Distributed, Real-Time Programming on Commodity POSIX Systems: A Preliminary Report*

Stefan D. Bruda, Petter Häggholm, and Scott Stoddard
Department of Computer Science, Bishop's University
Lennoxville, Quebec J1M 1Z7, Canada
{bruda,petter,scott}@cs.ubishops.ca

8 March 2006

Abstract

We present an incipient implementation of a programming language that allows programming of real-time applications distributed over a network. We have several goals in mind: First, the language should place the normal programmer (who tends to shy away from exceedingly formal constructs) in a comfortable environment. Secondly, programs written in this language should run on commodity systems, without relying on real-time support from the kernel. Thirdly, the language separates the code from timing restrictions, thus allowing for code re-use. Last but not least, the language should be built on a sound semantics and offer support for conformance testing and model checking.

1 Introduction

Real-time systems are systems where both the functional and the temporal behaviours are essential. The correctness of the system depends not only on the result of computations, but also on the time at which the results are produced. Building such systems has become increasingly difficult due to their use in larger and more complex applications. This increased complexity and the sophisticated demands of such systems in terms of safety and performance have generated increased interest in methodologies and tools that make the analysis and modelling of computing systems possible in terms of executable models.

Many real-time programming languages are in existence. Typical such languages allow the specification of time constraints along with the code being executed; the code then is restricted to run within the associated timing constraints. Examples include MPL [10] and RTC++ [8]. Timing restriction re-usability (which is absent from the mentioned languages) is introduced in an object oriented environment by HRTC++ [12], and can be further enhanced by introducing new constructs such as temporal abstract classes [11].

Languages like the ones mentioned above aim at easing the development and maintenance process of systems by encapsulating timing information into the objects; although certainly possible within the underlying (non-real-time) language, the development of distributed systems is not consciously considered. However, distributed systems (more generally, concurrent systems) are becoming popular at a tremendous rate. Languages such as DROL [17] address the distributed nature of systems explicitly. DROL is (once more) an extension of C++. It describes on a meta-level the semantics of message passing, including timing constraints; the semantics is based on asynchronous message passing. Timing constraints are, however embedded in the objects that communicate with each other. DROL is implemented on a specialized, real-time kernel.

*This research was supported by the Natural Sciences and Engineering Research Council of Canada and by the Fond québécois de recherche sur la nature et les technologies.

Another approach is taken in the modelling language ROOM (Real-Time Object Oriented Modeling) [15], which is based on establishing early operational models of the system and then refining them to implementation. Modeling of systems with ROOM is performed by designing *actors* communicating via point-to-point links by sending and receiving *messages*. The behavior of an actor is represented by an extended state machine. Incoming messages trigger transitions associated with the actor's state machine. ROOM does not constrain the actors in any way; instead, annotations to the message sequence charts can be used to indicate timing constraints, and constraints such as deadlines and periodicity are specified on end-to-end computations.

The particular way ROOM imposes timing constraints is worth noting as being opposed to the other programming languages. We believe that imposing end-to-end constraints is the logical way to go, for indeed timing constraints represent global, application level properties. When using a real-time language such as HRTC++ one must perform an extra step to infer the local constraints expressible by the language from the global constraints included in the specification. It is however conceivable that local constraints are some times needed (or at least handy) to obtain the desired behaviour.

A semantics of communication similar to ROOM's is also possible in a more general setting [9]: An RT-Synchronizers$^-$ system is defined as a collection of "active objects" that executes concurrently, communicating via messages; timing constraints are imposed on the messages. In this model, the functionality of objects can be implemented using a general purpose language (as opposed to ROOM's restricted model), and then the objects are glued together by interaction constraints. The constraints are established on top of ordinary objects. The RT-Synchronizers$^-$ semantics contains no reference to end-to-end (or global) timing constraints, though they are implicitly supported.

We also note that ROOM—as opposed to the other languages presented before—is a modelling rather than a general purpose language, so the language is based on formal models such as finite automata. Many other similar languages exist [13], along with languages based on process algebrae such as CSP [14] or their underlying semantics of labelled transition systems [7]. This kind of languages are preferred because of their suitability for formal model checking and conformance testing. On the other hand, they seem—at least in our experience—scary for the average computer science graduate (and thus for future programmers) who is rather used to programming in an imperative language.

