later studied and developed by Agha [2, 8], provides a flexible and uniform abstract representation of distributed computation in real-world systems. A wide range of entities, such as memory chips, subprograms, sensors, actuators, and entire computers, may be modeled as actors.

Actors extend the concept of objects to concurrent computation. Each actor potentially executes in parallel with other actors. Only the actor's own methods may operate on or change its local state. Actors communicate with each other by sending asynchronous messages. Graphically, an actor system may be illustrated as in Figure 2.1.

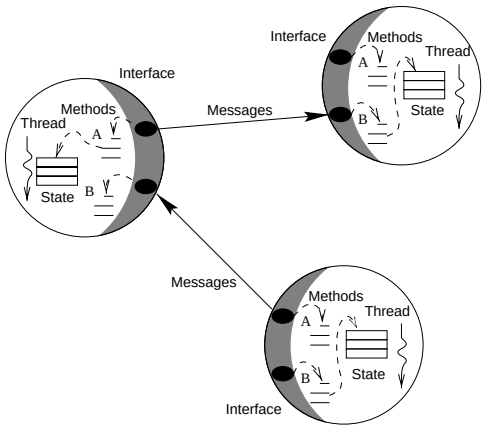


Figure 2.1: An Actor System: actors encapsulate threads and states; the public methods that operate on the states comprise their interfaces.

Each actor has a unique identification name (*aid*, actor id) which other actors can use to send it messages. Actors are reactive entities in that they start executing their methods only in response to receive messages. Message passing in the Actor model is asynchronous and point-to-point. The implication of this is as follows: (1) an actor can send messages to only actors it knows the *aid* of; however, an actor's *aid* may be communicated in messages, hence allowing dynamic reconfiguration of interconnection topology; (2) sending a message is a non-blocking operation: the sender does not wait for a reply from the receiver, instead immediately resumes its computation; (3) message processing at each actor is atomic and non-preemptive: once an actor begins executing a method, it continues non-preemptively until its completion; (4) messages sent but not yet processed by the receiver are conceptually buffered in the receiver's mailbox until they are picked up for processing by the receiver; (5) the system possesses a weak fairness property, i.e., messages sent out are guaranteed to eventually reach their destination and be processed (but may be subject to arbitrary communication delays); message delivery order is nevertheless not guaranteed: messages might be received in some order that is different from the order in which they are sent; (6) message deliveries are the only observable events in actor systems (as defined by Clinger [24]).
Each actor has the following three primitive operations (as showing in Figure 2.2):

- **send** (a, v) sends a message to address a with contents v.

- **create** $b(v)$ creates a new actor with behavior b and initial parameter v; **create** returns a unique *aid* of the newly created actor.
- **ready**(s) indicates that the actor has finished processing its current messages and is now ready to process another one with new behavior s.

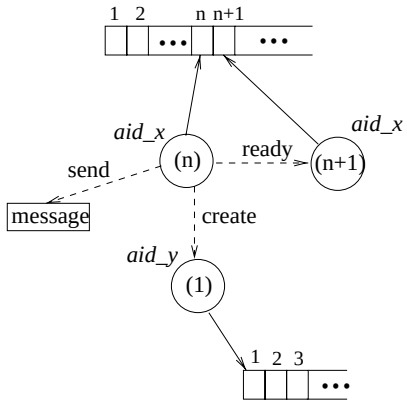


Figure 2.2: Three Primitive Operations: An actor has a unique *aid*, and may perform 3 primitive operations upon receiving a message. Message processing is non-preemptive. $(n+1)$ indicates the actor behavior when processing the $n+1$th message

We will not be concerned with an actor's internal behavior. Of particular interest to us is the fact that events on different actors may be related by a partial order corresponding to a *causality relation*. It is important to observe that actors are asynchronous, thus there can be an arbitrary delay between causally connected events. A *pending event* is an event whose antecedent event has occurred but the consequent has not. In a completed computation, there are no pending events.

An actor computation unfolds as some pending events are nondeterministically realized. As a concrete representation, we identify events with the acceptance of a message at an actor. Thus, all interaction in this concrete representation is through sending and receiving messages. Other types of events are possible but do not affect the formalism as long as causality and arrival orders are obeyed.

Figure 2.3 illustrates an event diagram which provides a pictorial representation of an actor computation. Figure 2.4 shows two possible unfoldings of the same computation.

The transitive closure of the causality relation and all the arrival orders is called the *combined order*. It is possible to map events to a global time line corresponding to real numbers so that the mapping preserves the combined order. Of course, such a global time is not unique — there could be many possible linear representations which would be consistent with the combined order since the combined order is only a partial order [24]. We discuss global time in more detail in section 2.2.

The advantage of the Actor model is that it not only provides a uniform view of distributed computation units, but also provides scalability as a by-product of only allowing message passing as the inter-object communication means. In addition, the asynchronous communication model does not infringe upon the concurrency inherent in applications.

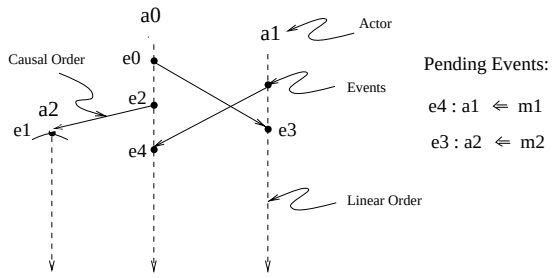


Figure 2.3: Events, Orderings and Pending Events. Message $m1$ to actor $a1$ caused by event $e4$ $(e4 : a1 \Leftarrow m1)$, and message $m2$ to actor $a2$ caused by $e3$ $(e3 : a2 \Leftarrow m2)$ are pending.

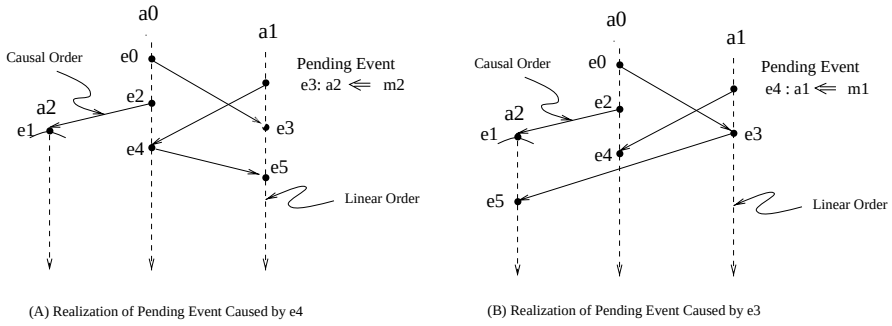


Figure 2.4: Two Possible (Nondeterministic) Transitions

In the Actor model, the function of each actor is to carry out a computation. We will call such actors *computational actors*. As we may have noticed, the notions of time and timing constraints among a group of computational actors are not captured in Actor systems. Hence, it is necessary to extend the base actor model in order to meet the needs of real-time computing. Specifically, we introduce the notion of time into the model.

2.2 Global Time and Timing Constraints in Real-Time Applications

In distributed asynchronous systems, a wide range of temporal properties of a system can be interpreted in terms of a relation "$\prec$" (*precedes* or *happens before*) over the events of the system. Such a relation should be a strict (i.e. irreflexive) partial order. A complement relation to the *precedence* relation is *concurrency*, or simultaneous relation ($\sim$), which is defined as:

$$e \sim e' \Leftrightarrow \neg(e \prec e') \wedge \neg(e' \prec e).$$

However, the question is whether there exists a mapping from the distributed events to a linear global time domain so that it preserves the precedence relation of the domain?

2.2.1 Global Time $\mathcal{T}$

Lamport's [49] causality effects is one way of viewing the 'precedes' relation. That is, $e \prec e'$ means that it is possible for event e to causally affect event e'; two events are concurrent if neither can causally affect the other. However, this is not adequate for describing some temporal properties in hard real-time systems. To illustrate this inadequacy, consider two processes p_1 and p_2 running on different processors. When the processes finish, they will send out *finish* signals, s_1 and s_2, respectively. These two processes are not causally related. But we wish to know when each signal arrives (and thus the order between their arrivals). This is a very common practical requirement in real-time systems. However, according to the 'causality' view, we can only conclude that s_1 and s_2 finish in some indeterminate order, irrespective of the speeds of the processors.

Another way to obtain precedence relations over events is by using a global time domain to mark the points of events in terms of *instants* in the domain. The sufficient condition for $e \prec e'$ is that e happens before e'. However, e *happens before* e' is not a necessary condition since it also depends on the granularity of the global time domain in which observation has taken place. Nevertheless, explicit time using an appropriately chosen domain may be useful (as in the previous example, it may be possible to time s_1 and s_2 and determine which finishes first).

Since the precedence relation on a set of events may not be determinate in some time domains, is there a *universal* or global time domain such that any precedence relation on events can be given an *equivalent* characterization in terms of that domain?

Clinger [24] has proven that the global time is realizable:

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 the 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 $t > t'$,

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

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, $|C_a(t) \Leftrightarrow C_b(t)| \leq \epsilon$. Second, method execution on each actor is atomic and non-preemptive. Finally, method execution time is insignificant compared to communication and scheduling overheads.

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

Definition 2.1 (Precedence Relation ($\prec$))
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 $e_1 \prec e_2$ iff $t_2 = t_1 + d$, where t_1, t_2 and $d \in \mathcal{T} \subset \Re_{\geq 0}$, $\Re_{\geq 0}$ is non-negative real numbers and $\mathcal{T}$ has not cluster points.

The definition of $\mathcal{T}$ excludes the cluster points. In distributed systems, the global quantitative precedence relation may be approximated by using a common clock, provided the granularity of observation is sufficiently coarse.

2.2.2 Timing Constraints in Real-Time Applications

For real-time applications, not only causal orders must be obeyed, but also some additional quantitative timing requirements between events must be met. Two types of such requirements for the timeliness of system behavior that are specific to real-time systems are *relative time* and *absolute time* constraints. As defined in [42], a relative time constraint requires that an event or action must occur within a given interval of time relative to the occurrence of another event; where an absolute time constraint specifies the system behavior for given points in time, with accuracy relative to a chosen granularity.

We extend the actor model to fix a unique global time called `real time`, and define a *timing constraint* (or just constraint) between events. The relative and absolute time constraints are expressed using one of the following two notations:

- `A : B [t1, t2]`
- `b-expr : A [t1, t2]`

where `0` $\leq$ `t1` $\leq$ `t2`, and `t1, t2` $\in \mathcal{T}$.

The meaning of constraint `A : B [t1, t2]` is that once event `A` happens, event `B` is constrained to happen within the time interval `[t1, t2]` relative to the time when event `A` happens. On the other hand, `b-expr : A [t1, t2]` specifies that event `A` must happen in the time interval `[t1, t2]`, if the logical expression `b-expr` is `true`. The logical expression encodes the

history of the computation (in terms of events that have already occurred, i.e. the history of the computation).

In order to capture the history of computation in an actor system that is of interest to the real-time coordination, we encode the relevant history in the state of an object called *RTsynchronizer*. An RTsynchronizer also contains a set of constraints that are expressed in terms of its state. The constraints constrain the behavior of a group of actors. Where different RTsynchronizers enforce their constraints conjunctively, different constraints may be conjunctively or disjunctively connected within an RTsynchronizer. Note that the actors affected by different RTsynchronizers may overlap. The next section discusses RTsynchronizers in detail.

2.3 RTsynchronizers

We first use a simple example to give an intuitive feeling about the underlying function of RTsynchronizers in real-time applications.

Imagine a pizza production line. A pizza maker puts an unbaked pizza into the baking line, which has a limited size; and a server takes it out within a certain amount of time: not too early nor too late!

The pizza maker and the pizza server can be modeled as two autonomous actors. Their functionalities are *to make a pizza and put it in the baking oven* vs. *to take the pizza out and serve it to customers*, respectively. How long the baking line is and how long the pizza should be baked are not related to how the pizza is made, nor how the pizza is taken out and served. Another characteristic of the problem is that the two actors work independently, but at the same time, because of the baking line size and baking time constraints, the maker and server must cooperate with each other to achieve a profitable pizza business. The cooperation is another aspect of the integrated system, which must be captured by a different abstraction. The role of RTsynchronizers is to specify such cooperation and ensure the specified constraints are met by the independent computational actors.

This real life example can be generalized as a *time-bounded producer/consumer problem* — an extension to the traditional producer/consumer problem with a time bounded buffer: there is one `producer` and one `consumer`. The producer and the consumer have a common shared buffer. The `producer puts` products into the buffer, while the `consumer gets` products from the buffer. The common buffer is bounded. In addition, each product must be kept inside the buffer for a fixed amount of time `[d1, d2]` before the consumer is allowed to consume the product. The producer and consumer work independently. However, as an integrated system, the producer and the consumer must cooperate with each other in a timely fashion.

The role of the RTsynchronizer is of a controller or a coordinator that expresses the cooperative aspects of individual objects and ensures that timing constraints are met. In the example, the `Producer`'s `put` and `Consumer`'s `get` behaviors need to be both qualitatively and quantitatively synchronized: qualitatively, if the buffer is full, `put` is not allowed and similarly, if the buffer is empty, `get` is not allowed; quantitatively, if put takes place at time `t`, `get` should take place in the time interval `[t + d1, t + d2]`. Constraints are in general orthogonal and external to objects' functionalities. They are based on the action names, or action *patterns*, but do not interfere with the execution of actions. The constraints exist in a context furnished by RTsynchronizers. The buffer size, the time duration `t`, etc., comprise the context for the constraints in the time-bounded buffer example. Figure 2.5 illustrates the specification of the producer and the consumer, and the role of RTsynchronizers.

```
Producer(){
  methods:
    put();
    other();
}

Consumer(){
  methods:
    get();
    other();
 }
```

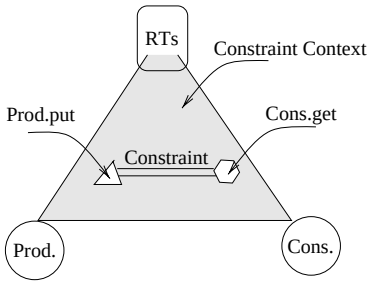


Figure 2.5: Producer/Consumer with Time-Bounded Buffer

Similar to traditional objects that provide a construct for specifying computations and for carrying out the specified computation, the functionality of RTsynchronizers is to provide such a structure for expressing coordination constraints based on the interface of objects, but independent of how an object's computation proceeds, and to ensure that such constraints among a group of individual objects are met. More generally, an RTsynchronizer provides an abstraction for programming an environment in which computational objects operate. Within such an environment, computational objects must obey certain rules enforced by their environment. For real-time applications, computational objects carry their computation in such an environment where timeliness is the key requirement.

RTsynchronizers may be modeled as actors. However, as we have discussed, such actors possess different properties and serve different purposes.

Conceptually, computational actors are independent of RTsynchronizers. However, their behaviors in the time domain may be influenced by the existence of coordinating RTsynchronizers. On the other hand, the encapsulated computational actors' certain behaviors, such as receiving messages, are observable by the coordinating RTsynchronizers. RTsynchronizers may accordingly change their own states upon observing certain behaviors of their coordinated computational actors.

Structurally, the two types of entities, computational actors and RTsynchronizers, are encapsulated and independent of each other. However, their "shadows" could affect each other. Figure 2.6 illustrates the abstract view of the relations between RTsynchronizers and their coordinated computational actors.

2.3.1 Restrictions on RTsynchronizers

We have separated the specification of real-time systems in such a way that we first only need to be concerned with the underlying functional computation (i.e. a computation with timing requirements lifted), and implementing computational objects to carry out the computation. We next embed such functionally correct objects in a environment constrained by RTsynchronizers, assuming that RTsynchronizers provide a valid constraint environment, to achieve the timeliness. However, a precondition for this approach to be correct is that the RTsynchronizers do not interfere with the well-behaved underlying computation.

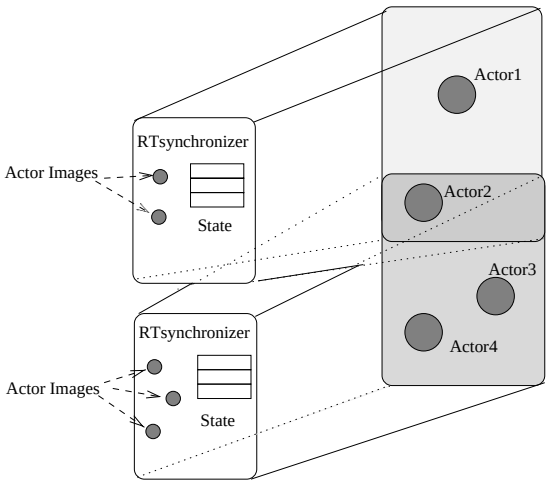


Figure 2.6: Relation between Actors and RTsynchronizers: groups of actors are coordinated by RTsynchronizers; each RTsynchronizer has its coordinating actors' images, representing the actors' observable behavior; different RTsynchronizers may overlap over their coordinating actors.

Since computational actors that carry out the logical computation are reactive objects, they act only upon receiving messages. The implication of this property is that if RTsynchronizers do not send out messages, the underlying computation will retain its functional properties in the presence of RTsynchronizers; only the scheduling of when computational objects should act is affected.

To be more specific, let us consider the event diagram given in Figure 2.3 with its real-time extension, i.e. a global time is added into the diagram as shown in Figure 2.7.

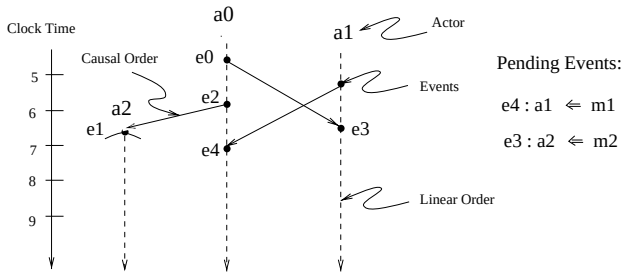


Figure 2.7: Events, Orderings and Pending Events. Message $m1$ to actor $a1$ caused by event $e4$ $(e4 : a1 \Leftarrow m1)$, and message $m2$ to actor $a2$ caused by $e3$ $(e3 : a2 \Leftarrow m2)$ are pending.

Without the presence of RTsynchronizer, there are two possible (non-deterministic) transitions indicating whether message $m1$ or message $m2$ is delivered first. However, if there exists an RTsynchronizer which only has a constraint, say, `true : m1 [10, 20]` and the current global time is 5, the existence of the RTsynchronizer subsumes the non-deterministic choice between

the two possible transitions (as repeated in Figure 2.8) and only allows the message $m2$ to be delivered at actor $a2$; i.e. only the transition shown in (A) becomes possible.

