Semantics for an Actor-Based Real-Time Language *

Brian Nielsen Gul Agha
Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801, USA

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

Abstract

We give formal semantics for a distributed concurrent object-oriented real-time programming language based on a variant of the actor model which includes an extention enabling the specification of time constraints on message-invocation. Real-time semantics must capture both the qualitative and quantitative aspects of the language, and provide a meaning for the real-time constructs. The real-time semantics of our language is given as timed graph, an existing real-time specification formalism. We present the semantics by first defining an operational semantics for the untimed language, and then translating this into a timed graph which interprets the time constructs. Our semantics is formulated independently of the underlying resources needed to execute a program; the semantics of a program thus defines the set of permissible concrete implementations.

1 Introduction

Semantics for concurrent programming languages usually focus on qualitative aspects—e.g., defining what constitutes a legal execution of a program and abstracting over the quantitative aspects of the program execution. This is unsuitable for real-time programming languages: the correctness of real-time systems depends on *what* actions are performed as well as *when* they are performed.

This paper demonstrates how such quantitative aspects can be captured and described formally. Specifically, we define semantics for a simple distributed object-based real-time language based on a variant of the actor model which includes an extention enabling the specification of time constraints on message-invocations. Our work will set the foundation for semantics for a considerably more complex language [12].

Our primary objective is to formulate the semantics of a real-time program independently of the resources (number of CPUs, network topology, speed, etc.) needed to execute it. Although the real-time behavior of a program is highly dependent on execution resources, a semantics including such information tends to be a formal model of a concrete implementation, rather than a specification for a set of possible implementations. In our semantics a real-time program is viewed as a (loose) specification for actual implementations. The semantics define what actions must take place as well as *when* they are permissible or required. It remains to be shown that a given implementation is satisfactory for the real-time program: i.e., the implementation refines the specification. Thus, reasoning about an implementation requires an explicit model of the specific resources that are available for the program's execution.

During design activities an overall system-wide specification is (possible through a series of stepwise refinements) broken down into a set of components and per-component time constraints. The finished design is expressed as a program in a real-time programming language. At this level, it must be validated that the program satisfies the overall system requirements. Then, the program is realized in a concrete implementation by choosing the environment and resources needed to execute the program. At this level it must be validated that the implementation satisfies the time-constraints expressed in the program. Formal semantics is a necessary condition for performing these validations (possibly mechanically).

We define the semantics in two steps. First, we define the operational semantics for an untimed actor language. We then extend our language with timing constraints and provide its semantics in terms of an existing real-time specification formalism, timed graphs.

*The research has been made possible by support from the Office of Naval Research (N00014-93-1-0273), by the National Science Foundation (NSF CCR 93-12495), and by grants from The Danish Technical Research Council and the Danish Research Academy. The author's would also like to acknowledge helpful comments and criticisms from Arne Skou, and from Dan Sturman, Shangping Ren, and other members of the Open Systems Laboratory.

2 The Actor Model

Actors [1, 2, 4] is a model for distributed concurrent computing systems. An actor system is comprised of autonomous objects, called actors, that communicate using asynchronous message passing. Messages that have been sent but not yet received are conceptually queued up in the receiver actor's mailbox. The receiver eventually removes the message and processes it. An actor encapsulates a state that can be accessed and modified from the outside only by sending it messages.

Each actor has a mail-address used by other actors to send it messages; an actor is able to send messages to actors whose addresses it knows. Moreover, these addresses may be communicated in messages thus allowing for dynamic configuration of the communication topology. An actor is inactive unless someone (including possibly the actor itself) sends it a message to process. The invocation of an actor by a message is called an *event*. The event that led to the message being sent is called it's *cause*.

As a response to a message a (thereby busy) actor may:

Compute and change state: When a message is invoked on an actor, it executes its deterministic behavior. This consists of computing expressions, changing state by assigning to state variables, sending messages or creating new actors. An actor processes one message at a time, i.e., actors are single threaded. After processing a message it is ready to accept the next (queued) message. (In [1, 2], an explicit *become*-primitive provides a restricted form of multithreading that is possible to represent in our semantics).

Send a message: The primitive **send**(a, cv) sends a message with communication value cv asynchronously to the actor with the address a.

Create new actors: The **newActor**(b, cv) primitive creates a new actor with behavior b and initialization value cv. **newActor** returns the address of the newly created actor. For brevity, the semantics of dynamic actor creation will be omitted in this paper.