We address in what follows the issue of real-time programming. We consider in the process the following characteristics (encountered in the languages presented above, though not necessarily together): a real-time system constructed using our programming language consists of a number of modules which are programmed in a general purpose programming language. The general purpose language itself is immaterial to the specification of timing constraints, which are instead imposed on the inter-module communication using constructs that are independent of the communicating modules. We aim for a separation between functional behaviour and timing constraints, which is natural for real-time specifications. In the process, we thus promote a component-oriented programming [5] style. Indeed, since modules themselves have nothing to do with real time, they can be changed and re-used at will, in a normal fashion.

Our language supports global (or end-to-end) timing constraints (which is once more the way real-time applications are specified). Recognizing that local constraints are potentially useful (even if not part of the specification), we also allow for the specification of point-to-point constraints (to messages that are passed from one module to another).

Initiating the transmission of inter-module messages is as much as possible transparent. A module issues a "method call" to another module in the same way as it issues a system call to a local method.

Building distributed systems is a central concern in designing the language. Many modules can be grouped into a process (or "node"), and many processes communicate to each other using internet sockets. Furthermore, the language is network-agnostic; no matter whether two modules run locally or on multiple machines in a distributed fashion, they communicate in the same manner.

Finally, a program written using our programming language runs on commodity operating systems. It does not rely on real-time support from the underlying operating system, about the only requirement is a POSIX-compliant operating system.

We are also interested in model checking and model-based testing. Our system provides a good foun-

$\langle \textbf{fun} : a \rangle$

$$\frac{E \vdash b \longrightarrow_{\lambda} E' \vdash b'}{\langle\!\langle \alpha, [E \vdash b]_a | \mu \rangle\!\rangle \longrightarrow_{\kappa} \langle\!\langle \alpha, [E' \vdash b']_a | \mu \rangle\!\rangle}$$

$\langle \textbf{snd} : a, \langle a' \Leftarrow \lambda, cv \rangle \rangle$

$$\langle\!\langle \alpha, [E \vdash \textbf{send}(a', cv); b]_a | \mu \rangle\!\rangle \longrightarrow_{\kappa} \langle\!\langle \alpha, [E \vdash b]_a | \mu; \langle a' \Leftarrow \lambda, cv \rangle \rangle\!\rangle$$

$\langle \textbf{sync\_snd} : a, \langle a' \Leftarrow a, cv \rangle \rangle$

$$\langle\!\langle \alpha, [E \vdash \textbf{sync\_send}(a', cv); b]_a | \mu \rangle\!\rangle \longrightarrow_{\kappa} \langle\!\langle \alpha, [E \vdash \textbf{wait}(a'); b]_a | \mu; \langle a' \Leftarrow a, cv \rangle \rangle\!\rangle$$

$\langle \textbf{wait} : a, a' \rangle$

$$\langle\!\langle \alpha, [E \vdash \textbf{wait}(a'); b]_a | \mu; ack(a) \rangle\!\rangle \longrightarrow_{\kappa} \langle\!\langle \alpha, [E \vdash b]_a | \mu \rangle\!\rangle$$

$\langle \textbf{rcv} : a, \langle a \Leftarrow a', cv \rangle \rangle$

$$\frac{E(w_a) \neq \lambda}{\langle\!\langle \alpha, [E \vdash \textbf{ready}(x); b]_a | \mu; \langle a \Leftarrow a', cv \rangle \rangle\!\rangle \longrightarrow_{\kappa} \langle\!\langle \alpha, [E[x \mapsto cv, w_a \mapsto a'] \vdash b]_a | \mu; ack(E(w_a)) \rangle\!\rangle}$$

$$\frac{E(w_a) = \lambda}{\langle\!\langle \alpha, [E \vdash \textbf{ready}(x); b]_a | \mu; \langle a \Leftarrow a', cv \rangle \rangle\!\rangle \longrightarrow_{\kappa} \langle\!\langle \alpha, [E[x \mapsto cv, w_a \mapsto a'] \vdash b]_a | \mu \rangle\!\rangle}$$

$\langle \textbf{delay} : d \rangle$

$$\langle\!\langle \alpha | \mu \rangle\!\rangle \xrightarrow{\varepsilon(d)}_{\kappa} \langle\!\langle \alpha | \mu \rangle\!\rangle$$

Figure 1: Actor configuration transitions.

dation for such a pursuit which is subject to future research.