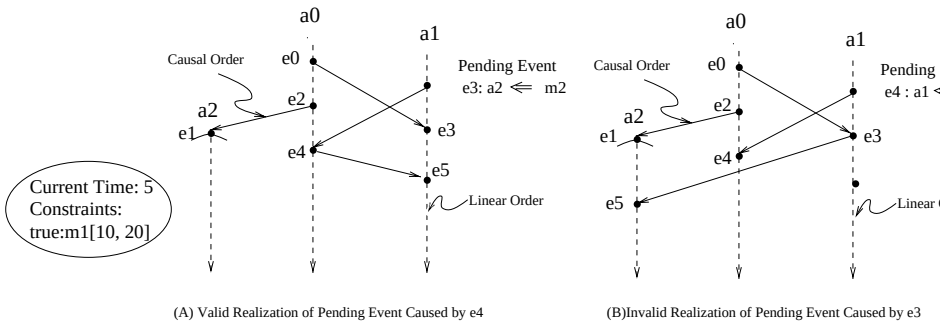


Figure 2.8: Constraint Subsumes Nondeterministic Transitions

In accordance with this observation, we require that RTsynchronizers, as special actors, not be allowed to send messages. Moreover, since the existence of RTsynchronizers is transparent to computational actors, no messages should be addressed to RTsynchronizers. Hence, RTsynchronizers do not receive messages either. However, they may change their states by observing certain messages being delivered at computational actors in their control. Such restrictions on RTsynchronizers assure that previously described precondition, i.e. RTsynchronizers should not interfere with underlying computations, is satisfied.

2.3.2 Design Issues

The design concerns come from three different directions as illustrated in Figure 2.6: from the computational object's point of view, the RTsynchronizer's point of view, and the integrated system's point of view. From the computational object's point of view, the existence of RTsynchronizers should be *transparent*. In other words, RTsynchronizers should be able to enforce constraints without communicating with the constrained computational objects. Such transparency not only enables reuse of distributed computational objects, but also allows separation of RTsynchronizers from constrained computational objects. Separating constraints from computational objects provides more clean and intuitive abstraction for reasoning about such constraints.

From the RTsynchronizer point of view, RTsynchronizers should rely only on computational objects' interfaces. The internal representation of objects should be opaque to RTsynchronizers. Using object interfaces as the connection boundary between computational objects and RTsynchronizers, sustains object encapsulation and provides modularity and reusability of both RTsynchronizers and computational objects: changing the internal representation of computational objects will not affect RTsynchronizers; the same RTsynchronizer can apply to different objects that have the same interfaces.

From the integrated system's point of view, the constraints specified by RTsynchronizers should be *declarative* and independent of system architectures, such as processor speeds, node connections, etc. In addition, RTsynchronizers should be allowed to overlap as shown in Figure 2.6 and should be composable. Composability enables decomposing complex constraints into simple and separately specified entities.

In addition, two other requirements for the timeliness of system behavior that are specific to real-time systems are *relative time* and *absolute time* constraints [42]. Hence, RTsynchronizer should provide a mechanism to express such requirements as stated in Section 2.2.

In order to ensure that timing constraints are met, RTsynchronizers need a reliable source of sufficiently accurate time signals. We assume that global absolute time is part of an RTsynchronizer's state. In other words, the global time is a parameter to each "observation" of an RTsynchronizer — an RTsynchronizer can look at the global clock.

2.4 Language Abstraction for RTsynchronizers

From a language point of view, an RTsynchronizer is like an ordinary actor in that it has a mutable state and can be created dynamically. However, unlike ordinary actors, an RTsynchronizer does not send or receive messages. Rather, it provides a declarative constraint specification between certain message deliveries in a group of computational actors. In addition, it changes state by observing some events at the actors it constrains; in contrast, actors change their states upon receiving messages.

An RTsynchronizer is created by instantiating a *template* using constrained actors' unique names and other values for template variables. For example,

new MyRTS (**actor**:aid1, aid2, aid3; **int**: d1, d2)

creates a new RTsynchronizer whose behavior is given by `MyRTS` which constrains certain message deliveries at actors `aid1`, `aid2`, and `aid3`; `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. The template variables, such as `d1`, and `d2` in the above example, are changeable.

```
RTsynchronizer MyRTS(actor actor1, actor2, actor3; int v1, v2) {
  Declare
    /* declare and initialize local variables; */

  Constrain
    constraint_1
    ...;
    constraint_n;

  Update
    message_pattern : state_variable = new_value;
    ...;
    message_pattern : state_variable = new_value;
    boolean_exp     : state_variable = new_value;
}
```

Figure 2.9: RTsynchronizer Template Example

The internal structure of a template for RTsynchronizers has, in general, three sections: `Declare`, `Constrain` and `Update`, which abstract the three main aspects of an RTsynchronizer

that: an RTsynchronizer has state; its functionality is to enforce a set of declaratively specified constraints on its constrained objects; and an RTsynchronizer changes its states upon observing certain events on controlled objects.

RTsynchronizer state variables are declared in the same manner as in ordinary objects: `typeName varName [= iniValue]`, where the initial values are optional in the declaration. In other words, state variables may obtain initial values at a later time. However, uninitialized variables may not be used in expressions.

In the following subsection, we discuss the remaining two components of RTsynchronizers, namely, `Constrain` and `Update`, in detail.

2.4.1 Specifying Constraints

Constraints in RTsynchronizers are declarative assertions about the quantitative precedence relation among observable events at constrained objects. Recall that in the Actor model, an observable event is defined as a message delivery at the message target actor (see Section 2.1). Hence, events are independent of actors' internal computation, and may be identified by the corresponding messages. However, messages are dynamic consumable entities. Therefore, we need a syntactic category to identify such dynamic units. *Message patterns* may be used for such purpose [4, 31, 65].

Definition 2.2 (Message Patterns)
A message pattern is a 3-tuple

```
((sender, seqNo), receiver.cX, b_exp)
```

where ***(sender, seqNo)*** *is message tag;* ***cX*** *represents message contents, including the method name, and other actual parameters;* ***b_exp*** *is a functional boolean expression which may contain enclosing RTsynchronizer state variables, but the evaluation of the expression does not have side effects.*

A message m is indicated by a pattern p if the following 4 conditions hold:

1. m's sender is the actor specified in the pattern-tuple as `sender`;
2. m is the `seqNo`*th* message sent by actor `sender` (`seqNo` is called the message's *sequence number*);
3. m's target is the actor specified in the pattern-quadruple as `receiver`;
4. the values transmitted in m, together with current enclosing RTsynchronizer state, satisfy the boolean expression `b_exp`.

Note that (1) `sender` and `seqNo` are message tags, which we assume are supplied by the run time system. They may be transparent to an application programmer and actors implementing the functionalities do not see this extra system level information; (2) the `b_exp` may reference the state variables of enclosing RTsynchronizer's (in which the pattern is specified) as well as values transmitted by the message; (3) a message indicated by a full pattern [1] is unique.

[1] We can relax the pattern restriction by using * to match *any* of the `sender`, or sequencing number `seqNo`, or both (a * could occur in these two positions). In this case, a pattern may identify more than

In the basic Actor model, messages are guaranteed to be delivered. When a message is delivered, it gets consumed. Thus, for each message there is one and only one corresponding event. Patterns identify messages, and hence identify events because of the 1-to-1 relationship between messages and events. Therefore, the time when an event happens may be denoted by a pattern which identifies that event.

Conceptually, each constraint ties two different types of events into an observation pair: *triggering* event and *consequence* event. These two events need not have logical causal relations. But observationally, the occurrence of a triggering event *causes* the *consequence* event to be constrained in some time span. We therefore express a constraint in the following format:

```
event1 :  event2 timeInterval
```

Such a constraint asserts that upon the occurrence of event `event1`, event `event2` must happen in the time frame `timeInterval`. The `timeInterval` is a relative time duration: the starting point of such interval is relative to the time when `event1` occurred. A `timeInterval [d1, d2]` is a valid time interval if and only if `d1`, `d2` $\in \mathcal{T}$, and `d1` $\leq$ `d2`. Where $\mathcal{T}$ is a non-negative real number domain excluding cluster points.

Another constraint format is:

```
boolean_exp :  event2 timeInterval
```

This constraint asserts that the occurrence of event `event2` is constrained if the `boolean_exp` evaluates to `true`, and the `timeInterval` refers to the absolute global wall-clock time frame.

In the two constraint formats, `events` are represented by message patterns. These two constraint formats capture the specific requirements for the timeliness of real-time system behavior, namely relative time and absolute time [42].

2.4.2 Satisfying Constraints

As we know, a constraint expression consists of three parts: a *triggering* pattern, a *consequence* pattern, and a *time interval*. A message that satisfies any pattern in a constraint expression may have an effect on future message deliveries.

In particular, a message satisfying a triggering pattern is not constrained in time; however, the consequence of delivering such a message is that another message is constrained in a time frame relative to the time when this message is delivered. In addition, the delivery may cause RTsynchronizers to change state as described below in Section 2.4.3.

However, if the consequence part of a constraint expression is satisfied by a message, that message is constrained in a specified time interval relative to when the trigger is fired. The validity of delivering such a message depends on the time when the triggering message of the constraint is delivered (t), the current time provided by the RTsynchronizers' clock (τ) and the specified time interval ($[d_1, d_2]$). The message may be delivered only if τ lies in $[t + d_1, t + d_2]$.

---

one message. However, the target actor name is mandatory. When the sender and message sequence number are not important and the boolean expression is the constant `true`, the pattern specification can be simplified as `receiver.cX`.

2.4.3 Updating RTsynchronizer States

RTsynchronizers encapsulate state. The state may be changed when an RTsynchronizer observes certain events. The `Update` section of an RTsynchronizer specifies when and how the RTsynchronizer mutates its state. More specifically, for the update constraint:

```
ptrn :  state_var = new_value
```

the delivery of a message indicated by `ptrn` causes the `state_var` to assume the new value. However, the state alteration and the delivery of the message of pattern `ptrn` happen atomically in the sense that no intermediate states between the delivery of the message and the state transformation are observable.

Since global absolute time is part of an RTsynchronizer's implicit state, absolute time progression causes implicit state changes, and also may cause explicit state variables to assume new values. For example, with a constraint:

```
cTime() == 5 :  timeUp = true,
```

the implicit RTsynchronizer state time change causes the explicit state variable `timeUp` to assume the value `true`. However, explicit state change caused by the progression of time does not happen until RTsynchronizer evaluates the validity of delivering constrained messages.

To further understand the effects of an RTsynchronizer's implicit state change, namely clock variable change, consider an automatic home heating system in which the heater is turned on only after 6 O'clock in the morning. We use an actor `Heater` to model the heating system. When `Heater` is created, it is sent with an initial message `start`. One way to program the time setting is as follows:

```
RTsynchronizer RTheater(actor Heater; real t) {
  Declare
    Boolean canStart? = 0;
  Constrain
    ((*,*), Heater.start(), canStart? == 0) (inf, inf);
  Update
    currentTime() >= t :  canStart? = 1;
}
```

Figure 2.10: A Heating System

Initially, `canStart?` is set to 0 (`false`) and the constraint disables the `start` message. When the global time (reflected by RTsynchronizer's implicit state `cTime()`) progresses silently to the point `t`, the explicit state variable `canStart?` is changed.

This example demonstrates how the implicit state change may affect explicit state change and further affect system evolution.

In a complex system, there may be more than one RTsynchronizer. As Figure 2.6 illustrates, the control power of RTsynchronizers may conjunctively overlap. More specifically, different RTsynchronizers may have the same constraint patterns. Thus, a message satisfying a pattern may be constrained by multiple RTsynchronizers and its delivery may cause multiple RTsynchronizers to change state. However, all the state changes and the delivery of the causal message

are atomic and indivisible. It is worth noticing that within an RTsynchronizer, generating constraints on the consequent events and updating RTsynchronizers' state variables are sequential, and follow the static scoping rule and specification order, i.e. top-down order.

2.5 Real-Time Programming Languages

For time-independent software applications, many high level general programming languages have been developed to raise the abstraction level and provide implementation independence. A good example is object-oriented programming languages. Object-oriented programming languages provide high level abstractions and separations, which abstract computational autonomous behaviors into objects, and separate data representations from implementations. Because of the concept of object, large software design can be broken down into smaller and more manageable independent parts [46]. In addition, separation between data representation and implementation allows the programmer to concentrate on different concerns at different design stages. Even here coordination problems arise [32].

Real-time programming languages are more complicated than non-real-time programming languages because the quantitative time domain must be taken into consideration — one of the reasons that Assembly languages prevailed in the development of real-time applications longer than in other areas of computing [39]. A natural approach for providing high-level real-time languages is to augment existing programming languages with real-time constructs. Programming languages like Flex [54], and RTC++ [43] are examples of such an approach. On the other hand, Ada [14] was specially designed for real-time applications. However, both Ada and augmented languages intermix real-time features with functional (or computational) aspects of the languages.

Akşit et al. [11, 10] proposed linguistic constructs called *real-time filters*, which lift the timing specification from the body of objects up to their interfaces. Structurally, it is cleaner. However, the potential difficulty of the filter approach is an inability to abstractly specify timing constraints among a group of distributed objects.

We discuss some of these high-level real-time programming languages in a little more detail in the following.

Ada

Ada was designed for the U.S. Department of Defense as a general standardized programming language and a real-time programming language [14, 89]. It includes most features conceivable in modern computer languages. It is well structured, quite modular and its units can be separately compiled.

The predefined library package `CALENDAR` in Ada provides programmers with wall-clock time. The only time-related language construct provided by Ada is *delay* which delays the executions of tasks by specified time periods. The specified time period is relative to the time when the *delay* statement is executed.

In Ada's semantics, atomic execution of statements is not enforced. Thus, execution of `delay (p - t)` can be interrupted after calculating `(p - t)` and before `delay` in a distributed real-time system. Moreover, the `delay` statement only guarantees that the process is not resumed before the delayed time; hence Ada does not provide a way to program precise periodic task execution [85].

Ada provides a rendezvous mechanism to synchronize tasks. Even though rendezvous may be set to be time bounded, the nondeterministic selection of an entry by the *select* statement introduces unpredictability. Moreover, rendezvous requires one task to call a named *entry* of another and does not take place until the called task decides to accept the call; in other words, it requires that each task be willing to wait for another. No asynchronous nonblocking inter-process communication is supported. Therefore, the rendezvous synchronization mechanism is not sufficient to specify arbitrary quantitative time dependencies among distributed asynchronous processes.

A later version of Ada (Ada 95), has enhanced the *delay* construct and provides a new *delay ... until* statement, which allows programmers to specify *absolute delayed* times. However, the semantics of both language constructs is that *delay* or *delay ... until* only guarantees the lower bound of the *delayed* time [1]. In addition, even though timing constraints can be specified based on the module's interface, the language itself does not prevent embedding the timing information inside the computational code.

In general, as a real-time programming language, Ada lacks adequate capabilities for expressing and enforcing temporal relations between and within different execution units in distributed systems.

FLEX

The programming language *Flex* [54, 47] is a derivative of C++. It provides a new language construct, *constraint block*, to describe constraints on time as well as other resources. A constraint block identifies restrictions that must apply while a section of code is in execution. It takes the form:

```
[label:] (constraint; constraint; ...) [⇝ {...}]{
   ...;
   statements;
   ...;
}
```

This block specifies a sequence of statements to be executed, and asserts that all of the constraints will be satisfied during the execution of the block.

In *Flex*, an event is defined as anything that may be identified with a specific instant such as the *start* and the *end* of some computation. Time is represented as a sequence of discrete, quantized instants. Control blocks can be nested. *Flex* supports absolute as well as relative timing constraints. It also defines a language construct *interval* to specify periodic tasks.

Flex provides an adequate level of abstractions for time. However, timing constraints are enforced on the control blocks which are at instruction level. Intermixing timing constraints into computational aspects of an application makes system design difficult. In addition, *Flex* only supports sequential real-time computing, and does not support parallel and distributed real-time computing.

Real-Time Mentat

The Real-Time Mentat programming language [38] is based on a combination of a macro data-driven computational model [30, 80] and an object-oriented programming paradigm.

In Mentat, a node in a program graph of a data-flow model is called an actor, which represents the computation at that point. Unlike other real-time programming languages which disallow recursion, dynamic memory allocation, dynamic process instantiation, etc., which are time unbounded and unpredictable actions, real-time Mentat provides the option of having timing constraints declared as either *hard* or *soft*. Only the actor which has *hard* timing constraints will be restricted to have bounded and predictable execution time and resource requirement. Furthermore, attributes: *start time, deadline, duration,* and *period* can be imposed on actors.

The difficulties with real-time Mentat include: (1) for distributed real-time systems, time dependencies among distributed objects are difficult to map to the data-flow model; (2) the time related language constructs in real-time Mentat are defined to specify the timing constraints on an individual actor, i.e. an individual operator; timing relations between different operators are not supported; (3) as with most other real-time programming languages, the time requirements are embedded inside the computational code which makes them difficult to reuse. Moreover, embedded constraints complicate the verification of time correctness and computational correctness.

RTC++

RTC++[43] is another extension of C++. It provides two additional entities: (1) real-time objects which are based on multi-threaded object models; (2) timing constraints. Real-time objects are active objects with timing constraints. The language is designed based on the belief that "a real-time programming language should support (a) timing specification at each operation, (b) timing constraints for each statement, and (c) specifying a periodic task".

Hence, RTC++ provides *within, at*, and *before* statements to express statement-level timing constraints and provides a *cycle* statement to specify a periodic task.

The disadvantages of RTC++ include;

1. Synchronization is needed within individual multi-threaded objects;
2. The multiple-threaded model increases difficulty in analyzing temporal behavior of distributed systems;
3. RTC++ unnaturally separates objects into two different types: ordinary objects and active objects. For example, object `A` and object `B` may have the same computational behaviors, but `A` has timing constraints, while `B` does not. In RTC++, `A` and `B` would belong to different classes and be unrelated. Furthermore, if timing constraints were added to object `A`, `A` would have to be redefined even though its computational behavior remained unchanged.
4. Since timing related constructs in RTC++ may apply at statement level, understanding, modifying and analyzing programs may be difficult.
5. Timing constraints in RTC++ are associated with individual objects, which limit programmers from specifying timing constraints between different objects in distributed systems.

Some other researchers have also attempted to extend SmallTalk with real-time features, such as RealTimeTalk [22]. Unfortunately, such languages suffer similar difficulties as previously discussed languages.

Composition Filters

Akşit et al. [11] noticed that in most current real-time object-oriented languages, changing application requirements or changing real-time specifications in a sub-class may require excessive redefinitions. They refer to this as *real-time specification inheritance anomaly* and propose the concept of *real-time composition filters* to resolve this anomaly.