Actors can model interaction with an external environment. We assume that the environment is itself modeled as an actor system capable of sending and receiving messages. The actors that are part of the environment are called *external* actors, and the actors within the system that are able to receive messages from the environment are called *receptionists*. Two actor systems can be composed to form a combined system.

To illustrate our approach, we describe a small real-time language based on the actor-model which supports constraints on message invocation. Specifically, the send-primitive has been extended to include timing information:

send$(a, cv)\langle r; d\rangle$

where d and r are positive real-valued constants that specify, respectively, the earliest and latest time when the message can be invoked. We call r the *release time* and d the *deadline*. Constraints on message execution are specified *relative* to event that caused it: because the execution of actors is asynchronous, the time the message is sent is not relevant. Thus, activation time is bounded if all hereditarily related causal events have a bound or, as we will discuss later, if the causing event is external. Further, the programmer can specify bounds on the amount of time an actor may use to process a message (computation time). This bound is syntactically stated as part of method declarations.

The language is illustrated by the actor program in Figure 1. A controller requests a pressure-sensor for its value. The sensor sends a message containing the value back to its customer (the controller). If the received value exceeds a certain critical value the controller opens a safety pressure valve. By sending itself a message, the controller periodically (plus/minus some error) re-checks the sensor value. Note that, the method-name to be invoked is, by convention, encoded as the first field of the communicated value.

```
actor pressureSensor () {
   real value;
   method read(actorAddr customer) ⟨1⟩{
      send(customer,(reading,value))⟨0; 2⟩;
   }
}
actor controller (actorAddr sensor,valve) {
   method readSensor() ⟨2⟩ {
      send(sensor,(read,self))⟨0; 3⟩;
      send(self,readSensor)⟨P − e; P + e⟩;
   }
   method reading(real value)⟨1⟩ {
      if(value ≥ critical) send(valve,open)⟨0; 2⟩
   };
}
```

Figure 1: Actor Program

3 Untimed Actor Semantics

This section gives the operational semantics for the actor-language without time. Operational semantics define how the state of the system changes when a primitive operation is performed, thus giving an abstract interpretation of the language. The actor semantics presented here is inspired by the work of [4].

The state of an actor system is represented by a configuration. A configuration can be thought of as an instantaneous snapshot of the system state made by a conceptual observer. It is modeled as a four-tuple $\left\langle\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho}$ where α represents actor-states, μ is a set of messages, and ρ and χ are sets of actor names. ρ is the set of receptionists, i.e., actors whose names have been exported to the environment. χ are the external actors known to this system.

The α mapping maintains the state of all actors in the system. Given the actor-name, α returns its state. An actor state holds the following information about an actor: execution state (busy or inactive), the values of its state-variables, and how far the actor has come in its computation. The state of a busy actor state is written $[E \vdash R \sqsubset e \sqsupset]_a$ (square brackets is the symbol of a busy actor). a is the address of the actor having this state. E is an environment (mapping from identifiers to their values) that keeps track of the values of the state-variables. $R \sqsubset e \sqsupset$ is a reduction context with a hole filled with subexpression e. Intuitively, e is the sub-expression to be reduced next when the actor performs a computation step. R is the remainder of the actor's behavior. An inactive actor is written $(E \vdash b)_a$. Here actor a is waiting for an incoming message. When a message arrives the actor's behavior b is applied to the incoming message and the actor becomes busy.

The messages sent but not yet received are represented by a set μ in the configuration. A message is a tuple $\langle a \overset{r,d}{\Leftarrow} cv\rangle$ consisting of a destination actor-address a, a value to be communicated cv, deadline d, and release time r. This timing information is ignored in the untimed case.

In the operational semantics, it is necessary to distinguish one or more actors from the set of actor states to indicate which actors are being changed by the transition. We call this the *focus* actor(s). The notation: $\alpha\,, as$ picks the focus actor, where as ranges over actor states. Focus messages are treated similarly. Each rule is given a label consisting of a tag indicating the primitive operation used, the name of the focus actor, and additional parameters. This label uniquely identifies which system component has performed which primitive operation. The operational semantics is given by Definition 1.

The **fun** transition defines the effect on system state when an actor performs a computation step (reduction of an expression). An actor can reduce expression e to e' if the reduction rules ($\rightarrow_\lambda$) of the behavior allows it. The transition system $\rightarrow_\lambda$ defines the semantics of the sequential language used to express actor behaviors. Since we do not rely on a specific language, we have omitted its definition.