We offer a semantics for such a system in Section 2. We then consider implementation issues and also the resulting RTSYNC programming language in Section 3. We note that this is a prototype rather than a full-blown language; we ellaborate on this and other concluding issues in both Section 3 and the concluding section.

2 Semantics

Our implementation follows a semantics based on the semantics of RT-Synchronizers^{-} [9] which is in turn based on the *actor model* [1]: distributed entities are modelled as self-contained objects called actors. An actor encapsulate a state, provides a set of public methods, and may call methods from other objects by means of message passing. Actors thus represent the layer of an RTSYNC program that executes the code specified by the user which (generally) interacts with the outside world. The actors execute concurrently, and are identified by unique names (sometimes "mail addresses"). The original message passing scheme is non-blocking and buffered, meaning that an actor resumes normal operation immediately after sending a message without waiting for a reply from the receiver, and that messages sent but not yet processed by the receiver are buffered until the receiver accepts them. Recognizing the however rare need for synchronization between actors, we also offer a blocking message passing scheme (introduced in the syntax by the **sync_snd** construct).

To send a message an actor calls a method of some other actor; a message $d.m(p)$ contains the name d of the destination actor, the name m of the method to be invoked at the destination, and possibly a set p of parameters to be passed to the method being called. For the blocking case the message needs to contain the name of the sender (though this needs not be specified explicitly by the programmer). We end up with an implementation syntax similar to the one noted above, but we use the notation $\langle a \Leftarrow a', cv \rangle$ in our semantical description, where a represents the destination of the message, a' the sender (or λ for non-blocking messages), and cv is the body of the message which includes the name of the method to be invoked

$\langle \mathbf{Sat}_{\prec} : \langle a \Leftarrow a', cv \rangle \rangle$

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

$\langle \mathbf{Sat}_{\succ} : \langle a \Leftarrow a', cv \rangle \rangle$

$$\frac{c_s = \begin{cases} \emptyset & \text{if } \langle a \Leftarrow a', cv \rangle \vDash p_2 \wedge d' \leq 0 \\ p_2 \prec d' & \text{otherwise} \end{cases} \qquad c_f = \begin{cases} p_2 \succ V(y) & \text{if } \langle a \Leftarrow a', cv \rangle \vDash p_1 \\ \emptyset & \text{otherwise} \end{cases}}{\{\!|\chi \uplus p_2 \succ d' | \xi_{\succ} |\!\} \xrightarrow{\langle a \Leftarrow a', cv \rangle}_{\gamma} \{\!|\chi \uplus c_f \uplus c_s | \xi_{\prec} |\!\}}$$

$\langle \mathbf{Sat}_{\emptyset} : \langle a \Leftarrow a', cv \rangle \rangle$

$$\frac{c_f = \begin{cases} p_2 \sim V(y) & \text{if } \langle a \Leftarrow a', cv \rangle \vDash p_1, \quad \sim \in \{\prec, \succ\} \\ \emptyset & \text{otherwise} \end{cases}}{\{\!|\emptyset | \xi_{\sim} |\!\} \xrightarrow{\langle a \Leftarrow a', cv \rangle}_{\gamma} \{\!|\emptyset \uplus c_f | \xi_{\sim} |\!\}}$$

$\langle \mathbf{Ack} : ack(a) \rangle \quad \sim \in \{\prec, \succ\}$

$$\{\!|\chi | \xi_{\sim} |\!\} \xrightarrow{ack(a)}_{\gamma} \{\!|\chi | \xi_{\sim} |\!\}$$

$\langle \mathbf{Delay}_{\sim} : e \rangle \quad \sim \in \{\prec, \succ\}$

$$\forall p_2 \prec d_i \in (\chi \ominus e), d_i \geq 0 : \{\!|\chi | \xi_{\sim} |\!\} \xrightarrow{\varepsilon(e)}_{\gamma} \{\!|\chi \ominus e | \xi_{\sim} |\!\}$$

Figure 2: Single constraint semantics; $\langle a \Leftarrow a', cv \rangle \vDash x_1(x_2)$ when $b \overset{def}{=} a = V(x_1) \wedge b(V[x_2 \mapsto cv])$

at the destination and also the parameters (the existence of a method name and parameters is immaterial for the message passing mechanism).