One cause of real-time specification inheritance anomalies is mixing of real-time specification and computational code. Therefore, in order to avoid the real-time specification anomaly and to be able to reuse both application code and time specification, application code and real-time specification must be separated.

Composition filters add real-time "filters" on object interfaces. Timing constraints are specified in real-time filters, but the definition of member functions are time independent. These filters affect incoming and/or outgoing member function calls. The conceptual data structure of composition filters is illustrated in Figure 2.11.

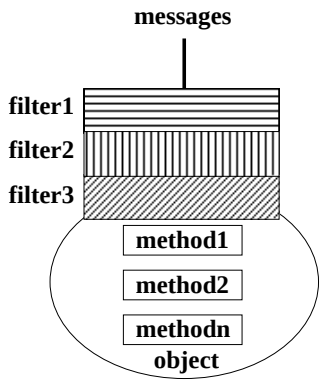


Figure 2.11: Composition Filter

With composition filters, application code and timing constraint specifications can be separately reused, but a difficulty in using composition filters in distributed real-time systems is that they only provide mechanisms for specifying timing constraints for individual objects. Timing dependencies among a group of distributed objects are not addressed in composition filters. Furthermore, no formal semantics are given for composition filters.

Esterel

`Esterel` [21, 20] is a synchronous real-time programming language. It is based on the *strong synchrony hypothesis* that `delay` statement terminates *exactly* when its ending event occurs and all control transmission inside a program and all simple operations take no time. In Esterel, the only instruction which may take time are the ones *explicitly required* to do so. A program has no internal clock and it simply reacts instantaneously to external stimuli itself producing some stimuli to its output lines, and does nothing in absence of external stimuli. Furthermore, there is no dynamic creation of processes in an Esterel program. With such framework, the need for nondeterminism in the system is suppressed.

However, the strong synchrony hypothesis that Esterel is based on seems unrealistic because real machines can not be infinitely fast. Besides, distributed real-time applications make synchrony even harder to apply. As with other real-time programming languages we have discussed so far, Esterel programs also embed timing requirements within underlying computational code which makes independent modification of timing requirements or computational code impossible and limits the reusability of correct general programming code.

Currently, a large project Rapide [57] launched by David Luckham at Stanford focuses efforts on developing a new technology for building large-scale, distributed, time-sensitive systems. The constraint language provides constructs for abstract specification of the behavior of a distributed system, including timing requirements. It is a constraint-based language, based on the timed poset model, that supplies constructs for defining patterns of events that are required (or forbidden) in the computation of a distributed system. Component interfaces and architectures may be given detailed abstract specifications using the specification language. However, with current Rapide constraint language, the patterns of events may not be only based on the component interface.

A good survey of the historical development of real-time languages and a comparison of high-level real-time programming can be found in Halang [39].

Our general programming approach builds on the earlier work of Frølund and Agha [31, 32] who developed *synchronizers* to provide a declarative specification of coordination constraints on groups of actors. Our approach is similar to Akşit's `real-time filters` approach in that we both construct new linguistic constructs to encapsulate the timing constraints. However, differently from real-time filters, which have the potential difficulty in abstractly specifying timing constraints among a group of distributed objects, our approach abstracts the timing constraints among a group of objects into distinct entities which provide declarative specification for group-wise timing constraints, and hence overcome the difficulty suffered by real-time filters.

Furthermore, RTsynchronizers support the description of timing constraints in absolute and relative time for a group of objects. As special objects, RTsynchronizers do not interfere with the computational aspects of the underlying computation, except message delivery. However, the existence of such objects in a computing system ensures that the system has correct behavior in the time domain, hence satisfies the criteria for real-time computing — "system which has to respond to externally-generated input stimuli within an *finite* and *specified* period" [95, 23].

RTsynchronizers express the timing constraints not as part of computational objects, but as independent entities. These entities provide an environment in which autonomous computational objects are constrained and forced to synchronize with each other in real time.

Separating real-time aspects from computational aspects of a real-time system, both conceptually and 'physically', i.e. using distinct objects, provides a high level modularity, and makes it easier and less error prone to design and maintain distributed real-time applications.

In addition, RTsynchronizers are transparent to computational objects, and their functionalities only depend on the interfaces, not the representations, of computational objects. Such properties make it possible to achieve code reusabilities in two directions: the same group of computational objects can be constrained by different RTsynchronizers in different applications — computational code reuse; the same RTsynchronizer can apply to a different group of computational objects as long as they have the same interfaces — constraint specification reuse.

Chapter 3

Language Description — Syntax and Semantics

Chapter 2 has explained, from a language point of view, our methodology in developing real-time systems and applications. We have designed a special language entity RTsynchronizer to specify timing properties of real-time applications. Such an entity is independent of how other computational components operate, but depends on the computational components' interfaces. The existence of such entities does not interfere with the computational object behaviors, but ensures that these behaviors comply with the specified timing constraints.

In this chapter, we formally define RTsynchronizers. We define a simple programming language (`RTCL`, Real-Time Coordination Language) that consists of RTsynchronizers and distributed computational objects. We also define an operational semantics for this language. The language is kept simple and focused on the essential features of RTsynchronizers and distributed objects.

The language is based on the Actor model of concurrent computation [2]. In the Actor model, messages are guaranteed to be delivered at the receiving objects, i.e. weak fairness is assured. However, the fairness property alone is not sufficient in real-time computing. *When* a message should be delivered is subject to real-time timing constraints, not just causal order constraints.

The semantics of our language extends that described in Agha et.al [7, 8, 32], and defines the meaning of quantitative real time timing constraints in distributed systems. In our language, message delivery is subject to constraints specified by RTsynchronizers.

In Section 3.1, we give a set of examples illustrating how RTsynchronizers perform their roles. These examples use pseudo code to give intuition about the RTsynchronizer concept and their functionality. Further, we give a formal syntax definition for our new language. In Section 3.2, we briefly explain the mathematical structure used to represent a distributed system in the basic Actor model, and define the mathematical structure representing a distributed system in the extended Actor model ART — the Actor model with real-time extensions, and discuss constraint evaluations and the operational semantics of ART systems. Section 3.3, presents conclusions drawn from this.

3.1 Language Syntax and Examples

In this section, we define the formal syntax of our language RTCL — Real-Time Coordination Language. This language is an extension of the simplified actor language (SAL) described in Agha [2]. In particular, the new type of actors, RTsynchronizers, and the associated primitives which create such objects and activate their constraint enforcement, are added into the primitive actor language. Before we give a formal definition of RTCL syntax, and discuss each newly added element in detail, we first give a set of examples to illustrate how RTsynchronizers are used. These examples are written in pseudo code aimed to achieve intuitive understanding about RTsynchronizers' functionality.

3.1.1 Examples

In this section, we present canonical examples using RTsynchronizers. With RTsynchronizers, computational behaviors and quantitative temporal behaviors can be specified separately and transparently with each other. These examples illustrate that we can reuse the correct general computational code in real-time applications; reuse the correct timing constraints code on a different group of objects as long as the objects have the same interface as required by RTsynchronizers; and compose simple time constraint specifications to express complex timing requirements.

Example 1: Real-Time Synchronization

In distributed systems, we may require that two or more different actions conducted by different objects must happen *at the same time* or 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 must happen *simultaneously* with the grasp of the glass by the other hand.

With RTsynchronizer, the timing constraints can be specified as [1] in Figure 3.1.

```
RTsynchronizer SimuRobot(actor lHand, rHand) {
 Constrain
   lHand.release() : rHand.grasp() [0, 0];
}
```

Figure 3.1: Synchronized Robot Hands

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

[1] When the sender, the sequence number are not important and the boolean expression is `true` or `1`, a pattern can be simplified to only use the `receiver.cX`.

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 delivery periodic.

```
RTsynchronizer PeriodicE(actor O; real P) {
      /* P is the period. */
  Declare
    boolean start = 0;
    int count = 0;
  Constrain
    ((*, *), O.getInfo(), start == 0) : ((O, *), O.getInfo(), 1) [P, P];
    ((O, count), O.getInfo(), start == 1) :
                          ((O, count + 1), O.getInfo(), 1) [P, P];
  Update
    ((O, *), O.getInfo(), 1) : start = 1; count ++;
}
```

Figure 3.2: Periodic Events

Example 3: Time Bounded Buffer

Consider an extension to the traditional producer/consumer with a bounded buffer problem, where each product must be held inside the buffer no sooner than time `t1`, and no longer than `t2`. We define an RTsynchronizer to enforce time constraints without changing the computational behavior of regular bounded buffers as shown in Figure 3.3.

```
RTsynchronizer TBBuffer(int size; real t1, t2; actor Producer, Consumer){
  Declare
    int count = 0;
  Constrain
    ((*,*), Consumer.get(), count == 0) (inf, inf);
    ((*,*), Producer.put(), count == size) (inf, inf);
    Producer.put() : Consumer.get() [t1, t2]
  Update
    Producer.put() : count ++;
    Consumer.get() : count --;
}
```

Figure 3.3: Time Bounded Buffer

Example 5: Dyeing Control System

Selic et al. [79] describe an industrial dye control process as a canonical process control problem. The problem is as follows:

There is a sensor set which has three sensors: a liquid level sensor (`SL`), a temperature sensor (`SH`) and a pressure sensor (`SP`). There is a dye valve (`Dye`) which controls the incoming dye liquid and a drain valve (`Drain`) which controls the outgoing dye liquid. There is a heat controller (`HCtrl`) which controls the temperature and a pressure controller (`PCtrl`) which controls the pressure. Sensors are responsible for reading the corresponding data periodically with period `p` and signaling the corresponding control devices and valves. Finally, there is a timer `Timer` to monitor the duration of the dyeing process, and a hatch `Hatch` to cover the dyeing tank. The dyeing system is shown in Figure 3.4.

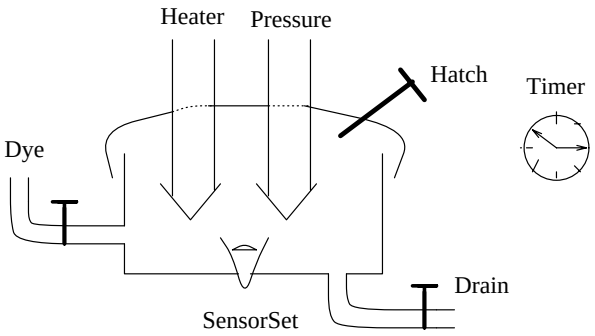


Figure 3.4: Dyeing Machine

The following constraints must be satisfied:

Req. 1 When the dyeing liquid reaches level L, the temperature reaches H, and the pressure reaches P, the dyeing process must start within d time units, if not active already.

Req. 2 The dyeing process takes T time units.

Req. 3 The dyeing liquid must be drained out within D time units after the process finishes.

Req. 4 For safety reasons, the temperature and the pressure in the tank must be reduced to room levels, before the dyeing liquid can be drained out.

Req. 5 All three sensors read their data periodically with period p.

Req. 6 When a sensor finds the related requirement is not met, it signals the corresponding device. The device must make adjustment within t time units.

Req. 7 Other requirements: the drain (or dye) valve must not open when heat or pressure is being applied.

We use actors to model each component of this equipment. Specifically, we use four RT-synchronizers to represent timing requirements listed above: `STSyn` (Figure 3.5) is responsible for starting and terminating the dyeing process (**Req.** 1, 2, and 3); `Safety1Syn` (Figure 3.6)

enforces safety properties (**Req.** 4); `PeriodSyn` (Figure 3.7) causes the sensors to repeat their actions periodically (**Req.** 5); `ReactSyn` (Figure 3.8) enforces the requirement that devices react to messages within the applicable timing constraints (**Req.** 6); and finally, qualitative coordination relations, such as **Req. 7**, are enforced by `Safety2Syn` (Figure 3.9).

```
RTsynchronizer STSyn(actor SL, SH, SP, Timer;  real L, H, P, T, d, D){
     /* L, H, P are required parameters for level, temperature
      and pressure, respectively; T is the time that the dyeing
      process lasts; d is the time limit by which the  dyeing process has to
      start; and D is the time limit by which the liquid has to be
      drained out. */
Declare
   Boolean: ready := start := Lready := Tready := Pready := false;
   Real: readyTime; startTime
Constrain
  (Timer.start, ready) [0, readyTime + d];  /*Req. 1 */
  (Timer.end, start) [startTime + T, startTime + T];  /*Req. 2 */
  (Timer.end, start) : (SL.sig(l), l == 0 && start) [0, D];  /*Req. 3 */
Update
  (Timer.start, ready) : startTime = cTime() ;
        /*recording the time when the process starts */
  ((SL, *), SL.sig(l), l == L) : Lready := true;
  ((SH, *), SH.sig(t), t == H) : Tready := true;
  ((SP, *), SP.sig(p), p == P) : Pready := true;
  Lready && Tready && Pready : ready := true
                               start := true
}
```

Figure 3.5: Starting and Terminating RTsynchronizer

In this section, we have shown a few canonical examples using RTsynchronizers. As we have seen, with RTsynchronizer, computational behaviors and quantitative temporal behaviors of applications can be specified separately and transparent to each other, and hence the code can be reused independently. Furthermore, with RTsynchronizer, incremental modification and modular specification of timing constraints become possible.

3.1.2 Syntax

The previous subsection has shown us how RTsynchronizers are used. In this subsection, we formally define the syntax for our newly designed language — Real Time Coordination Language (RTCL). As in `SAL`, behaviors are described as lambda abstractions in an untyped, call-by-value lambda calculus in `RTCL`. A message has three components: a tag, which is attached by system implicitly; a receiver actor identification name; and a value which constitutes the contents of the message. Before a message is delivered at the receiver actor, it must pass a 'safety' check enforced by RTsynchronizers. When an receiver actor delivers a message, it applies its current lambda abstraction to the value part of the message.

`RTCL` also maintains the primitive operations defined in `SAL`, namely:

```
RTsynchronizer Safety1Syn(actor SH, SP, Drain; real rmT, rmP){
   /* rmT is the room temperature; rmP is the room pressure. */
Declare
   Real: ct, cp;
         /* variables for current temperature and pressure. */
   Constrain
      ((*, *), Drain.open, ct > rmT || cp > rmP) [inf, inf];  /*Req. 4 */
   Update
     ST.sig(t) : ct := t;
     SP.sig(p) : cp := p
}
```

Figure 3.6: Safety 1 RTsynchronizer

```
RTsynchronizer PeriodSynL(actor SL; real p) = new PeriodicE(actor SL; real p);
RTsynchronizer PeriodSynT(actor ST; real p) = new PeriodicE(actor ST; real p);
RTsynchronizer PeriodSynT(actor SP; real p) = new PeriodicE(actor SP; real p);
      /*Req(5) , reuse previously defined PeriodicE*/
}
```

Figure 3.7: Periodic RTsynchronizer

```
RTsynchronizer ReactSyn(actor SL, SH, SP,HCtrl, PCtrl, Drain, Dye;
                        real L, T, P, t){
   /* L, T, P are the required liquid level, temperature, and
    Pressure, respectively; t is the time limit when the control
    device has to response. */
 Constrain  /* Req. 6 */
   ((*,*), ST.sig(v), v < T) : HCrtl.heat [0, t];
   ((*,*), ST.sig(v), v > T) : HCrtl.cool [0,t];

   ((*,*), SP.sig(v), v < P) : PCrtl.increase [0, t];
   ((*,*), SP.sig(v), v > P) : PCrtl.decrease [0,t];

   ((*,*), ST.sig(v), v < L) : Dye.close [0, t];
   ((*,*), ST.sig(v), v > L) : HCrtl.open [0,t];
}
```

Figure 3.8: React RTsynchronizer

```
RTsynchronizer Safety2Syn(actor HCtrl, PCtrl, Drain, Dye){

 Declare
  Boolean: hCtrlOn = pCtrlOn = false
 Constrain  /* Req. 7 */
   hCtrlOn : Drain.open  [inf, inf];
   hCtrlOn : Dye.open  [inf, inf];
   pCtrlOn : Drain.open  [inf, inf];
   pCtrlOn : Dye.open  [inf, inf];
 Update
   HCtrl.open : HCtrolOn = true;
   HCtrl.close : HCtrolOn = false;
   PCtrl.open : PCtrolOn = true;
   PCtrl.cpen : PCtrolOn = true;
}
```

Figure 3.9: Safety 2 RTsynchronizer

- `send` sending messages.
- `ready` indicating the actor's state change, i.e., the actor terminates the processing of the current message and is ready for accepting a new message with new behavior obtained by processing the previous message.
- `newadr` allocating a new fresh actor name.
- `initbeh` initializing newly allocated actors with their first behavior.

The notion of constraints is represented as a new primitive language feature and modeled as RTsynchronizers in `RTCL`. RTsynchronizers do not contribute or interfere with the underlying computation, but only observe and constrain the message deliveries at computational actors. Constraints are enforced or unenforced by an actor executing the `enforce`, or `unenforce` operations, respectively:

- `enforce` activating RTsynchronizers.
- `unenforce` terminating RTsynchronizers' constraint power.

The language syntax for `RTCL` involves 6 primitive categories, namely, irreducible values (**Value**); variables (**Var**); expressions (**Expression**), **Value** $\subseteq$ **Expression**; functional expressions (**Fexpression**), **Fexpression** $\subset$ **Expression** (such expressions do not have side effects when being evaluated); RTsynchronizers (**RTsyn**); message patterns (**Pattern**); and constraints (**Constraint**).

Let x range over variables **Var**, a over actor names **Actor**, s over **RTsyn**, v over **Value**, and e over **Expression**. A value in `RTCL` may be an actor, an RTsynchronizer, a variable, a pair or a lambda abstraction; and an expression is inductively defined as a value, an n-ary operation, or an application. The formal syntactic definitions for values and expressions are expressed in Figure 3.10.

$$
\begin{array}{lcl}
v & ::= & a \mid s \mid x \mid \lambda x.e \mid \mathtt{pr}(v_1, v_2) \\
e & ::= & v \mid e_1\ e_2 \mid f_n(e_1, \cdots, e_n)
\end{array}
$$

Figure 3.10: Syntax for Values and Expressions in RTCL

The syntax for the other 3 categories, i.e. RTsynchronizers, patterns and constraints are defined in Figure 3.11.

```
RTsyn        ::=  name (formals) {
                      Declare {state_var [ = value] ;} *
                      Constrain {cnstrnt_exp ;} *
                      Update {(pattern | b_exp):{state_var =value;} *}*
                  }
cnstrnt_exp  ::=  cnstrnt ; constrnt_exp | cnstrnt || cnstrnt_exp
cnstrnt      ::=  pattern : pattern time_dur |
                  pattern time_dur
pattern      ::=  ((a, i), a.e, b_exp)
time_dur     ::=  [d, d]
formals      ∈    Var
state_var    ∈    Var
a            ∈    Actor
i            ∈    Int
e            ∈    Expression
b_exp        ∈    Fexpression
d            ∈    ℜ≥0 ∪ {inf}
inf               special value indicating infinity
```

Figure 3.11: Syntax for RTsynchronizers, Constraints and Patterns

We use an example to illustrate an RTCL program and explain the reduction of such a program. We focus our attention on the new objects — RTsynchronizers and constraint evaluations.