Definition 1 *Configuration transitions* $\rightarrow_\kappa$

$\langle$**fun** : $a\rangle$

$$\frac{E \vdash R \sqsubset e \sqsupset \rightarrow_\lambda \; E' \vdash R \sqsubset e' \sqsupset}{\left\langle\!\left\langle\, \alpha\,, [E \vdash R \sqsubset e \sqsupset]_a \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho} \rightarrow_\kappa \left\langle\!\left\langle\, \alpha\,, [E' \vdash R \sqsubset e' \sqsupset]_a \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho}}$$

$\langle$**term** : $a\rangle$

$$\left\langle\!\left\langle\, \alpha\,, [E \vdash \sqsubset]_a \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho} \rightarrow_\kappa \left\langle\!\left\langle\, \alpha\,, (E \vdash b)_a \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho}$$

$\langle$**snd** : $a, \langle a' \overset{r,d}{\Leftarrow} cv\rangle\rangle$

$$\left\langle\!\left\langle\, \alpha\,, [E \vdash R \sqsubset \mathbf{send}(a', cv)\langle r; d\rangle \sqsupset]_a\} \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho} \rightarrow_\kappa \left\langle\!\left\langle\, \alpha\,, [E \vdash R \sqsubset \mathbf{nil} \sqsupset]_a \mid \mu\,, \langle a' \overset{r,d}{\Leftarrow} cv\rangle \,\right\rangle\!\right\rangle_{\chi}^{\rho}$$

$\langle$**rcv** : $a, \langle a \overset{r,d}{\Leftarrow} cv\rangle\rangle$

$$\left\langle\!\left\langle\, \alpha\,, (E \vdash b)_a \mid \mu\,, \langle a \overset{r,d}{\Leftarrow} cv\rangle \,\right\rangle\!\right\rangle_{\chi}^{\rho} \rightarrow_\kappa \left\langle\!\left\langle\, \alpha\,, [E[FV(b) \mapsto cv] \vdash b \sqsubset]_a \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho}$$

$\langle$**in** : $\langle a \overset{r,d}{\Leftarrow} cv\rangle\rangle$

$$\left\langle\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho} \rightarrow_\kappa \left\langle\!\left\langle\, \alpha \mid \mu\,, \langle a \overset{r,d}{\Leftarrow} cv\rangle \,\right\rangle\!\right\rangle_{\chi'}^{\rho},$$
$$\text{if a} \in \rho, \text{and FV(cv)} \cap \text{Dom}(\alpha) \subseteq \rho$$
$$\text{where } \chi' = \chi \cup (\text{FV(cv)} - \text{Dom}(\alpha))$$

$\langle$**out** : $\langle a \overset{r,d}{\Leftarrow} cv\rangle\rangle$

$$\left\langle\!\left\langle\, \alpha \mid \mu\,, \langle a \overset{r,d}{\Leftarrow} cv\rangle \,\right\rangle\!\right\rangle_{\chi}^{\rho} \rightarrow_\kappa \left\langle\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\right\rangle_{\chi}^{\rho'},$$
$$\text{if a} \in \chi, \text{where } \rho' = \rho \cup (\text{FV(cv)} \cap \text{Dom}(\alpha))$$

□

An actor terminates (**term**) its execution when it has reached the end of its behavior, an empty reduction context. The actor then becomes inactive and ready to process a new message in an environment with the updated state variables left by the previous processing. The interpretation of **send** is given by the **snd**-rule. A new message is added to μ. Message reception (message invocation) is described by the **rcv** transition. The message is removed form μ, and the receiver actor—formerly inactive, now busy—applies its behavior to the message. **in** and **out** respectively imports and exports messages, and updates the set of receptionists and external actors.

4 Real-Time Actor Semantics

This section defines the semantics of the real-time actor model. It is given in two steps. First, we give our interpretation of timed graphs used the underlying formalism for our semantics. The operational semantics for actors and

the timing constructs are then translated into a timed graph. Timed graphs is a real-time specification formalism similar to timed automata, originally proposed by Alur and Dill (see [5]), but has no accept states. The graphs we use here are inspired by the ones defined in [11].