With this structure for messages, the semantics of actors is given by the transition system $\longrightarrow_{\kappa}$ shown in Figure 1. The state of a set of actors is modeled as a pair $\langle\!\langle \alpha | \mu \rangle\!\rangle$, with α and μ representing the actor state and the set of pending messages, respectively. The state of an actor a is written $[E \vdash b]_a$ with E the environment and b the remainder of the actor behaviour. The actor reduces its behaviour until it reaches **ready**(x); this signifies that actor a is waiting for a message (whose content is then bound to x). The transition system $\longrightarrow_{\lambda}$ defines the semantics of the programming language used to program actor behaviour. The environment variables w_a are fresh variables initialized with the special value λ; they hold the sender of the last blocking message (in order to send an acknowledgment when the called method completed), or λ whenever the last message was non-blocking.

We note that one cannot make timing assertions about the execution of an actor program directly. In order to make this observation explicit the delay actions $\varepsilon(d)$ is introduced, with d a positive number representing the passage of d time units. The actor evolves by performing a sequence of instantaneous actions and by letting the time pass.

Messages are intercepted by *synchronizers*. They are reactive agents that exist to monitor and enforce time constraints upon the inter-actor message communication. Every actor-actor relation (a relation between actors a and b exists whenever a references a method of b, or vice-versa) may have an associated ("coordinating") synchronizer. This synchronizer then intercepts all communication between the two actors and enforces user-defined timing constraints expressed as real-time constraints on pairs of message invocations.

The semantics of one constraint in a synchronizer is given by the transition system $\longrightarrow_{\gamma}$ shown in Figure 2. The state variables of a synchronizer are represented by an environment V that maps variables to their values. A constraint can have two forms: $p_1 \implies p_2 \sim y$, $\sim \in \{\prec, \succ\}$, where p_1, p_2 are message patterns, and y is a positive, real valued variable or constant; the first form specifies that a message matching the pattern p_2 must follow a message matching p_1 no later than y time units. Conversely, the second form specifies that y time units must pass after a message matching p_1 occurs before a message matching p_2 is

$\langle \textbf{Action} : l \rangle$

$$\frac{\forall i \in [1 \ldots n], \gamma_i \xrightarrow{l}_{\gamma} \gamma'_i}{\langle\!| \gamma_1, \ldots, \gamma_n | V |\!\rangle \xrightarrow{l}_{\sigma} \langle\!| \gamma'_1, \ldots, \gamma'_n | V' |\!\rangle},$$
$$l \in \{\langle a \Leftarrow a', cv \rangle, \varepsilon(e), ack(a)\}$$

Figure 3: Synchronizer semantics.

allowed. Whenever an invocation matches p_1 the constraint fires and thus creates a new demand instance matching p_2. Such a demand is represented by a triple $p_2 \sim d$, with d (a real number) denoting the deadline or release time of p_2. The state of a constraint is represented as a constraint configuration $\langle\!| \chi | \xi_{\sim} |\!\rangle$; $\xi_{\sim}$ is a static description of a constraint $p_1 \implies p_2 \sim y$ and χ is a multi-set of demands instantiated from this static description.

The passage of time is controlled by the **Delay** rule. The elapsed time e is subtracted from d_i in each demand $p_i \sim d_i$; this is written $\chi \ominus e$. The delay rule ensures that deadline constraints are always satisfied.

A synchronizer is then represented by a synchronizer configuration $\langle \overline{\gamma} | V \rangle$ where γ is a set of constraint configurations ranged over by γ_i and V represent the state variables of a synchronizer by means of a mapping from identifiers to their values. The semantics of a synchronizer is shown in Figure 3. Triggers can be attached to events, and their effect (not shown in the figure but rather trivial to consider) is to transform V according to the assignments specified therein.

For convenience we differentiate between point-to-point synchronizers (just synchronizers for short) and global synchronizers. The global synchronizers impose general (and not only point-to-point) constraints.

Constraining an actor program is defined as a special form of parallel composition $\|$. An interaction constraint system configuration is written as $(\!(\sigma_1, \ldots, \sigma_n)\!)$, where σ ranges over synchronizer configurations. The composition of an actor configuration and an interaction constraint system is defined by the transition system $\longrightarrow_{\kappa\sigma}$ shown in Figure 4.