In Section 3.1.1, we described the time-bounded buffer example. The RTCL program for the time-bounded buffer, the producer, the consumer and the RTsynchronizer are depicted in Figure 3.12, where the manager enforces an RTsynchronizer which ensures that each item produced by produce must be kept in buffer for at least 5.5 time units, but not longer than 10 time units; and the buffer can hold 20 items.

As shown in the above example, the constructs we discussed in Chapter 2 may be directly mapped into our RTCL programs.

```
Producer = Rec(λb. λself. λm.
  seq(if(m == put) send(self, put);
      ready(b(self))))

Consumer = Rec(λb. λself. λm.
  seq(if(m == get) send(self, get);
      ready(b(self))))

Manager = λ⟨p, c⟩.
  enforce(TBbuffer(p, c, 5.5, 10, 20))

TBbuffer = λp. λc. λt1. λt2. λs.
  RTsyn(
    declare
      int count :=0;
    constrain
      Producer.put() : Consumer.get() [t1, t2];
      ((*,*), consumer.get(), count == 0)  (inf, inf);
      ((*,*), Producer.put(), count == s)  (inf, inf);
    update
      Producer.put() : count ++;
      Consumer.get() : count --
  ))
```

Figure 3.12: The RTCL Program for the Time-Bounded Buffer Example

3.2 Semantics of ART System

This section is divided into two subsections. We first introduce the basic Actor semantics developed by Agha et. al [2, 7, 8]. We then extend the basic actor model in the real-time domain and further discuss the operational semantics of the extended model.

3.2.1 Basic Actor Semantics

Actors are self-contained, concurrently interacting entities. They communicate via message passing which is asynchronous and fair. Therefore, at any given time instance, the only elements in an actor system are a finite set of actors and a finite set of unprocessed messages. These two sets define the system *configuration* at that time instance.

Actor names are unique in the system. Each actor has its own independent mail queue which is not shared by any other actors. Actor behavior is history sensitive. It depends on the messages the actor has received.

The reactive aspect of actors indicates that an actor is in different state when no messages have arrived. We call such a state an *idle* state and denote it as (v), where v is the actor's current behavior: the behavior that will respond to the next incoming message. The state when an actor is processing a message is called a *busy* state and is denoted as $[e]$, where e is the expression currently being processed by the actor. There is also another state in which the actor is newly created and no initial behavior has been assigned. Such a state is called *unknown* and denoted as $(?x)$, where x is the name of the creating actor. In summary, an actor may be in one of the states shown in Figure 3.13.

as	::=	$(?x)$	unknown
	|	(v)	idle
	|	$[e]$	busy

Figure 3.13: Three Possible Actor States

A message is a structured entity containing the following components:

1. a *tag* which distinguishes it from all other messages in the system;
2. a *target* which is the actor identification name to which the content of the message is to be delivered; and
3. *content* which contains the information available to the target actor when it processes the message.

A message is denoted as:

$$m ::= < tag : aid \Leftarrow cv >^{2}$$

where, $tag \in \mathcal{MT}$ = {all possible message tags}, $aid \in \mathcal{AN}$ = {all possible actor names}, $cv \in \mathcal{CV}$ = {all possible message contents}.

[2]Message tags are useful in pattern-matching. We may omit this field when it is not used.

An actor system configuration is defined as a structured entity:

$$\langle\!\langle\ \alpha\ \mid\ \mu\ \rangle\!\rangle$$

where α is a function which maps an actor name ($\mathcal{AN}$) to its state ($\mathcal{AS}$), i.e., $\alpha : \mathcal{AN} \rightarrow \mathcal{AS}$; μ is currently unprocessed message set.

The operational semantics of an actor system is defined by the transition rules given in Figure 3.14.

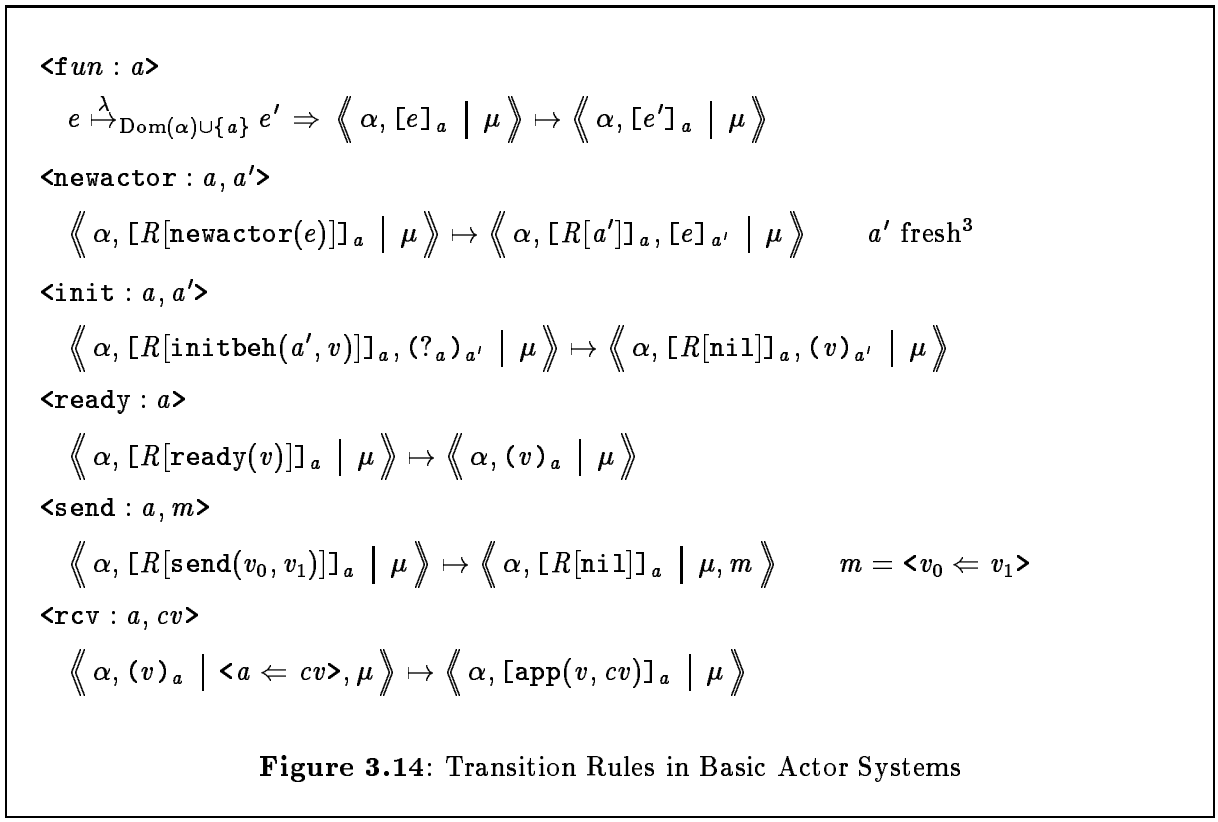


Figure 3.14: Transition Rules in Basic Actor Systems

The <fun : a> transition defines the effect on actor state when the actor a executes an internal computation step. The internal computation may be specified by any sequential language. Our semantics is not dependent on such sequential semantics, hence we use $\overset{\lambda}{\mapsto}$ to abstract sequential reduction. A newly created actor must be given an initial behavior by its creator, as shown in the rule <init : a, a'>. The <ready : a> transition indicates the state change of the actor a from *busy* to *idle* and thereby becoming ready for processing a new message with possibly new behavior. The semantics of the primitive operation send is given by the <send : a, m> rule. The newly sent message is added to the message pool μ of the system configuration.

The actor's reactive characteristic is reflected in the receive transition <rcv : a, cv>. When an actor is in an *idle* state and there are messages sent to it, the actor will pick one message at a time, bind the actual parameters carried in the message to its formal parameters and process it non-preemptively until termination. Meanwhile, the processed message is removed from the

[3] R is the reduction context

system. The fairness of actor semantics guarantees that all the messages in the system will *eventually* be delivered and processed by their target receivers.

It is worth noticing that the `rcv` transition is the only transition which is not a computational reduction internal to individual actors. For instance, when an actor a does a `send` transition, it is actually reducing the internal expression of $[R[\texttt{send}(v_0, v_1)]]$ to expression $[R[\texttt{nil}]]$, and the result of such a reduction is that a new message is added to the system. In contrast, the `rcv` transition creates a new reduction expression. However, once the new expression is created, how to reduce the expression is deterministic and internal to an actor's behavior. In addition, such a reduction process is non-stoppable. Hence the only control point left for us when we need to impose more restrictive requirements than fairness is the non-internal reduction transition `rcv`.

As we may have noticed, in this standard actor model, there is no notion of quantitative time. The *combined* order [24] (or logical time [49]) is the only constraint imposed on message delivery of the reactive actors. Such a model may abstract the underlying computation of even real-time applications when the real-time constraints are lifted from the applications. However, when timing constraints are enforced on computations of systems in which the quantitative time was originally not a concern, the basic actor model shows its insufficiency. Thus, it is necessary to extend the basic model in order to model distributed real-time computing.

3.2.2 ART Systems and Its Transition Rules

As we have explained, the basic Actor model captures the fundamental properties of general purpose distributed computing in which only logical time is concerned, and message deliveries on individual objects are constrained only by the computational causal order. However, in real-time distributed computing, not only must the computational causal order be followed to ensure functional correctness, but also some quantitative precedence orderings among different (maybe independent) computing units must be enforced to achieve the timeliness of such systems. The ART model is an extension of the actor model, which takes the quantitative time aspect of real-time computing into consideration. In particular, statically specified real-time constraints (which are encapsulated by RTsynchronizers), and dynamically created *expectation instance*s on message deliveries are the new basic element of ART systems.

Similar to an Actor system configuration, an ART configuration is a structured entity representing the system state at a given instant of time. In contrast to the standard actor configuration, the given time instance τ, is represented as the state of an special actor `Timer`, at which the system snapshot is taken now becomes part of the configuration. Formally, an ART system configuration is defined as follows:

Definition 3.1 (ART Configuration)

$$\left\langle\!\!\left\langle\ \alpha,\ (\tau)_{\texttt{Timer}}\ \middle|\ \mu\ \middle|\ \langle\ \sigma \parallel \varsigma\ \rangle\ \right\rangle\!\!\right\rangle$$

where as defined in the basic Actor configuration (see section 3.2.1)

- α is an actor map, which maps a finite set of actor names to their behavior;
- `Timer` is a special actor whose state (τ,$\tau \in \mathcal{T}$) indicates the current global time with respect to the configuration when the snapshot is taken.

- μ is a finite multi-set of messages, yet to be processed;
- σ is an RTsynchronizer map, which maps a finite set of RTsynchronizer names to their states;
- ς is a multi-set of expectation instances on message deliveries that must be satisfied. The formal definition of expectation instance will be given in Definition 3.2.

RTsynchronizers are special actors that may be in an *unenforced* state ($\otimes$) or currently active with state s as showing in Figure 3.15.

rs	::=	$\otimes$	unenforced
	|	(s)	enforced

Figure 3.15: Two Possible RTsynchronizer States

In the presence of RTsynchronizers, messages are divided into two categories: *guided* messages and *unguided* messages. Informally, guided messages are those which are indicated by at least one message pattern specified in an RTsynchronizer. In other words, guided messages need to be guided by RTsynchronizers in regard to their delivery at message destinations; and they may affect RTsynchronizers' state changes as well. In contrast, unguided messages are not affected by RTsynchronizers and do not affect RTsynchronizer states either.

An RTsynchronizer's control power may be turned off by an actor performing `unenforce` operation. Thereafter, the RTsynchronizer has an unenforced state $\otimes$. The RTsynchronizer state change as well as the `unenforce` actions are atomic. When the RTsynchronizer is enforced, its state contains the values of state variables, static constraint expressions and update expressions.

Another component in an ART system configuration is the *expectation instance* set ς, which is a collection of expectation instances. Expectation instances are defined as follows:

Definition 3.2 (Expectation Instance)
The structured unit $s : p \triangleright \delta$ is called an expectation instance, where p is a message pattern defined in the state of enforcing RTsynchronizer s; δ specifies the time interval during which a message of pattern p is expected to be delivered. δ is of the form $[d1, d2]$, where $d2 \geq d1 \geq 0, d_i \in \mathcal{T} + \{\inf\}$, denoting the starting and ending time points. $\mathcal{T}$ is a subset of a non-negative real number domain without cluster points, i.e. $\mathcal{T} \subseteq \Re_{\geq 0}$.

Before we present operational semantics for ART systems, we first define a group of notations and a group of functions based on these notations. These definitions will simplify later explanations and formula.

Definition 3.3 (Semantic Domains)

- $\mathcal{AN}$: the set of all possible actor names
- $\mathcal{AS}$: the set of all possible actor states represented by expressions of the language. Expressions provide the following information:

  - execution state (*unknown*, *idle* or *busy*)
  - the actors script and acquaintances
  - the current expression it is reducing if in a *busy* state
- $\mathcal{RN}$: the set of all possible RTsynchronizer names
- $\mathcal{RS}$: the set of all possible RTsynchronizer states represented by expressions of the language. Expressions provide the following information:
  - the enforcing state (unenforced, or enforcing)
  - values of state variables
  - the static constraint expressions
  - the static update expressions
  - the dynamic expectation instances
- Π: the set of all possible message patterns
- $\mathcal{M}$: the set of all possible messages
- $\mathcal{D}$: the set of all possible time intervals
- $\mathcal{E}$: the set of all possible expectation instances
- $\mathcal{T}$: nonnegative real number domain with no clustered points

For convenience, we assume that a ranges over domain $\mathcal{AN}$, s over $\mathcal{RN}$, sb over $\mathcal{RS}$, p over Π, m over $\mathcal{M}$, d over $\mathcal{D}$, and e over $\mathcal{E}$, respectively, using the convention that "x range over X" means that the meta-variable x and decorated variants such as x', x_0, ..., range over the set X.

Definition 3.4 (Functions)

- α, a map which maps actor names to actor states.
  $\alpha : \mathcal{AN} \rightarrow \mathcal{AS}$
- σ, a map which maps RTsynchronizer names to RTsynchronizer states.
  $\sigma : \mathcal{RN} \rightarrow \mathcal{RS}$
- Sat_σ, a boolean function which returns T if message m satisfies pattern p under the enclosing RTsynchronizer s.
  $Sat_\sigma : \mathcal{M} \times \Pi \times \mathcal{RN} \rightarrow \{\mathtt{T}, \mathtt{F}\}$

$$Sat_\sigma(m, p, s) = \begin{cases} \mathtt{T} & \text{if the evaluation of pattern matching returns T} \\ \mathtt{F} & \text{otherwise} \end{cases}$$

- $Safe_\sigma$, a boolean function which returns T if message m is safe for delivering at time τ with respect to an expectation instance $(s : p \triangleright [t, t1])$.
  $Safe_\sigma : \mathcal{E} \times \mathcal{M} \times \mathcal{T} \rightarrow \{\mathtt{T}, \mathtt{F}\}$

$$Safe_\sigma((s : p \triangleright [t, t1]), m, \tau) = \begin{cases} \mathtt{F} & \text{if } (\sigma(s) \neq \otimes) \ \wedge (Sat_\sigma(m, p, s) = \mathtt{T}) \wedge ((\tau < t) \vee (\tau > t1)) \\ \mathtt{T} & \text{otherwise} \end{cases}$$

- Dlv_σ, a function which returns T if message is deliverable with respect to the current expectation instance set.
  $Dlv_\sigma : \mathcal{M} \times 2^{\mathcal{E}} \times \mathcal{T} \rightarrow \{\mathtt{T}, \mathtt{F}\}$

$$Dlv_\sigma(m, \varsigma, \tau) = \begin{cases} \mathtt{T} & \text{if } (\exists e \in \varsigma \ (Safe_\sigma(e, m, \tau) = \mathtt{T})) \ \vee \ (\forall e \in \varsigma \ (Sat_\sigma(m, p, s) = \mathtt{F})) \\ & \quad \text{where } e = s : p \rhd [t1, t2] \\ \mathtt{F} & \text{otherwise} \end{cases}$$

- $NewEI_\sigma$, a function which returns a set of new expectation instances created by delivering a message.
  $NewEI_\sigma : \mathcal{M} \times \mathcal{T} \rightarrow 2^{\mathcal{E}}$
  $NewEI_\sigma(m, \tau) = \cup_{s \in \mathrm{Dom}(\sigma)} \{(s : p' \ \rhd [t1 + \tau, t2 + \tau]) \mid (Sat_\sigma(m, p, s) = \mathtt{T}) \wedge ((p : p' \ [t1, t2]) \text{ is in } \sigma(s)) \}$ [4]

- $RemEI_s$, a function which returns a set of expectation instances satisfiable by delivering a message.
  $RemEI_s : \mathcal{M} \times 2^{\mathcal{E}} \times \mathcal{T} \rightarrow 2^{\mathcal{E}}$

$$RemEI_\sigma(m, \varsigma, \tau) = \{e \mid (e \in \varsigma) \ \wedge \ (Safe_\sigma(e, m, \tau) = \mathtt{T})\}$$

- $Update_s$, a function which changes the RTsynchronizer to a new state.
  $Update_s : \mathcal{M} \times \mathcal{T} \times \mathcal{RN} \rightarrow \mathcal{RS}$

$$Update_\sigma(m, \tau, s) = \begin{cases} newState & \text{if } \exists p \ Sat_\sigma(m, p, s), \text{ and } p \text{ is an update pattern in } s \\ \sigma(s) & \text{otherwise} \end{cases}$$

  newState is the new state of RTsynchronizer s with state variables being assigned new values as specified in update section.

As in Alur's Automata model [13] in which state changes are instantaneous and may be followed by a stall to let time progress, we consider time as part of the system configuration, and use a special actor `Timer` to model the global time entity. The concurrency between state change and time progression is modeled as interleaving between state transition and time transition. This approach allows us to uniformly express the system state evolution and time progression.

In particular, an ART system evolves in two functional different phases: *quiescent* time progress in which objects' states remain unchanged (which is similar to macro-step in Gerber [34, 35]); and *instantaneous* state change, a micro-step.