4.1 Timed Graphs

A graph consists of nodes and edges, where nodes repressent system states, and edges repressent possible actions. Timed graphs are equipped with a set of clocks and an enabling condition for each edge (transition). An enabling condition is a predicate on the clocks: a transition can be taken only if the associated predicate is true. Further, a subset of clocks can be reset when the transition is taken. A progress condition is associated with each node. The graph may stay in a given node only if the progress condition for that node remains true.

Let **X** be the set of clock names used in this graph. The domain of clock values is chosen to be the set of positive reals, $\mathcal{R}_{\geq 0}$. Let n range over the set of nodes of the graph, γ over the set of actions, ξ over the set of enabling conditions. ν is a subset of clock names. We write $n \xrightarrow{(\gamma,\xi,\nu)}_{\sigma} n'$ when the graph in state n can take action γ with enabling condition ξ and reset (set to zero) the clocks in ν and thereby reach state n'. Enabling conditions are built using the logical connectives ($\vee$, $\wedge$, $\neg$) and primitive predicates. Primitive predicates have the form $x \leq c$ or $x \geq c$, where x ranges over clock-names and c over $\mathcal{R}_{\geq 0}$. Progress conditions are defined similarly. The progress condition associated node n is given by the function $\mathrm{Act(n)}$.

An example of a timed graph is illustrated in Figure 2.

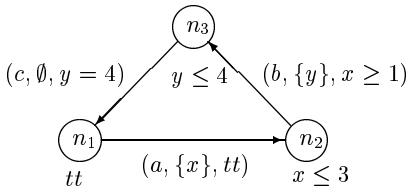

Figure 2: A timed graph with two clocks (x, y) and three nodes (n_1,n_2,n_3). The graph specifies that action b must occur between 1 and 3 time units after a, and c must occur 4 timeunits after b

Definition 2 *Timed Graph* $\rightarrow_G$*:*

$$\frac{n \xrightarrow{(\gamma,\xi,\nu)}_{\sigma} n' \wedge \xi(\varphi)}{(n,\varphi) \xrightarrow{\gamma}_G (n', \nu(\varphi))} \langle \textbf{action} : \gamma \rangle \qquad \frac{\forall r < d.\mathrm{Act(n)}(\varphi + \mathrm{r})}{(n,\varphi) \xrightarrow{\varepsilon(d)}_G (n, \varphi + d)} \langle \textbf{progress} : \varepsilon(d) \rangle$$

□

Our semantics of timed graphs is given by Definition 2. It consists of a rule for making actions (**action**) and a rule for delaying (**progress**). A configuration of the graph is represented by a (n, φ)-pair. n is the graphs current node and φ maintains the state of its clocks which is a mapping from clock-names to clock values. The notation $\nu(\varphi)$ denotes the clock-mapping that is equal to φ except for the clocks in ν which are reset. By the pre-condition, the enabling condition must evaluate to true for the transition to be enabled. A special action τ denotes internal transitions in the graph.

The **progress**-rule defines how the system makes progress. The passage of d time units is written $\varepsilon(d)$. $\varphi + d$ advances all clocks d time units. The graph may stay (let time pass) in a given node only if the progress condition for that node ($\mathrm{Act(n)}$) remains true. Allowing the graph to remain in a node for a specified amount of time enables the specification of certain loose properties, e.g., an action must happen in an interval.

4.2 Translation into Timed Graphs

So far, only the untimed semantics has been presented. We define the real-time semantics by translating the untimed transition system into a timed graph that takes into account timing information. The translation is given in two steps. First, the nodes and edges of the graph are defined, as well as the use of timers and enabling conditions. The necessary progress conditions follow.

The time bounds (r, d) carried by messages ($\langle a \overset{r;d}{\Leftarrow} cv \rangle$), that was uninterpreted in the untimed semantics, will now be given a meaning. The bounds specified on processing time are represented semantically by a function $u = ct(m)$ returning the required upper bound (u), where m is a message.

The nodes of the graph corresponds to system states, and edges to possible actions that causes state change. There is thus essentially a one-to-one correspondence between nodes and actor-configurations, and between edges and possible actions in a given configuration.

The nodes of the graph are actor configurations extended with two mappings for bookkeeping purposes: ac, mc. Whenever a message is invoked on an actor, we associate a (fresh) clock x with that actor. The mapping ac ($x = ac(a)$) records this association. The clock is reset when the message is invoked, thus recording the amount of time passed since the actor was activated. The clock is used to restrict the amount of time an actor uses to process a message, and to restrict the invocation time of messages sent as result of this processing. Note that a fresh clock is needed for each invocation because sent messages exist independently of the sender. The mapping mc associates a clock with each message ($x = mc(m)$) which, combined with the release time and deadline, specify when it is invokable. Whenever a message is sent this mapping is updated to record the correct timer.