As noted in the latter part of Section 3.2.1, transitions other than `rcv` transition are computational reductions internal to individual actors and may not directly affect other actors' states . Furthermore, they are deterministic in the sense that the chosen step is determined by the sequential semantics. In contrast, the `rcv` transition is not a computational reduction internal to individual actors. Instead, it is a bridge connecting outside entities to the internal behavior. Hence the `rcv` rule provides a possible point for enforcing further requirements that are outside the computational behavior specification. In other words, real-time constraints are

[4] There is no need to do the $Safe_\sigma$ testing because the function $NewEI_\sigma$ is called only when the message is safe to be delivered at the destination. The same reasoning applies for $RemEI_\sigma$ and $Update_\sigma$ functions.

imposed on the standard actor computation; these constraints are reflected in deciding when the `rcv` transition may be taken.

More formally, we extend the local transitions defined for a sequential language ($\stackrel{\lambda}{\mapsto}$), and standard actor system transitions (a rigorous theory of actor systems is developed in [8]) by providing transitions for the real-time actor program in Figure 3.16, Figure 3.17 and Figure 3.18.

The ART transition rules are similar to the standard operational actor semantics on the meaning of `fun`, `newactor`, `ready` [5], and `send`. As we have explained before, these transitions are actors' internal reduction. They do not affect other objects' states in the system — thereby the RTsynchronizer map (σ) and the expectation set (ς) remain the same before and after the transitions.

Two new primitive transitions `enforce` and `unenforce` are added into the ART transition systems. With such transitions, ART systems can create or deactivate an RTsynchronizer. Different from its corresponding part in standard actor semantics, the transition rule `<rcv : m>` has three parts:

- The prerequisite for taking the transition at the current configuration is that it is safe to deliver message m, i.e. delivering m does not violate the timing constraints. A message is safe to deliver if either m is not an expected message, or the current time τ falls in the expected time interval for the message m, namely function $Dlv_\sigma(m, \varsigma, \tau)$ must evaluate to be `true`.

- The same as the standard `rcv` rule, the behavior of the receiving actor is applied to the message contents. Thus the underlying computation is started.

- If the message m is *guided* by RTsynchronizers, the delivery of such message may cause:

  - RTsynchronizers' state change, formulated as $Update_\sigma(m, \sigma,)$;
  - furthermore, if m is an expected message, delivering it will satisfy an expectation instance in the system; hence, some satisfied expectation instance will be removed from the expectation instance set ($RemEI_\sigma(m, \varsigma, \tau)$);
  - the message m may also be a "causal" message in a constraint expression, thereby triggering creation of new expectation instances. Hence new expectation instances ($NewEI_\sigma(m, \tau)$) are added into the expectation set.

System failure happens when there is a yet-to-be-satisfied expectation instance whose expected duration cannot be satisfied now or in the future. In this case, the system goes into a failure state $\perp$. The failure transition, also instantaneous, is given in Figure 3.17.

As we have seen, all these transitions are instantaneous — τ remains the same before and after the transitions. Interwoven with these instantaneous transitions, time may quiescently progress forward. Since there is only one timer in the system, each individual entity in the system will have a uniform view of the new time (Figure 3.18).

[5] The `ready` transition is embedded in `rcv` transition in ART

[7] A well formed RTsynchronizer expression contains values of local variables and static constraints

[8] When an message is sent, it is tagged with sending actor's information. After `send`, the sending actor's message counter is increased.

[9] $\uplus$ is multi-set minus operation.

<fun : a>

$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow$

$$\left\langle\!\left\langle \alpha, [\![e]\!]_a,\ (\tau)_{\texttt{Timer}} \mid \mu \mid \langle\, \sigma \parallel \varsigma \,\rangle \right\rangle\!\right\rangle \mapsto \left\langle\!\left\langle \alpha, [\![e']\!]_a,\ (\tau)_{\texttt{Timer}} \mid \mu \mid \langle\, \sigma \parallel \varsigma \,\rangle \right\rangle\!\right\rangle$$

<newactor : a, a'>

$$\left\langle\!\left\langle \alpha, [\![R[\texttt{newactor}(e)]]\!]_a,\ (\tau)_{\texttt{Timer}} \mid \mu \mid \langle\, \sigma \parallel \varsigma \,\rangle \right\rangle\!\right\rangle \mapsto$$

$$\left\langle\!\left\langle \alpha, [\![R[a']]\!]_a, [\![e]\!]^0_{a'},\ (\tau)_{\texttt{Timer}} \mid \mu \mid \langle\, \sigma \parallel \varsigma \,\rangle \right\rangle\!\right\rangle \qquad a' \text{ fresh } [6]$$

<enforce : s>

$$\left\langle\!\left\langle \alpha, [\![R[\texttt{enforce}(sb)]]\!]_a,\ (\tau)_{\texttt{Timer}} \mid \mu \mid \langle\, \sigma \parallel \varsigma \,\rangle \right\rangle\!\right\rangle \mapsto$$

$$\left\langle\!\left\langle \alpha, [\![R[s]]\!]_a,\ (\tau)_{\texttt{Timer}} \mid \mu \mid \langle\, \sigma \cup \{(sb)_s\} \parallel \varsigma \,\rangle \right\rangle\!\right\rangle$$

sb is a well formed RTsynchronizer behavior expression [7]

<unenforce : s>

$$\left\langle\!\left\langle \alpha, [\![R[\texttt{unenforce}(s)]]\!]_a,\ (\tau)_{\texttt{Timer}} \mid \mu \mid \langle\, \sigma, (sb)_s \parallel \varsigma \,\rangle \right\rangle\!\right\rangle \mapsto$$

$$\left\langle\!\left\langle \alpha, [\![R[\texttt{nil}]]\!]_a,\ (\tau)_{\texttt{Timer}} \mid \mu \mid \langle\, \sigma, \otimes_s \parallel \varsigma \,\rangle \right\rangle\!\right\rangle$$

<send : a, m>

$$\left\langle\!\left\langle \alpha, [\![R[\texttt{send}(v_0, v_1)]]\!]^n_a,\ (\tau)_{\texttt{Timer}} \mid \mu \mid \langle\, \sigma \parallel \varsigma \,\rangle \right\rangle\!\right\rangle \mapsto$$

$$\left\langle\!\left\langle \alpha, [\![R[\texttt{nil}]]\!]^{n+1}_a,\ (\tau)_{\texttt{Timer}} \mid \mu, m \mid \langle\, \sigma \parallel \varsigma \,\rangle \right\rangle\!\right\rangle$$

if $m = \texttt{<}(a, n) : v_0 \Leftarrow v_1\texttt{>}$ [8]

<rcv : a, cv>

$$\left\langle\!\left\langle \alpha, [\![R[\texttt{ready}(v)]]\!]_a,\ (\tau)_{\texttt{Timer}} \mid m, \mu \mid \langle\, \sigma \parallel \varsigma \,\rangle \right\rangle\!\right\rangle \mapsto$$

$$\left\langle\!\left\langle \alpha, [\![\texttt{app}(v, cv)]\!]_a,\ (\tau)_{\texttt{Timer}} \mid \mu \mid \langle\, (Update_\sigma(m, \tau, s))_{s \in \mathrm{Dom}(\sigma)} \parallel \varsigma \uplus \{ei\} \uplus NewEI_\sigma(m, \tau) \,\rangle \right\rangle\!\right\rangle$$

if $m = \texttt{<}tag : a \Leftarrow cv\texttt{>}$, $Dlv_\sigma(m, \varsigma, \tau) = \texttt{T}$, and $ei \in RemEI_\sigma(m, \varsigma, \tau)$ [9]

Figure 3.16: ART Systems Transition Rules — Instantaneous Transitions

<failure :>

$$\left\langle\!\left\langle\, \alpha,\ (\tau)_{\texttt{Timer}} \;\middle|\; \mu, m \;\middle|\; \langle\, \sigma, (sb)_s \parallel \varsigma, s : p \rhd [t1, t2] \,\rangle \right\rangle\!\right\rangle \mapsto \bot$$

if $Sat_\sigma(m, p, s) \wedge \tau \geq t2$, where $\bot$ is the failure configuration.

Figure 3.17: ART Systems Transition Rule — Failure Transitions

<progress : d>

$$\left\langle\!\left\langle\, \alpha,\ (\tau)_{\texttt{Timer}} \;\middle|\; \mu \;\middle|\; \langle\, \sigma \parallel \varsigma \,\rangle \right\rangle\!\right\rangle \mapsto \left\langle\!\left\langle\, \alpha,\ (\tau')_{\texttt{Timer}} \;\middle|\; \mu \;\middle|\; \langle\, \sigma \parallel \varsigma \,\rangle \right\rangle\!\right\rangle$$

$$\tau' = \min\{t1 \mid \exists\ (s : p \rhd [t1, t2]) \in \varsigma, \tau < t1\}$$

Figure 3.18: ART Systems Transition Rules — Quiescent Transitions

3.3 Conclusion

Functionality and timeliness of real-time systems are two different concerns. They can be carried by two different types of abstractions, namely, computational objects (actors) and RT-synchronizers, respectively.

We have defined an operational semantics for such computational objects and RTsynchronizers in our test language RTCL. The constructs proposed in Chapter 2 may be directly mapped into RTCL.

From a programming language point of view, embedding different concerns in different types of objects, we have achieved better modularity and code reusability. From a semantics point of view, by distinguishing the underlying computation and the constraint environment in which the computation is carried out, the computation model (as we have derived ART) is cleaner and more intuitive.

Furthermore, in our transition rules, we separate *when* a transition is allowed to apply from *how* to choose an optimal transition rule from a set of possible transitions. The *what* is in the actor state. The *when* is in the synchronization part. The new transition rules use information from both. For instance, when there is more than one message available that satisfies the constraints, which message will be delivered first is omitted from our semantics. This further separates the semantics of ART systems from the scheduling mechanism and detailed implementation strategies.

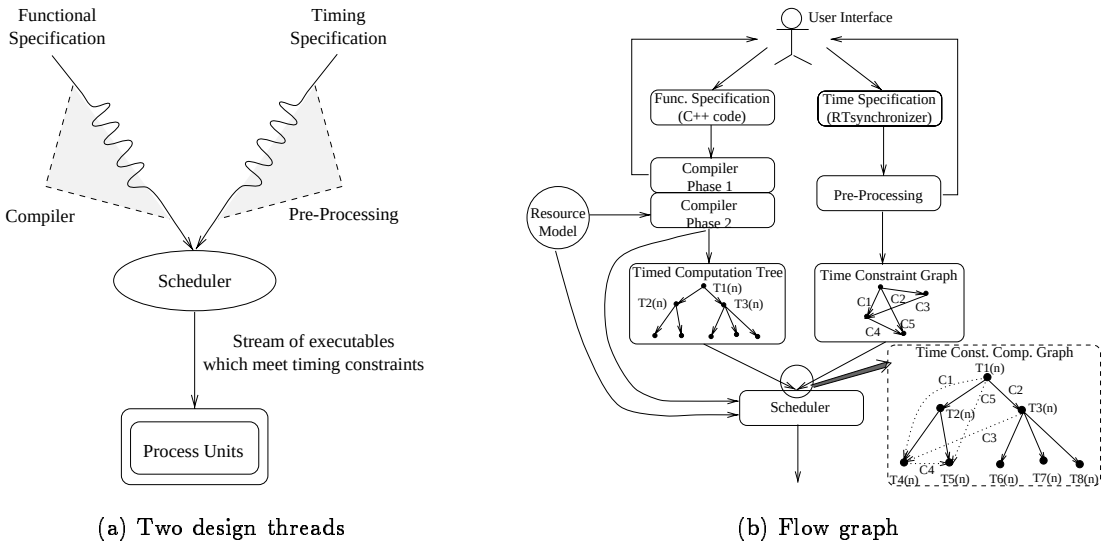
Our approach to defining real-time semantics is inspired by the recent research in formal specification languages for real-time systems. Even though these languages often are in the form of extended automata [12, 13, 63], or process algebras [59, 77], we use transition systems for the simplicity of composing real-time constraint semantics with standard actor semantics given by transition rules.

Formally defining the operational semantics for our language RTCL not only provides the theoretical soundness for our separation theory, but also prepares us for developing implementation as we will discuss in the next chapter.

Chapter 4

Language Implementation

In our language paradigm, a real-time application software design can be considered as two threads that start from separated computational specifications and timing specifications, and eventually join together at a scheduler. (See in Figure 4.1(a)). The scheduler combines timing and functional information from the two streams and dispatches compiled computational code to process units with respect to the timing constraints (Figure 4.1(b)).



(a) Two design threads

(b) Flow graph

Figure 4.1: Real-Time Application Design Flow Graph under Separation Paradigm

A program written in a general-purpose programming language, like C or C++, must be compiled before it can be executed on a computer. The first phase of compiling checks syntax and attempts to pinpoint incorrect user code. Only when the program is grammatically correct can the compiler continue to the second phase — integrating hardware features and generating architecture-dependent executable code.

Similarly, for user specified timing constraints, we need a tool to detect infeasible specifications, and to give programmers feedback at an early design stage. An obvious and necessary

condition is that timing constraint specifications must be feasible before a scheduler can find a schedule. Unfortunately, most real-time systems [?, 33] perform schedulablity analysis on compiled and transformed code. One disadvantage of such an approach is that scheduling information given by analysis of compiled code is hard to use at the source code level. In addition, in compiled code, the timing constraints may be rearranged, re-divided, and redistributed, which further complicates the analysis procedure; not to mention that if the specification is infeasible, all this compilation and transformation work is wasted.

We propose a static pre-processing approach to detect infeasible constraints at the source code level and develop an algorithm it. Although, in general, for a given set of tasks with timing constraints, the problem of finding a schedule to satisfy the constraints is NP-hard [87], distinguishing infeasible constraint sets can be done in polynomial time. We present the pre-processing algorithm in detail in Section 4.1.

Given that a set of timing requirements is feasible, we are still faced with a number of high-level trade-offs in structuring the implementation architecture. We devote Section 4.2 to the discussion of high-level design decisions made in constructing such systems. The decisions are concerned with (1) overall systems architecture framework, (2) strategies used for resolving issues such as inter-object consistency and evaluation of constraints, and (3) the trade-off between guaranteeing constraint satisfaction and passively monitoring or reporting violated constraints.

Finally, in section 4.3, we summarize the results and draw conclusions.

4.1 Pre-Processing — Generating Time Constraint Graphs and Detecting Infeasible Time Constraints

According to our ART model discussed in previous chapters, real-time application software is a synthesis of two orthogonal concerns: a set of distributed objects, which produce the functionalities of the software; and a set of distributed RTsynchronizers which enforce time constraints among groups of those objects and hence assure the timeliness of the software.

It is obvious that the consistency of timing constraint specifications is a necessary condition for a scheduler to be able to find a schedule. By infeasible timing constraint specification, we mean that the specification cannot be realized in a global time (i.e. regardless of the architecture resource, such as processor speed, communication latency, etc.). Although, in general, for a given set of tasks with timing constraints, the problem of finding a schedule to satisfy the constraints is NP-hard [87], distinguishing infeasible constraint sets can be done in polynomial time if the constraint expression is in an appropriate format.

Unfortunately, schedulability analysis is traditionally done at a late stage, and is based on compiled code, and is thus machine dependent. We believe it to be useful and important to provide real-time application programmers early feedback regarding timing specifications, and the feedback should be based on their high-level timing requirement source code.

The goal of pre-processing is to detect infeasible timing constraint specifications using high-level language constructs and therefore provide software designers useful information to make early corrections. An infeasible timing requirement implies that the specification cannot be met regardless of the scheduling strategy or architecture used. However, even if a timing specification is not infeasible, it may not be possible to find a schedule on a given architecture. In other words, early infeasibility detection is architecture independent, i.e. if a timing constraint specification

is detected as infeasible, no architecture platform can implement such constraint requirements. However, even if the timing constraint specification is not being detected infeasible, or even if it is feasible, it does not mean for a given architecture platform, such a specification is realizable. The situation is similar to `while` loops or any `recursive` calls in general programming language programs. The program may be syntactically correct, but fail at run-time because of stack overflow in a given architecture.

4.1.1 Mapping Constraints into Linear Inequalities

Recall that each constraint is of the form `A : B [d1, d2]`, where `A` and `B` are message patterns. The semantic meaning of `A : B [d1, d2]` is that when a message of pattern A is delivered at the message's destination object, it requires a message of pattern `B` to be delivered (assuming such a message is in the system) in the time interval `[d1, d2]` which is relative to the time when `A` happens. Using t_a and t_b, respectively, to represent the time when `A` and `B` happen, the above constraint is denoted as two inequalities:

$$\begin{cases} t_a \Leftrightarrow t_b \ \leq \ \Leftrightarrow d1; \\ t_b \Leftrightarrow t_a \ \leq \ d2 \end{cases} \tag{4.1}$$

Theorem 4.1 *The assertion of `A : B [d1, d2]` is equivalent to the linear inequalities system (4.1) in that the assertion is feasible if and only if the inequalities (4.1) have at least one set of solutions.*

The proof of the theorem is straightforward. But the conclusion serves as the foundation of our detecting algorithm: the methods used to solve linear inequality systems may be applied in deciding if a real-time constraint specification is feasible.

Recall that separate RTsynchronizers work conjunctively. Hence the union of constraint assertions in a system is of positive conjunctive normal form, i.e. of the form:

$$(C_{11} \vee \cdots \vee C_{1n}) \wedge \cdots \wedge (C_{m1} \vee \cdots \vee C_{mk}),$$

where each component represents constraints in an RTsynchronizer, and C_{ij} is the ith constraint in RTsynchronizer j.

To give an example, assume we have two RTsynchronizers: `rtsyn1` and `rtsyn2` as follows:

```
rtsyn1(){
  ...;
  Constraint:
    p1 : p2  [d1, d2] ||
    p3 : p4  [d3, d4];
     /* denoted as A || B */
    p5 : p6  [d5, d6];
     /* denoted as C */
 }
```

```
rtsyn2(){
  ...;
  Constraint:
    p1 : p3  [d1, d2]
     /* denoted as D */
 }
```

These constraint assertions may be specified as:

$$(A \vee B) \wedge (C) \wedge (D)$$

Even though the monotone conjunctive normal form (MCNF) may fail to map directly to a linear inequality system like (4.1), it is valuable because a polynomial algorithm has been discovered to convert MCNF into a monotone disjunctive normal form (MDNF), which then may be directly transformed into a linear system.

4.1.2 Transforming MCNF of Constraint Assertions into MDNF

Given monotone conjunctive normal form, we can use an approach similar to the naive polynomial multiplication method to expand CNF into disjunctive normal form (MDNF) as shown in the following algorithm:

Algorithm for transforming MCNF into MDNF

```
MCNF-to-MDNF(⋀_{i=1}^{n} C_i){
      while n != 1 do {
         C = expand(C_1, C_2);
         MCNF-to-MDNF(C ∧ (⋀_{i=1}^{n-2} C_i))}
      return C_n
 }
```

With the disjunctive normal form, each component in our context can be transformed into a linear inequality system. The Bellman-Ford Algorithm can be used to decide if the linear inequality system has a solution. In the next subsection, we will first discuss how to build the inequality systems from the disjunctive normal form of constraint assertions written in our language specification constructs. Thereafter, we will apply the Bellman-Ford algorithm in our application domain. Finally, we will discuss the complexity of the pre-processing algorithm.

4.1.3 Generating Time Constraint Graphs and the Bellman-Ford Algorithm

By our language definition, constraint assertion has two formats: `A : B timeInterval` or `B timeInterval`. In the case of `A : B timeInterval`, the `timeInterval` is relative to the time when a message of pattern `A` is delivered; while in the case of `B timeInterval`, the `timeInterval` is a absolute global time span. However, if we imagine the boolean constant `true` indicating the time when the 'universe' is created, or the origin point (`0`) of time, the absolute global time span `timeInterval` is then relative to the origin point (`0`). Thus, the second format is a special version of the first format. Further, the origin point of time is unique in the system because of the assumption that timers in RTsynchronizers are synchronized with each other.

Applying Theorem 4.1, each conjunction of constraint assertions may mapped to a set of linear inequalities 4.2:

$$\begin{cases} x_1 \Leftrightarrow x_2 \leq d_1; \\ x_i \Leftrightarrow x_j \leq d_k; \\ \cdots \\ x_m \Leftrightarrow x_n \leq d_p; \end{cases} \tag{4.2}$$

Given such a constraint system, we can construct a corresponding *constraint graph* $G = (V, E)$, where

$$V = \{v_0, v_1, \cdots, v_n\}$$

and

$$E = \{(v_i, v_j) : x_j \Leftrightarrow x_i \leq d_k \textit{ is a constraint}\} \cup \{(v_0, v_1), (v_0, v_2), \cdots, (v_0, v_n)\}$$

Each vertex v_i in the graph, for $i = 1, 2, \cdots, n$, corresponds to one of the n unknown variables x_i; each directed edge in the graph corresponds to one of the m inequalities involving two unknowns. The additional vertex v_0 is incorporated, to guarantee that every vertex is reachable from it. If $x_j \Leftrightarrow x_i \leq d_k$ is a constraint, then the weight for edge (v_i, v_j) is $w(v_i, v_j) = d_k$, while the weight of each edge leaving v_0 is 0. Figure 4.2 shows an example of mapping an linear system into a constraint graph.

$$\begin{cases} x_1 \Leftrightarrow x_2 \leq 0; \\ x_1 \Leftrightarrow x_5 \leq \Leftrightarrow 1; \\ x_2 \Leftrightarrow x_5 \leq 1; \\ x_3 \Leftrightarrow x_1 \leq 5; \\ x_4 \Leftrightarrow x_1 \leq 4; \\ x_4 \Leftrightarrow x_3 \leq \Leftrightarrow 1; \\ x_5 \Leftrightarrow x_3 \leq \Leftrightarrow 3; \\ x_5 \Leftrightarrow x_4 \leq \Leftrightarrow 3; \end{cases} \Longrightarrow$$

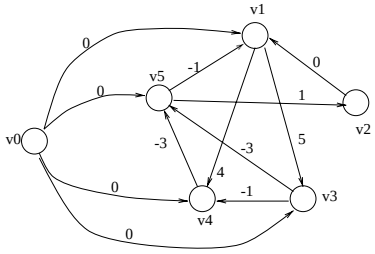


Figure 4.2: Time Constraint Graph: An example of mapping a linear inequality system into a constraint graph

Theorem 4.2 *Given a system $Ax \leq b$ of constraints, let $G = (V, E)$ be the corresponding constraint graph. It is feasible to satisfy the constraint specification if and only if G contains no negative-weight cycles.*

The theorem is directly derived from graph theory [26]. With such a constraint graph, the Bellman-Ford algorithm may be used to detect if the graph has negative-weight cycles. The algorithm is given below.

Bellman-Ford Algorithm [26]

```
Bellman-Ford(G, w, s)
  Initialize-Single-Source(G, s)
  for i =  1  to  |V[G]| - 1
     do for each edge (u, v) in E[G]
         do Relax(u, v, w)
  for each edge (u, v) in E[G]
     do if d[v] > d[u] + w(u, v)
        then return false
  return true
```

```
Initialize-Single-Source(G,s)
  for each vertex v in V[G]
     do d[v] = infinity
  d[s] = 0


Relax(u, v, w)
  if d[v] > d[u] + w(u, v)
    then d[v] = d[u] + w(u, v)
```

When the conjunctive component indicates that the constraints are infeasible, we may continue to use the same algorithm to test the next conjunctive component until we either obtain a positive return or have tested the rest of the components and found that the constraints

are infeasible to satisfy. In case of failure, the designers must reconsider the specification and make corrections.

We use a simple example to show how the pre-processing may help us detect inconsistent timing constraints. Consider again the time-bounded producer/consumer example. Assume that the `Producer` and the `Consumer` are started by another object `Starter` with the following constraints: the `Producer` must start its `put` 5 time units after the `Starter` sends out the `start` signal; the `Consumer` must start its `get` within 10 time units after the `Starter` sends out the `start` signal. In addition, every item the `Producer` puts in the buffer, must be kept in the buffer for at least 10 time units. These timing requirements are enforced by `TBBuffer2` as shown in Figure 4.3.

```
RTsynchronizer TBBuffer2(int size; actor Starter, Producer, Consumer){
  Declare
    int count = 0;
  Constrain
    ((*,*), Consumer.get(), count == 0) [inf, inf];
    ((*,*), Producer.put(), count == size) [inf, inf];
    Starter.start() : Producer.put() [5, inf];
    Starter.start() : Consumer.get() [0, 10];
    Producer.put() : Consumer.get() [15, inf]
  Update
    Producer.put() : count ++;
    Consumer.get() : count --;
}
```

Figure 4.3: Time Bounded Buffer

For simplicity, we use $x1$ and $x2$ to denote the time when `Starter.start()` and `Producer.put()` occur, respectively, and $x3$ for `Consumer.get()`. The above constraints may be expressed in the set of linear inequalities as shown in Figure 4.4. The corresponding constraint graph indicates that there exists a negative-weight cycle, hence the timing constraint specification is infeasible to realize by any given architecture.

$$\begin{cases} x1 - x2 \leq -5; \\ x3 - x1 \leq 10; \\ x2 - x3 \leq -15; \end{cases} \Longrightarrow$$

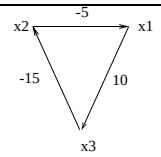


Figure 4.4: Infeasible Timing Constraints

4.1.4 Significance of Conducting Infeasibility Detection

The pre-processing serves two purposes. First, it provides system designers early feedback about their specification at the source code level. Second, since the timing specifications will eventually be combined with the the functional specification to generate both the functionally and chronologically correct executables, discarding inconsistent constraints at an early stage

will enormously decrease the time taken by a scheduler to combine those two dimensions and generate a correct schedule. The second advantage will become more clear after the next section.

4.1.5 Complexity of the Infeasibility Detecting Algorithm

As we have seen, the detecting process has two steps: first, convert the conjunctive normal form into disjunctive normal form; second, apply Bellman-Ford Algorithm to each component of the DNF.

Theorem 4.3 *Given a conjunctive normal form* $f = \bigwedge_{I \in F} \bigvee_{i \in I} x_i$, *it can be transformed into equivalent disjunctive normal form* $g = \bigvee_{J \in G} \bigwedge_{j \in J} x_j$ *in* $o(n^{C-1})$, *where* $C = |F|$, *and* $n = \max\{|I|,\ I \in F\}$.

We skip the formal proof and instead give an simple example to show the rationality as follows:

$$\underbrace{\underbrace{(x_1 \vee x_2) \wedge (x_3 \vee x_4)}_{1} \wedge (x_5 \vee x_6)}_{2}$$

$$\begin{aligned}
=_1 \quad & ((x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_2 \wedge x_3) \vee (x_2 \wedge x_4)) \wedge (x_5 \vee x_6) \\
=_2 \quad & (x_1 \wedge x_3 \wedge x_5) \vee (x_1 \wedge x_3 \wedge x_6) \vee (x_1 \wedge x_4 \wedge x_5) \vee (x_1 \wedge x_4 \wedge x_6) \vee \\
& (x_2 \wedge x_3 \wedge x_5) \vee (x_2 \wedge x_3 \wedge x_6) \vee (x_2 \wedge x_4 \wedge x_5) \vee (x_2 \wedge x_4 \wedge x_6)
\end{aligned}$$

In this example, without losing generality, we assume that each term in CNF has the same number of literals, using a distribution law to expand the last two terms, as in $(=_1)$, the result of the expansion has $n_1 \times n_2$ terms, and the process takes the $n_1 \times n_2$ amount of time, where n_1, and n_2 is the number of literals in each term, respectively. Repeat the process with the newly generated DNF of size $n_1 \times n_2$, as in $(=_2)$, and it gives the final DNF of size $n_1 \times n_2 \times n_3$, and takes the $n_1 \times n_2 \times n_3$ amount of time. In our example $n_1 = n_2 = n_3 = 2$.

For any arbitrary CNF, we can repeatedly expend one term at a time, and the time it takes to expend any two terms is the product of the two terms' sizes, as shown in the above example. Therefore, the conclusion of theorem holds.

Theorem 4.4 *Bellman-Ford algorithm runs in time* $O(VE)$.

The rationale for the proof is as follows: the initialization takes $\Theta(V)$ time, each of the $|V| \Leftrightarrow 1$ passes over the edges takes $O(E)$ time, and the `for` loop takes $O(E)$ time, hence the algorithm takes $O(VE)$ time.

Our constraint specifications are transformed into a set of constraint graphs. In case the running result of the first constraint graph returns negative, that is, there is no feasible solution for it, we have to run the Bellman-Ford algorithm for the next constraint graph until we either check all the constraint graphs or get a positive result from an individual run of Bellman-Ford algorithm. Therefore, in the worst case, the Bellman-Ford algorithm is repeated D times, where D is the number of conjunctive terms in the transformed DNF.

Hence, the whole procedure for detecting the feasibility of constraint specification takes exponential time in the worst case.

We have provided a simple polynomial time algorithm based on the derived constraint graph for early detection of inconsistent constraints at the source-code level, specified with our language constructs. Such infeasibility detection provides system designers early feedback about their specification at the source code level, and makes early adjustment on timing requirements possible. Furthermore, since the timing specifications will eventually be combined with the computational specification to generate both functionally and timely correct executables, discarding inconsistent constraints at an early stage may enormously decrease the time taken by a scheduler to combine those two dimensions and generate a correct schedule.

4.1.6 Related Work

Real-time scheduling research is still mostly based on compiled code [29, 84] and the schedulability analysis is task (or process) oriented [78, 27]. One of the difficulties with these approaches is that the feedback based on the compiled code must be mapped back to UN-compiled source code, which most of the time is not straightforward. Even though model checking on real-time specification languages has achieved promising results [44]. In some respects, our work is similar to that carried out independently by Al Mok who has studied infeasible timing constraints detection based on a real-time logic [60, 61]. On the other hand, our work focuses on a programming language; we detect source code level infeasibility.

Our general programming approach builds on the earlier work of Frølund and Agha [31, 32] who developed *synchronizers* to provide a declarative specification of coordination constraints on a group of actors. However, the semantics of RTsynchronizers are quite different, and are focused on *quantitative* constraints [71, 70, 64]. The infeasibility detection algorithm is the result of applying linear inequality and graph theories [26] into the real-time application domain.

4.2 Implementation Issues and Resolving Strategies

In general, for real-time applications, especially hard real-time applications, the variation range of the systems is known a priori, and hence gives the possibility to explore all the computation paths at compile time; for hard real-time systems, sometimes it is necessary to investigate all the computation paths in order to guarantee the time constraints.

A *timed computation tree* is a computation tree in which each node is associated with a time value, or more generally, a function of some input values which returns a time value. A *time value* (function) represents the time required to complete that specific computation, and is system architecture, application code, and input dependent.

With the current compiler [9, 67] and partial evaluation [66, 19] technology, we can easily generate such a graph. Combining the timed computation tree with the time constraint graph (Section 4.1) may take exponential time even though the algorithm is straightforward — search the timed computation tree and add each edge in time constraint graph into the computation tree. Fortunately, the combination is done off-line. However, since some of the time information may be parameterized and may not be known until run time, on-line scheduling is still unavoidable [97].

Given such an exponentially large *constraint computation graph* — the composition of the timed computation tree and time constraint graph — it is very difficult, if not impossible, for any scheduler to search the graph and produce a correct schedule within an acceptable amount of time. Hence, for a given schedule generated by a scheduler with limited time, we need to

validate the corresponding constraints and perhaps report the violation of constraints before dispatching the job to the process units.

Since the constraints are distributed in RTsynchronizers, and the constrained actors within each RTsynchronizer are also distributed, it is important to maintain the inter-object consistency in order to guarantee the correct result of constraint evaluation.

In the following subsections, we will discuss RTsynchronizer structures, the distribution of RTsynchronizers' state information and strategies for maintaining inter-object consistency among RTsynchronizers and their controlled actors. Thereafter, we will discuss the scheduling strategy.

4.2.1 RTsynchronizer Data Structures

Recall that RTsynchronizers, like other regular objects, have their encapsulated states; but unlike other regular objects, RTsynchronizers do not receive or send messages, instead, they are able to observe and enforce timing constraints on message deliveries at controlled objects. Messages are hence divided into two categories according to the interest of RTsynchronizers: *unguided* messages, which do not affect RTsynchronizers' state or cause any other messages to be constrained, and the counter part *guided* messages. *Guided* messages are further divided into two types: *triggering* messages and *time constrained* messages. Syntactically, the triggering messages are identified by the pattern on the left-hand side of ":" in a constraint expression; while the constrained messages are captured by the pattern on the right-hand side of ":" and are in general associated with constraining time intervals. Semantically, the deliveries of triggering messages may cause RTsynchronizers' state change, or cause another message's delivery time to be constrained, but it is not itself constrained. Only time constrained messages, as their name indicates, are restricted to be delivered in the constrained time frame.

Consider the following constraint:

```
O1.mthd1() :  O2.mthd2() [2, 3]
```

Assume there is a message `m1` of pattern `O1.mthd1()` at time `0`, and `m1` is a guided message, but not constrained. Therefore, it can be delivered at time `0` at object `O1`. Upon the delivery of message `m1` at time `0`, a message of pattern `O2.mthd2()` is *constrained* to be delivered within time `[2, 3]`. Further, assume at time `1`, RTsynchronizer observes an message `m2` of type `O2.mthd2()`. Since the current time does not fall in the constrained interval, `m2` cannot be dispatched to the destination object `O2`, instead being buffered in the queue. Only when time passes `2`, but before it reaches `3`, the constraint is satisfied and the message `m2` is dispatched to the target object.

In summary, when a *triggering* message is delivered, it may change RTsynchronizer states, or generate an *expectation* instance — a constrained message pattern and its associated time frame in which a message of such pattern must be delivered. An *expectation* instance is `satisfied` when the constrained message is dispatched for delivery at the message's destination object in the *expected* time interval.

It should be clear by now that in addition to user defined state variables, each RTsynchronizer needs a mechanism to bookkeep its observations and *expectation*s. The constraint manager is for such purpose.

The constraint manager maintains a set of queues for each constrained message pattern specified in the RTsynchronizer, where each element of the queue is an expectation — a struc-

tured entity `(s : p ▷ timeInterval)` indicating the constrained message and its expected time during which the message may be delivered. When an expectation is satisfied, it is removed from the queue.

Figure 4.5 illustrates the date structure of an RTsynchronizer and a message flow depicted by (1) to (5).

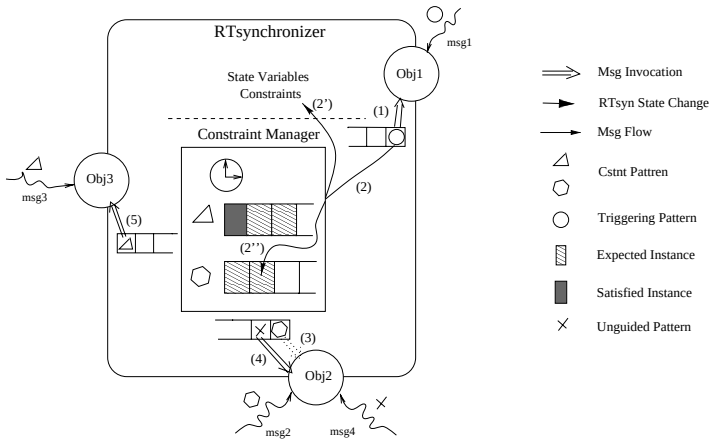


Figure 4.5: RTsynchronizer's structure and a message delivery sequence in the system: (1) a *triggering* message `msg1` is delivered at `obj1` which causes RTsynchronizer state changes (2') and generates an *expectation* (2"); message `msg2` targeted at `obj2` *fills* an expectation, but the current time does not fall in the constrained time interval, and therefore is not allowed for delivery (3); since there is an *unguided* message `msg4` in the `obj2`'s mail queue, instead, it is delivered at `obj2` (4); message `msg3` targeted at `obj3` *satisfies* an expectation at current time and is dispatched on the actor (5).

Because the same message may match multiple constraint message patterns in different RTsynchronizers, it is essential that RTsynchronizers have the same view about *current time* so that the evaluation of satisfaction will be consistent.

4.2.2 Synchronized Clocks

From the previous subsection, we know that each RTsynchronizer must agree on the *current time*, which implies that the RTsynchronizers' clocks need to be synchronized.

Even though it is infeasible to implement *perfectly synchronized* clocks in distributed systems, *approximately synchronized* clocks, i.e. clocks that satisfy the following two conditions are implementable [62]:

$$(1+\rho)^{-1} \leq \frac{C_p(t) \Leftrightarrow C_p(t')}{(t \Leftrightarrow t')} \leq (1+\rho) \tag{4.3}$$

$$|C_p(t) \Leftrightarrow C_q(t)| \leq \epsilon \tag{4.4}$$

where $C_p(t)$ is the reading of clock C_p at real-time t and $\epsilon > 0$.

Condition 4.3 ensures that clock drifting rates are bounded; and the second condition assures that difference between any two different clock readings is bounded within some skew ϵ.