Let A range over actor configurations, and let A^{ac}_{mc} denote a node in the graph with actor configuration A and actor clocks ac and message clocks mc. The nodes and edges of the graph is defined in Definition 3.

Definition 3 *Nodes and Edges* $\rightarrow_\sigma$

$$\frac{A \xrightarrow{\gamma}_\kappa A'}{A^{ac}_{mc} \xrightarrow{(\gamma',\nu,\xi)}_\sigma A'^{ac'}_{mc'}}$$

$case\, \gamma = \langle \textbf{rcv} : a, m\rangle :$
$\quad \gamma' = \tau_\gamma,\ \nu = \{y\},\ \xi = r \le x \le d,\ x = mc(m)$
$\quad mc' = mc,\ ac' = ac[a \mapsto y],\ y$ fresh clock
$\quad \text{where m} = \langle a \overset{r,d}{\Leftarrow} cv\rangle$

$case\, \gamma = \langle \textbf{snd} : a, m\rangle :$
$\quad \gamma' = \tau_\gamma,\ \nu = \emptyset,\ \xi = tt$
$\quad mc' = mc[m \mapsto x], x = ac(a)$

$case\, \gamma = \langle \textbf{fun} : a\rangle, \gamma = \langle \textbf{term} : a\rangle :$
$\quad \gamma' = \tau_\gamma,\ \nu = \emptyset,\ \xi = tt$
$\quad ac' = ac,\ mc' = mc$

$case\, \gamma = \langle \textbf{in} : m\rangle :$
$\quad \gamma' = \gamma,\ \nu = \{y\},\ \xi = tt$
$\quad mc' = mc[m \mapsto y],\ y$ fresh clock, $\text{ac}' = \text{ac}$

$case\, \gamma = \langle \textbf{out} : m\rangle :$
$\quad \gamma' = \langle \textbf{out} : \langle a \overset{r',d'}{\Leftarrow} cv\rangle\rangle$
$\quad \xi = tt,\ \nu = \emptyset$
$\quad r' = r - x,\ d' = d - x$
$\quad \text{where m} = \langle a \overset{r,d}{\Leftarrow} cv\rangle, \text{x} = \text{mc(m)}$ □

Whenever an actor (a) recieves a message (m), a hitherto unused clock[1] is reset and stored in the ac mapping. The enabling condition (ξ) must be true for the invocation to take place. When a message is sent, the clock needed to define its invocation time is recorded in mapping mc. The appropriate clock to use is the one allocated to the sending actor when it became active. Thus, $mc(m) = ac(a)$. The invocation time is thus correcly related to the invocation of the event that caused it to be sent, as required by the informal language definition. **fun** and **term** require no clocks and no enabling conditions.

A message sent from the environment to the system (**in**-transition) is handled similar to receive (**rcv**-transition): a fresh clock is allocated (and reset) to record the age of the message. Messages sent to external actors are forwarded (**out**-transition) with modified release time and deadline to compensate for the amount of time the message has spent inside the system.

The progress conditions for our semantics, presented in Definition 4, is defined by inspecting the transitions possible in a given node. These transitions each contribute with a condition. The node's progress condition is the conjunction of these.

Definition 4 *Progress Conditions* Act(n)

$(\gamma_i, \nu_i, \xi_i) \in \{(\gamma, \nu, \xi) \mid n \xrightarrow{(\gamma,\nu,\xi)}_\sigma\}$

$$\text{Act(n)} = \bigwedge_{\text{i}} \text{act}_\text{i}$$

$case\, \gamma_i = \langle \textbf{snd} : a, m\rangle,\ \gamma_i = \langle \textbf{fun} : a\rangle,\ \gamma_i = \langle \textbf{term} : a\rangle :$
$\quad \text{act}_\text{i} = \text{ac(a)} \le \text{ct(m)}$

$case\, \gamma_i = \langle \textbf{rcv} : a, m\rangle :$
$\quad act_i = ((y \ge ct(m) \rightarrow x \ne d) \ \vee\ (y \le ct(m) \rightarrow tt))$
$\quad y = ac(a),\ x = mc(m), \text{where m} = \langle a \overset{r,d}{\Leftarrow} cv\rangle$