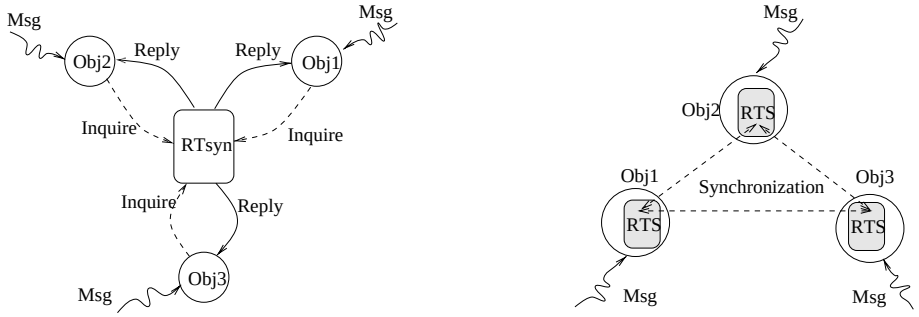
As discussed in Liskov [55], in many real-time applications we can use *approximately* synchronized clocks to simulate *perfectly* synchronized ones.

We assume that in our framework, RTsynchronizers' clocks are approximately synchronized with each other and with the universal wall-clock time. In addition, the approximation satisfies the above two conditions.

4.2.3 Distribution of RTsynchronizer State Information

At the programming level, objects do not need to know they are under some RTsynchronizers' control. However, at implementation level, each controlled object knows its controlling RTsynchronizers. The function of RTsynchronizers is to enforce certain timing behaviors of a group of autonomous objects. There are two extreme approaches to implement such functionality, namely, *centralized* and *decentralized* approach.

In the centralized approach, each controlled object communicates with its controlling RTsynchronizer before delivering any guided messages. More specifically, if a message is guided by an RTsynchronizer, the message destination object first inquires the controlling RTsynchronizer, and only when the object gets the 'permissive' reply, can the object deliver the message. As shown in the figure 4.6(a), the controlling RTsynchronizer can be a bottleneck, especially when the controlled group is large.



(a) Centralized Approach

(b) Decentralized Approach

Figure 4.6: Centralized & Decentralized RTsynchronizers

On the other extreme of the spectrum, the controlling RTsynchronizer clones n copies, where n is the number of controlled objects under this RTsynchronizer. Each clone is embedded inside an individual object as showing in Figure 4.6(b). The advantage of this approach is that we avoid the bottleneck problem. However, it gives rise to other problems. One problem is an inter-object consistency problem. In order to maintain state consistency among different clones, each clone must synchronize with others. Using the simplest two-phase commit protocol [37] to achieve the synchronization among the n clones, we need $2(n \Leftrightarrow 1)$ extra synchronization messages. The other problem is that the security of autonomous objects may be broken: the object may look at its own copy of the controlling RTsynchronizer and know an other object's control condition; although individual objects should not know if other objects are under some

RTsynchronizer's control. Therefore, there is a trade-off between centralized and decentralized approaches.

We choose a mixed approach in which there is still a central RTsynchronizer, but partial state information is distributed into related individual objects. This way, we not only avoid the bottleneck problem, but also maintain the controlled object's integrity. In addition, the non-functional communication is reduced in comparison with the decentralized approach.

Recall that the state of an RTsynchronizer consists of explicit and implicit parts as shown in Figure 4.5. The explicit part contains state variables and constraints, which are statically user defined, while the constraint manager comprises the implicit part of an RTsynchronizer. It contains a synchronized clock, and a set of queues to bookkeep the expectations. In order to lessen the bottleneck contention and reduce synchronization communications, instead of cloning the whole RTsynchronizer, we make an *image* of each expectation queue and embed the image into related object. In addition, we extend the controlled object state further to include a synchronized clock as showing in Figure 4.7.

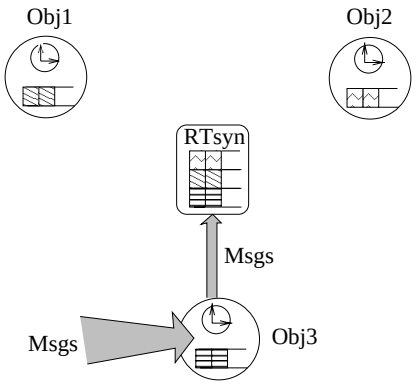


Figure 4.7: Hybridizing Centralized and Decentralized Approaches

The difference between the image of an expectation queue and the corresponding one inside the constraint manager is that in the image of an expectation queue, each element is a *relaxation* of the matching instance in the original queue. Recall that each expectation instance is of the form `(rtsyn : ptn ▷ timeInterval)`, and the pattern is defined by a 3-tuple `(tag, receiver.cX, b_exp)`. The *relaxation* of an expectation instance is defined by replacing the `b_exp` with `true`. The rationale behind the relaxation is to avoid leaking information and to maintain the object's integrity.

With the information from their images of the expectation queues and their clocks, constraint objects are able to prevent some non-eligible deliveries of messages without asking their RTsynchronizers. Only when a message satisfies the destination object's local expectation instance, does the object further inquire of the RTsynchronizer. If the message is eligible for delivery, synchronization happens among the RTsynchronizer, the message destination object, and those objects whose image queues will be affected by delivering this message, while the RTsynchronizer plays the coordinator role.

As an example, let us consider the `time-bounded buffer` example given in section 2.3. For convenience, we include the example below:

```
Producer(){
  methods:
    put();
    other();
}

Consumer(){
  methods:
    get();
    other();
 }
```

```
RTsyner TBBuffer(int size, real time,
          actor Producer, Consumer){
  Declare
    int count = 0;
  Constrain
    count == 0   : Consumer.get (inf, inf);
    count == size: Producer.put (inf, inf);
    Producer.put : Consumer.get [0, time]
  Update
    Producer.put : count ++;
    Consumer.get : count --;
}
```

Figure 4.8 illustrates the process of delivering a constrained message: a `put` message is sent to a `Producer` (1). Since it is a constrained message, and the `producer`'s local expectation queue initially is empty, thus trivially satisfies empty expectation, `Producer` at this time has to query its controller `TBBuffer` (2). If the buffer is full, i.e. `count == size`, the `put` message is not valid for execution and put back into the `Producer`'s mail queue (5). In the case when the buffer is not full, `TBBuffer` will first lock the `Consumer` (3) because of the atomicity requirement between the delivery of the message `put` and the change of `Consumer`'s expectation queue. Then the `put` message is delivered at the `Producer` and the `Consumer`'s expectation queue is modified (4). Another message `get` arrives at the `Consumer` (6). If the `Consumer`'s local time doesn't satisfy the expectation, there is no need to communicate with RTsynchronizer about the validity of delivering the message, instead it is directly put back to `Consumer`'s mail queue (7). Other unguided messages (8), however, can be delivered at `Consumer`.

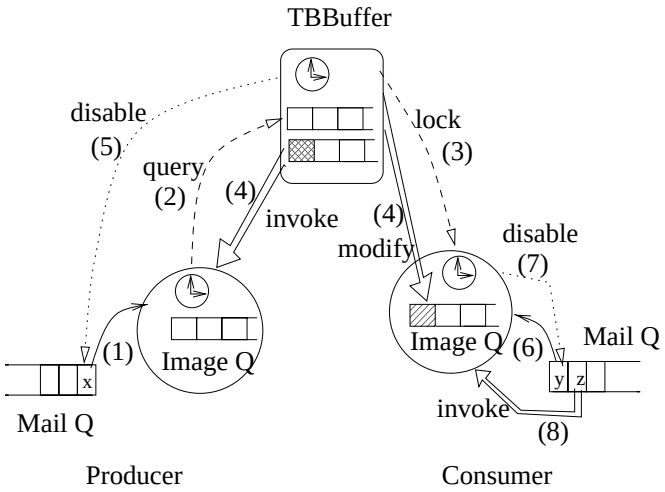


Figure 4.8: The Detailed Steps of Delivering a Message at a Time-Bounded Buffer

4.2.4 Terminating the Operation of an RTsynchronizer

RTsynchronizers are dynamic objects in that their functionalities may be turned off by their creators. Upon observing that a terminating message is delivered at its creator, the RTsynchronizer may take either a *relaxed* or *stringent* approach to end its operation. If the RTsynchronizer adapts the relaxed approach, previously generated expectation instances are flushed out and the controlled actors are set free — unprocessed messages are processed by those previously controlled actors at their will. In contrast, with the stringent approach, the terminating RTsynchronizer will still be operating until all the previously expected constraint instances have been handled, or all the expectation queues are empty.

Which approach is superior depends on specific applications. We take the relaxed strategy in our implementation for its simplicity.

4.2.5 Scheduling Strategy

As we know that the information available to scheduler is static information — time constrained computation graph, generated by compiler and the pre-processing procedure, and run-time information — expectation instances. Unfortunately, it is NP-hard to give a complete schedule based on a set of constraints. Our strategy is to use both static and dynamic information at run time and reconcile the NP-hardness and possibility of violated time constraints by bookkeeping guided messages and expected message instances as specified in RTsynchronizers, and making a '*best*' partial schedule and validating the schedule before sending jobs to process units. In other words, the user source code level specification RTsynchronizers, act as guides to a scheduler, and the scheduler in turn notifies the RTsynchronizers of message deliveries. The innovation of our approach is that the static specification is dynamically used in adaptive scheduling as in Figure 4.9.

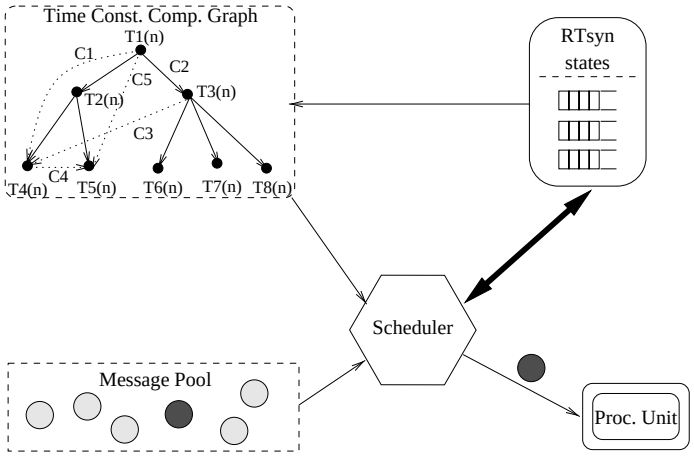


Figure 4.9: Adaptive Scheduling

The scheduler first checks the expectation queues in each RTsynchronizer and (for simplicity) uses the earliest deadline first scheduling policy to *locate* the next message for execution; when all the expectation queues are empty, the scheduler runs the static search algorithm on the

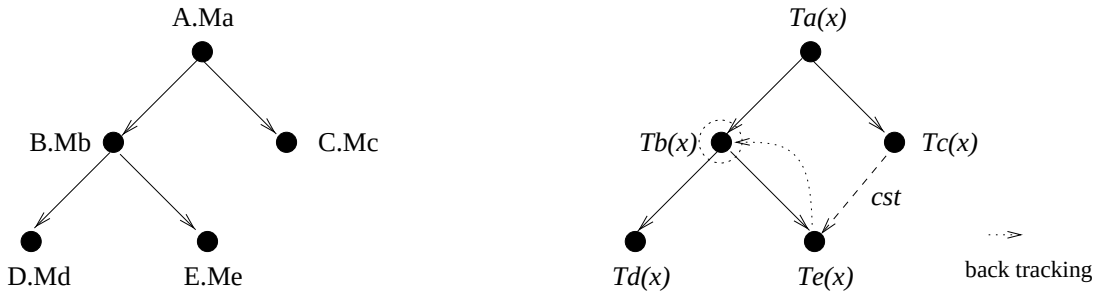
time constrained computation graph for a fixed amount of time to generate the best schedule it can find during that period of time. *Safe progress, unsafe wait* principle is applied in such scheduling: if delivering such a message will not cause a violation of any constraints in the future (*safe*), the message is delivered; otherwise (i.e. *unsafe*), the message firing is postponed until further information is available. More specifically, given a constraint `a.m1 : b.m2 [d1, d2]`, schedule `m1` while `m2` is not available or it is impossible to schedule `m2` in the time frame of `[d1, d2]` relative to the time when `m1` is scheduled to be delivered would be hazardous to the constraints; postponing scheduling `m1` is more appropriate. On the other hand, if `m1` is available and it is possible to schedule `m2` `d1` time after and `d2` time before `m1`, `m1` will be scheduled.

However, even such a schedule may not be correct because of the RTsynchronizer state change, the message destination objects need to query their controlling RTsynchronizers before delivering the constrained messages as shown in the time bounded buffer example Figure 4.8 (2). Delivery of a constrained message is reported to corresponding controlling RTsynchronizers.RTsynchronizers state changes caused by this delivery, if any, are atomic and synchronized with the message delivery.

Notice that our constraint time intervals on messages are relative to the delivery of `triggering` messages. By cleaning out the expected effects before initiating new triggering messages, we decrease the risk of violating timing constraints. In addition, this limits the size of the expectation queue and thus decreases the overhead of searching for the earliest deadline among the expectation instances.

Nevertheless, locating the next message according to the expectation queues could be a complicated process when an expected message has not arrived in the system. It may need to back-track in the time constrained computation graph.

Consider the computation tree in Figure 4.10 (a). Method `Ma` of object `A` sends a message to object `B` at method `Mb` and a message to object `C` at method `Mc`. Method `Mb` of object `B` in turn sends out messages to object `D` at `Md` and `E` at `Me`; and a time constraint: `C.Mc : E.Me [d1, d2]`. After pre-processing, we derive a time constrained computation graph as in Figure 4.10 (b).



(a) Computation Tree (b) Time Constrained Computation Graph

Figure 4.10: Locating the Next Message for Execution

Assume that at the current time, only two messages have been delivered, namely, `A.Ma` and `C.Mc`, and the message of `C.Mc` was delivered at time `t`. Hence, by the constraint specification, we expect to deliver a message of `E.Me` during time `[d1 + t, d2 + t]`. Clearly, `d2 + t` is the earliest deadline among all the expectation instances at the current time. Thus, the scheduler

is trying to dispatch a message of `E.Me` to a process unit. Unfortunately, the message of `E.Me` is sent by a delivery of message `B.Mb`, and that message has not be scheduled. Therefore, in order to satisfy the expectation instance, the scheduler has to back-track the computation path and check the availability of the causal message and dispatch the first available one. In our example, because of previous delivery of message of `A.Ma`, the message of `B.Mb` which is the causal message of our expectation instance `E.Me` is available, and should be scheduled with a deadline `(d2 + t) - Tb(x)`.

The back tracking problem is caused by RTsynchronizers lack of information about the underlying computation. One way to solve the problem is to not only to update the RTsynchronizer's expectation queues upon delivery of any guided messages, but also update the time constrained computation graph and map the `trigger` — `effect` time constraints into end-to-end time constraints on the time constrained computation graph. The scheduler instead of searching the expectation queues for the expected instance with the earliest deadline, searches the time constrained computation graph. This approach may provide a better schedule, but is computationally more expensive because of the mapping process for every delivery of guided messages and graph searching. Our approach avoids unnecessary end-to-end constraint mapping and the search for the expected instance with earliest deadline is based on linear queues, and hence simpler. However, this approach could miss some of the end-to-end deadlines. It again reflects the canonical trade-off in a scheduling problem — time given to scheduler and the schedule quality returned by the scheduler.

4.3 Conclusion

We have described a distributed implementation framework for real-time systems and discussed some major issues and trade-offs encountered in the implementation. In summary, our approach is conservative in that it seeks to avoid conflict extremes. We assume the underlying architecture is an asynchronous distributed system, however, we assume the wall clock on each computation unit is synchronized. We use a pre-process procedure to detect infeasible time constraints, but we do not explore the whole computation tree to statically generate a complete schedule off-line. Guided messages are checked for validation before they are delivered; however, the scheduler does not blindly pick up a message from the message queue and treat RTsynchronizers as passive monitors; instead, the scheduler communicates with RTsynchronizers to achieve adaptive scheduling. Each individual RTsynchronizer is partially distributed (controlled objects share some of the RTsynchronizers state information), and is also centralized — it is a single entity when communicating with schedulers.

Chapter 5

Conclusion

We have observed that in distributed systems, the computational behavior of individual objects and the cooperation behavior among these objects are two different concerns. For instance, in distributed real-time systems, *how* an object performs its computation and *what* an object does are orthogonal to *when* the object does it. These two separate concerns should be captured by separate language constructs in order to achieve better modularity and code reusability. We have developed a new language concept and corresponding language construct RTsynchronizer to support such separation in distributed real-time computing.

RTsynchronizers support the description of timing constraints in absolute and relative time for a group of objects. As special objects, RTsynchronizers do not interfere with the computational aspects of the underlying computation. However, the existence of such objects in a computing system ensures that the system has correct behavior in the time domain, hence satisfies the criteria for real-time computing — "a system which has to respond to externally-generated input stimuli within an *finite* and *specified* period".

RTsynchronizers express the timing constraints not as part of computational objects, but as independent entities. The entities provide an environment in which autonomous computational objects are constrained and forced to synchronize with each other in real time.

Separating real-time aspects from computational aspects of a real-time system, both conceptually and 'physically', i.e. using distinct objects — RTsynchronizer, provides a high level modularity, and makes it easier and less error prone to design and maintain distributed real-time applications.

In addition, RTsynchronizers are transparent to computational objects, and their functionalities only depend on the interfaces, not the representations, of computational objects. Such properties make it possible to achieve code reusability in two directions: the same group of computational objects can be constrained by different RTsynchronizers in different applications — computational code reuse; the same RTsynchronizer can apply to different group of computational objects as long as they have the same interfaces — constraint specification reuse.

We have developed an operational semantics for the systems with both computation objects and RTsynchronizers. The operational semantics is represented by a set of transition rules. Fro a programming language point of view, by abstracting two different properties into two different types of objects, we have achieved better modularity and code reusability. From a semantics point of view, by distinguishing the underlying computation and the constraint environment the computation is carried in, the computation model is more clean and intuitive.

In particular, in our transition rules, we separate *when* a transition is allowed to apply from *how* to choose a 'best' transition rule from a set of possible transitions. For instance, when more than one message is available and satisfies the constraints, which message will be invoked first is left out in our semantics. This further separates the semantics of ART systems from the scheduling mechanism and detailed implementation strategies.

We have also developed an algorithm for detecting infeasible declarative timing specifications at the source code level. Such detecting not only provides system designers early feedback about their specifications, but also decreases the time taken by a scheduler to combine computational code with timing requirements to generate a correct schedule.

Further, we have described a distributed implementation framework for real-time systems and discussed some major issues and trade-offs encountered in the implementation. In summary, our approach is moderate in that it seeks to avoid conflict extremes. We assume the underlying architecture is an asynchronous distributed system, however, we assume the wall clock on each computation unit is synchronized. We use a pre-process procedure to detect infeasible time constraints, but we do not explore the whole computation tree to statically generate a complete schedule off-line. Guided messages are checked for validation before they are invoked; however, the scheduler does not pick up an arbitrary message from the message queue and only takes RT-synchronizers as passive monitors; instead, the scheduler communicates with RTsynchronizers to do adaptive scheduling. Each individual RTsynchronizer is partially distributed (controlled objects share some of the RTsynchronizers state information) and is also centralized —- it is single intact entity when communicating with schedulers.

In summary, this research has presented a modular linguistic approach, which can ease the design of real-time distributed systems, increase the modularity of programs and reusability of general-purpose code, and increase the analyzable and hence the reliability of distributed real-time systems.

However, this research does not put an end to problems in the distributed real-time computing area. Further research based on our language paradigm is needed, such as the semantic definition of simultaneous events in a quantitative time domain. In a qualitative time domain, i.e. ordinal time domain, "simultaneous" events in distributed systems are well defined — no intermediate observable state between two simultaneous events; however, in a quantitative time domain, such a definition is not sufficient. One reason is that each event is associated with a quantitative time point in cardinal time domain, and if we assume the same definition for "simultaneous" events as in the ordinal time domain, some undesired phenomenon may happen: time may progress backwards.

Currently, we assume that for each ART system configuration, there is only one time which serves as the global time and we require that two different configurations must agree on the "current time" before they can be composed. The composition between ART configurations with different timer needs to be further studied.

In addition, building a complete programming environment for RTsynchronizers is another important direction for future work.

Bibliography

[1] Ada95 6.0 Reference Manual. Technical report, Internet's Public Ada Library, 1995.

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

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

[4] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Proceedings of the 3rd IFIP Working Conference on Dependable Computing for Critical Applications*, September 1992.

[5] G. Agha, I. Mason, S. Smith, and C. Talcott. *Funcations of Open Distributed Systems*. MIT Press, 1997. to be published.

[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. 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.

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

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*, chapter 10. Addison-Wesley, 1988.

[10] M. Akşit 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.

[11] M. Akşit, 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.

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

[13] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.

[14] Inc. American National Standards Institute. The Programming Language Ada Reference Manual. In *The Programming Language Ada Reference Manual*. Springer Verlag, 1983.

[15] J. Armstrong, R. Virding, and M. Williams. *Cocurrent Programming in Erlang*. Prentice hall, 1993.

[16] J.L. Armstrong, B.O. Däcker, S.R. Virding, and M.C. Williams. Implementing a Functional Language for Highly Parallel Real-Time Applications. In *International Conference on Software Engineerings for Telcommunication Systems and Services*, pages 157 — 163. IEEE, March 1992.

[17] J.W.de Bakker, C. Huizing, W.P.de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice*, Mook, The Netherlands, June 1991. Springer-Verlag. Lecture Notes in Computer Science 600.

[18] F. Baude and G. Vidal-Naquet. Actors as a Parallel Programming Model. In C. Choffrut and M.Jantzen, editors, *Proceedings of STACS 91*, volume 480 of *LNCS*, pages 184–194. Springer-Verlag, 1991.

[19] A. A. Berlin. Compiling Scientific Code Using Partial Evaluation. *IEEE Computer Magazine*, 23(12):25–37, Dec. 1990.

[20] G. Berry, P.Couronné, and G. Gonithier. Synchronous Programming of Reactive Systems: An Introduction to ESTEREL. In *Programming of Future Generation Computers*, pages 33–55. Elsivier Science Publishers, Amsterdam, 1988.

[21] Gerard Berry and laurent Cosserat. The ESTEREL Synchronous Programming Language and its mathematical Semantics. In *Seminar on Concurrency*, pages 389–448. Springer Verlag, 1984. LNCS 197.

[22] E. Brorsson, C. Eriksson, and J. Gustafsson. RealTimeTalk An Object-Oriented Language For Hard Real-Time Systems.

[23] Alan Burns and Andy Wellings. *Real-Time Systems and Their Programming Languages*. Addisoin Wesley, 1989.

[24] W. Clinger. Foundation of Actor Semantics. Technical Report AI-TR-633, MIT, 1981.

[25] M. Colnarič, D. Verber, and W. A. Halang. Supporitng High Integrity and Behavioural Predictability of Hard Real-Time Systems. *Informatica — An International Journal of Computing and Informatics*, 19(1):59 – 70, Feb 1995.

[26] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 25, pages 515–550. MIT Press, 1990.

[27] Z. Deng, J.W.-S Liu, and J. Sun. Dynamic Scheduling of Hard Real-Time Applications in Open System Environment. Technical Report UIUCDCS-R-96-1981, University of Illinois at Urbana-Champaign, October 1996.

[28] K. Erciyes, Ö. ÖZkasap, and N.Aktas. A Semi-Distributed Load Balancing Model for Parallel Real-Time Systems. *Informatica — An International Journal of Computing and Informatics*, 19(1):83 – 97, Feb. 1995.

[29] R. Gerber et al. Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes. In *Real-Time Systems Symposium, 1994*, 1994.

[30] Antony A. Faustini and Edgar B. Lewis. Toward a Real-Time Dataflow Language. *IEEE Software*, 3(1):139 — 145, January 1986.

[31] S. Frølund and G. Agha. A Language Framework for Multi-Object Coordination. In *Proceedings of ECOOP 1993*, volume 707 of *LNCS*. Springer Verlag, 1993.

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

[33] R. Gerber and S. Hong. Compiling Real-Time Programs with Timing Constraint Refinement and Structural Code Motion. *IEEE Transaction on Software Engineering*, 21(5), May 1995.

[34] R. Gerber and I. Lee. Communicating Shared Resources: A Model for Distributed Real-Time Systems. In *Proceedings of Real-Time System Symposium*, pages 68–78, 1989.

[35] R. Gerber and Insup Lee. A Layered Approach to Automating the Varification of Real-Time Systems. *IEEE Transactions on Software Engineering*, 18(9):768–784, September 92.

[36] A. Goswami and M. Joseph. Defining Time Domains for Computation. In H.S.M. Zeda, editor, *Real-Time Systems, Theory and Applications.* North-Holland, 1990.

[37] J.N. Gray. Notes on Database Operating Systems. In R. Bayer, R.M. Graham, and G. Seegmuller, editors, *Operating Systems — An Advanced Course*, volume 66, pages 393–481. Springer-Verlag, 1978. Lecture Notes in Computer Science.

[38] A. S. Grimshaw, A. Silberman, and J. .W. S. Liu. Real-Time Mentat at Programming Language and Architecture. In *IEEE Globecom'89*, 1989.

[39] W. A. Halang and K. Mangold. Real-Time Programming Languages. In M. Schiebe and S. Pferrer, editors, *Real-Time Systems Engineering and Applications*, chapter 6, pages 141–200. Kluwer Academic, 1992.

[40] D.K. Hammer, E.J. Luit, and O.S. van Roosmalen. DEDOS, A Distributed Real-Time Environment. *IEEE Parallel and Distributed Technology*, 2(4):32 — 41, 1994.

[41] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[42] B. Hoogeboom and W. A. Halang. The Concept of Time in the Specification of Real-Time Systems. In M. Schiebe and S. Pferrer, editors, *Real-Time Systems Engineering and Applications*, chapter 2, pages 11–40. Kluwer Academic, 1992.

[43] 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.

[44] Farnam Jahanian, Aloysius K. Mok, and Douglas A. Stuart. Formal Specification of Real-Time Systems. Technical Report 25, The University of Texas at Austin, 1988.

[45] M. Joseph and A. Goswami. Relating Computation and Time. Technical report, Department of Computer Science, University of Warwick, Coventry, 1988.

[46] S. T. Keane. Realization of An Ada Architecture for Heterogeneous Distributed Embedded Processors. In *13th AIAA/IEEE Digital Avionics Systems Conference*, 1994.

[47] Kevin B. Kenny and Kwei-Jay Lin. Building Flexible Real-Time Systems Using the Flex Language. *IEEE Computer*, 24(5):70 — 78, 1991.

[48] E. Kligerman and A. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9):941 — 949, September 1986.

[49] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[50] B.W. Lampson. Atomic Transactions. In B.W. Lampson, H.Siegert, and M. Paul, editors, *Distributed Systems — Architecture and Implementation*, volume 105, pages 246–265. Springer-Verlag, 1981. Lecture Notes in Computer Science.

[51] F. Laroussinie. From Timed Automata to Logic and Back. Technical Report RS-95-2, Basic Research in Computer Science, January 1995.

[52] R. Lauber. Foreward. In M. Schiebe and S. Pferrer, editors, *Real-time Systems Engineering and Applications*, chapter Forward. Kluwer Academic, 1992.

[53] J. Lin, D.C. Kung, and P. Hsia. An Object-Oriented Approach for Modeling and Analysis of Safety-Critical Real-Tiem Systems. *Informatica — An International Journal of Computing and Informatics*, 19(1):43 – 58, Feb. 1995.

[54] K. Lin and J. W. Liu. Flex: A language for Real-Time Systems Programming. Technical Report 1634, UIUC, 1990.

[55] Barbara Liskov. Practical Uses of Synchronized Clocks in Distributed Systems. In R. Ladner, editor, *Proceedings of the Tenth Annual ACM symposium on Principles of Distributed Computing*, 1 – 9, pages 1–9, Montreal, Canada, August 19 – 21 1991. ACM, ACM Press.

[56] Leo YuHsiang Liu and R.K. Shyamasundar. RT-CDL: A Distributed Real-Time Design Language and Its Operational Semantics. *Comput. Lang.*, 20(1):1 — 23, 1994.

[57] David Luckhman. Guide to the Rapide 1.0 Langauage Reference Manuals. Technical report, Stanford University, http://anna.stanford.edu/rapide/rapide.html, 1996.

[58] C.W. McKay and C. Atkinson. Supporting the Evolution of Distributed, Non-stop, Mission and Safety Critical Systems. *Informatica — An International Journal of Computing and Informatics*, 19(1):7 – 24, Feb. 1995.

[59] R. Milner. *Communication and Concurrency*. Prentice Hall International (UK), 1989.

[60] A. K. Mok. Real-Time Logic, Programming and Scheduling. In B. Randell, editor, *Real-Time Systems, Proceedings of the Joint University of Newcastle Upon Tyne/International computers Limited Semina*, Newcastle, September 1989.

[61] A. K. Mok. Real-Time Scheduling and the Boeing 777. In *European Educational Forum School on Embedded Systems 1996*, Veldhoven, NL, November 1996.

[62] Sape Mullender. *Distributed Systems*, chapter 5. Addison-Wesley, second edition, 1993.

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

[64] B. Nielsen and G. Agha. Semantics for an Actor-Based Real-Time Language. In *Proceedings of the Fourth International Workshop on Parallel and Distributed Real-Time Systems*, Honolulu, April 1996.

[65] O. Nierstrasz and M. Papathomas. Viewing Objects as Patterns of Communicating Agents. In *Proceedings OOPSLA/ECOOP '90*, pages 38–43, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[66] V. Nirkhe and W. Pugh. Application of Partial Evaluation to Hard Real-Time Programming. In W.A. Halang and K.Ramamritham, editors, *Proceedings of the IFAC/IFIP Workshop*, number 1 in IFAC Workshop Series, pages 71–77. Pergamon Press, 1992.

[67] N. Oxhoj, J. Palsberg, and M. I. Schwartzbach. Making Type Inference Practical. In *Proceedings of ECOOP'92*, pages 329–349. Springer-Verlag (LNCS 615), 1992.

[68] Carlos Puchol. A Solution to the Generalized Railroad Crossing Problem in ESTEREL. Technical Report 05, The University of Texas at Austin, 1995.

[69] S. Ren and G. Agha. *RTsynchronizers*: Language Support for Real-Time Specifications in Distributed Systems A Language Framework for Multi-Object Coordination. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 18-21 1995.

[70] S. Ren and G. Agha. *RTsynchronizers*: Language Support for Real-Time Specifications in Distributed Systems. *ACM SIGPLAN Notices*, 30(11):50 – 59, November 1995.

[71] S. Ren, G. Agha, and M. Saito. A Modular Approach for Programming Distributed Real-Time Systems. *Journal of Parallel and Distributed Computing*, 36(1):4–12, July 1996.

[72] Onno S. Van Roosmalen. DEAL: An Object-Oriented Language for Distributed Real-Time Systems. In *International Workshop on Object-Oriented Real-Time Depandable Systems (WORDS)*, Dana Point, CA, USA, October 1994. IEEE.

[73] M. Saito and G. Agha. A Modular Approach to Real-Time Synchronization. *OOPS MESSENGER*, 1(1):13 – 20, Jan 1996.

[74] I. Satoh and M. Tokoro. Semantics for a real-Time Object-Oriented Programming Language. In *Int. Conf. on Computer Languages*, Toulous, France, 1994. IEEE.

[75] M. Schiebe. The Origins of Real-Time Processing. In M. Schiebe and S. Pferrer, editors, *Real-Time Systems Engineering and Applications*, chapter 1, pages 1 – 10. Kluwer Academic, 1992.

[76] M. Schiebe and S. Pferrer, editors. *Real-Time Systems Engineering and Applications*. Kluwer Academic, 1992.

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

[78] Karsten Schwan and Hongyi Zhou. Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads. *IEEE Transactions on Software Engineering*, 18(8):736 – 747, 1992.

[79] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. Wiley Professional Computing, 1994.

[80] A. Silberman. Real-Time Mentat. Technical report, Department of Computer Science, UIUC, 1995.

[81] P.K. Sinha. *Distributed Operating Systems — Concepts and Design*. IEEE Computer Society Press, 1997.

[82] J.A. Stancovic. Misconceptions about Real-Rime Computing: a Serious Problem for the Next Generation. *IEEE Computer*, 21(10):10 – 19, 1988.

[83] J.A. Stankovic. A Perspective on Distributed Computer Systems. *IEEE Transactions on Computers*, C-33(12):1102 — 1115, 1984.

[84] A. Stoyenko, V. Hamacher, and R. Holt. Analyzing Hard-Real-Time Programs for Guaranteed Schedulability. *IEEE Transactions on Software Engineering*, 17(8):737 – 750, August 1991.

[85] 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.

[86] 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.

[87] J.D. Ullman. NP-Complete Scheduling Problems. *Journal of Computer System Science*, 10:384–393, 1975.

[88] Jan van Katwijk and Hans Toetenel. Loose Specification of Real Time Systems. *Informatica — An International Journal of Computing and Informatics*, 19(1):25 – 42, Feb. 1995.

[89] R.A. Volz and T.N. Mudge. Timing Issues in the Distributed Execution of Ada Programs. *IEEE Transactions on Computers*, 36(4):449 — 459, April 1987.

[90] A. Vrchoticky. Compilation Support for Fine-Grained Execution Time Analysis. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, Orlando, Florida, June 1994.

[91] G. J. Whitrow. *The Natural Philosophy of Time*. Oxford: Clarendon Press, 1980.

[92] Claes Wikström. Implementing Distributed Real-Time Control Systems in a Functional Programming Language. In *Software Engineering for Telecommunication Switching Systems*, Florence, Italy, March 1992.

[93] Claes Wikström. Distributed Programming in Erlang. In *Frist International Symposium on Parallel Symbolic Computation*, 1994.

[94] T. Winograd. From Programming Environments to Environments for Designing. *Communications of the ACM*, 38(6):65–74, June 1995.

[95] S. J. Young. *Real-Time Languages: Design and Development.* Ellis Horwood, 1982.

[96] M. Younis, T. Marlowe, and A. Stoyenko. Compiler Transformations for Speculative Execution in a Real-Time System. In *Proceedings of the* 15*th Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[97] G. Yu and L.R. Welch. A Novel Approach to Off-line Scheduling in Real-Time Systems. *Informatica — An International Journal of Computing and Informatics*, 19(1):71 – 82, Feb. 1995.

[98] H.S.M. Zedan, editor. *Real-Time Systems, Theory and Applications.* North-Holland, 1990.

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

Vita

Shangping Ren was born on December 12, 1963 at Maanshan, a small town in Anhui Province in China.

In the fall of 1981, Shangping enrolled in Hefei Polytechnic University where she pursued a major in Information and Computer Engineering and received her Bachelor of Engineering Degree from the Hefei Polytechnic University in June 1985. Afterwards, she continued her graduate study in the same department and concentrated on research in the area of Artificial Intelligence developing a mini-expert system for teaching Symbolic Integration. She received her Master of Engineering Degree in June 1988.

From August 1988 to August 1990, Shangping worked as an Assistant Professor in Computer Science Department at Anhui University.

In August 1990, Shangping first enrolled in the Ph.D. program in the Department of Curriculum and Instruction at the University of Illinois at Urbana-Champaign. One and one half years later, in January 1992, she transferred to the Department of Computer Science at the University of Illinois at Urbana-Champaign. During her tenure in the Department of Computer Science, Shangping served as a Research Assistant in the Open Systems Laboratory under the direction of Dr. Gul Agha and has also served as a Teaching Assistant in the Department of Computer Science.

AN ACTOR-BASED FRAMEWORK FOR REAL-TIME COORDINATION

Shangping Ren, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1997
Dr. Gul A. Agha, Advisor

Programming language research should be driven by the needs of specific application domains, such as real-time embedded systems, multimedia, distributed data-base management applications, etc. For instance, most often large real-time applications involve distributed and concurrently but asynchronously operating devices. The correctness of the integrated systems depends not only on the correct operation of each individual device, but also on the correct coordination among these devices. Furthermore, the correctness of device operation and cooperation requires that both the underlying logical computation be correct and the computation satisfy some notion of quantitative timing requirements.

This thesis is a case study for how a language should be designed to suit a specific application domain. Specifically, I have designed RTsynchronizers (Real Time Synchronizers), a high-level programming language abstraction for specifying real-time coordination constraints between objects in a distributed open system. RTsynchronizers enforce timing constraints on a group of computational objects. They are defined in terms of the interface of members in the group; thus they are independent of the representations of computational objects. During program execution, RTsynchronizers affect the scheduling of distributed objects to enforce real-time relations between events. With RTsynchronizers, we achieve the ability to separately specify and program functional components and the coordination constraints among these components, and the ability to reuse the computational code as well as coordination constraint code.

This research uses both theoretical and experimental methods. Specifically, an operational semantics based on RTsynchronizers and regular computational objects (modeled as actors) is defined. RTsynchronizers are implemented as C++ objects which are used by system schedulers in deciding the appropriate messages to be invoked at message destination computational objects. The implementation demonstrates the feasibility of making the existence of RTsynchro-

nizers transparent to computational objects. Furthermore, an algorithm for detecting infeasible real-time coordination constraints at source code level is developed.