$case\, \gamma_i = \langle \textbf{out} : m\rangle :$
$\quad \text{act}_\text{i} = \text{x} \le \text{d}, \text{x} = \text{mc(m)}, \text{where m} = \langle a \overset{r,d}{\Leftarrow} cv\rangle :$

$case\, \gamma_i = \langle \textbf{in} : m\rangle : \text{act}_\text{i} = tt$ □

The progress conditions define what actions are required to take place when, and are thus a crucial part of the semantics. Both restriction on computation time and message invocation affect the progress condition:

- An actor is allowed to perform its computation-steps any time within the specified computation limit, but is required to finish before the limit.
- A message is allowed to be invoked any time in its enabling interval. Since the receiver may be busy during this interval, messages are not guarenteed to be invoked. However, if the receiver actor is passive, thereby ready to accept a new message, and the allowed computation time has passed since its last activation, the message must be invoked before its deadline.

5 Discussion

During the recent years research in formal specification languages for real-time systems have received a lot of attention. A variety of time models and time extensions to traditional specification languages has been proposed and debated. Often, the models take the form of extended automata (Timed automata [5], Timed Graphs [5, 11], or process algebras (Timed CCS and Timed Modal Specifications [9], Timed CSP [15]). Although these are intended as specification languages, and not as programming languages *per se*, they have

[1] Allthough not done here, it is possible to recycle this clock once all messages referring to it has been invoked.

served as foundation for our semantics, and have been important sources of inspiration.

Others have defined semantics for real-time languages. In [14] semantics of a real-time object-oriented language is given by translating programs to RtCCS—a version of CCS [10] extended with time using explicit tick transitions. The resulting translation includes an abstract model of the execution environment (number of CPU's, scheduler, execution time). Similarly the semantics presented in [16] assumes knowledge about excution time of assignments and assumes a CPU for each process. Both, these semantics thus model relative concrete system, rather than being specifications for a set of possible systems, as is our goal.

A system in the TRA-formalism (Time Restriced Automata) [7], consists of a set of TRA's communicating by signaling events asynchronously through communication channels. A TRA specification defines timing constraints on causal events on in and output channels. This approach to specifying timing constraints resembles that of our **send** primitive. In TRA, it is only possible to specify physically realizable properties; time is required to advance between causally dependent events. Our model allows physically unrealizable programs to be specified, however, unrealizable behavior is only *required* when demanded by the programmer (e.g., deadlines of zero). No physical implementation is able to refine such a program. Actors differ from the TRA-model in many other ways.

The problem of defining the behavior of a real-time program in the presence of a limited set of shared resources is addressed by the Communicating Shared Resources (CSR) formalism [8, 13]. Here, a process always runs on some, possible shared, resource. A set of processes can be mapped to different sets of resources, hence describing different implementations. However, unlike our proposal, one cannot reason about program properties without giving a specific resource model. We find the CSR approach useful when validating refinements.

As future work, we plan to investigate if existing verification techniques for timed graphs can be used to verify properties for (restricted classes of) actor programs. We are also examining (automated) testing techniques for validating time constraints of implementations. Finally, we are looking into semantics and implementation of a more complex real-time language that allows expression of much more general timing constraints than the simple language presented here.

References

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

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

[3] Gul Agha. The Structure and Semantics of Actor Languages. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 1–59. Springer-Verlag, 1991.

[4] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, page 68pp, To be published.

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

[6] G. Berry and L. Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Lecture Notes in Computer Science*, volume 197, pages 389–448. Springer Verlag, 1984.

[7] Azer Bestavros. Specification and Verification of Real-time Embedded Systems using Time-constrained Reactive Automata. In *Proc. Real-Time Systems Symposium*, pages 244–253, San Antonio, TX, USA, 1991. IEEE.

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

[9] Jens Chr. Godskesen. *Timed Modal Specifications*. PhD thesis, Department of Mathematics and Computer Science, Institute for Electronic Systems, Aalborg University, October 1994.

[10] Robin Milner. *Communication and Concurrency*. Prentice Hall International (UK), 1989. 0-13-114984-9.

[11] 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.

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

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

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

[15] Steve Schneider. An Operational Semantics for Timed CSP. Tech. Report TR-1-91, Oxford University, February 1991.

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