3 The RTSYNC system

An RTSYNC (distributed) program consists in a set of RTSYNC *nodes*. A node is a standalone executable that can work in isolation or communicate with other nodes through internet domain sockets. Each node contains a number of *actors* (which perform non-constrained actions) and *synchronizers* (which enforce timing constraints over inter-actor communication). Both the inter-node communication and the intra-node message passing is coordinated by *postmasters*. Each RTSYNC node runs exactly one such a postmaster.

RTSYNC nodes are C++ programs that contain blocks of RTSYNC specific code. The RTSYNC compiler takes source programs in this format, parses the RTSYNC specific-code, and so generates C++ code which it outputs along with the surrounding C++ code. This resultant code file can then be compiled with a C++ compiler to produce an executable. During compilation a run time library must be linked in. The library contains all of the runtime code needed by the program (parent classes, postmaster code, logging, etc.).

Our programming language produces code that does not require real-time resources from the operating system. Indeed, about the only requirements are a POSIX-compliant operating system, an ANSI C++ compiler, and the potable BOOST libraries [2].

In a system such as ours dynamic priority for the participating threads (assigned according to the priorities of the pending messages) is imperative. Scheduling threads for execution according to their priorities is accomplished by a system that blocks all but the highest priority threads; the unblocked threads run then concurrently being scheduled on the processor(s) according to the (non real time) kernel scheduler. Message queues are implemented on a per-thread basis. Message priorities (which in turn determine thread

Untimed actions

$$\frac{\langle\!\langle \alpha|\mu \rangle\!\rangle \xrightarrow{l}_{\kappa} \langle\!\langle \alpha'|\mu' \rangle\!\rangle \qquad l \in \left\{ \begin{array}{l} \langle \textbf{fun} : a\rangle, \langle \textbf{snd} : a, m\rangle, \langle \textbf{sync\_snd} : a, m\rangle, \\ \langle \textbf{wait} : a, a'\rangle, \langle \textbf{ready} : a\rangle \end{array} \right\}}{\langle\!\langle \alpha|\mu \rangle\!\rangle \,\|\, (\!(\sigma_1, \ldots, \sigma_n)\!) \xrightarrow{l}_{\kappa\sigma} \langle\!\langle \alpha'|\mu' \rangle\!\rangle \,\|\, (\!(\sigma_1, \ldots, \sigma_n)\!)}$$

Receive

$$\frac{\langle\!\langle \alpha|\mu \rangle\!\rangle \xrightarrow{l}_{\kappa} \langle\!\langle \alpha'|\mu' \rangle\!\rangle \qquad \bigwedge_{i\in[1\ldots n]} \sigma_i \xrightarrow{\langle a \Leftarrow a', cv\rangle}_{\gamma} \sigma_i' \qquad l = \{\langle \textbf{recv} : a, \langle a \Leftarrow a', cv\rangle\rangle\}}{\langle\!\langle \alpha|\mu \rangle\!\rangle \,\|\, (\!(\sigma_1, \ldots, \sigma_n)\!) \xrightarrow{l}_{\kappa\sigma} \langle\!\langle \alpha'|\mu' \rangle\!\rangle \,\|\, (\!(\sigma_1', \ldots, \sigma_n')\!)}$$

Synchronization

$$\frac{\bigwedge_{i\in[1\ldots n]} \sigma_i \xrightarrow{ack(a)}_{\gamma} \sigma_i'}{\langle\!\langle \alpha|\mu \rangle\!\rangle \,\|\, (\!(\sigma_1, \ldots, \sigma_n)\!) \xrightarrow{ack(a)}_{\kappa\sigma} \langle\!\langle \alpha'|\mu' \rangle\!\rangle \,\|\, (\!(\sigma_1', \ldots, \sigma_n')\!)}$$

Delay

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

Figure 4: Combined semantics of a RT-Synchronizers$^-$ system.

priorities) are assigned dynamically, based on their time constraints.

In essence our implementation is based on the generalized rate-monotonic scheduling theory [16], under which all the tasks meet their deadlines as long as the system load lies below a certain bound. Given the assumed lack of real-time support from the kernel (and the limited actual support currently available on real-time systems) this mechanism is implemented using signals, which implies a certain time coarseness; however, the coarseness is similar to the one induced by the network communication stack so we do not consider it as a serious limitation.

The distributed nature of the system is very flexible, being implemented in a "peer to peer" fashion. No centralized control exists, and individual "nodes" are free to enter and leave the network at any time.

Timing violations are of course always detected at run time. Once such a violation is detected a handling routine is run. The programmer has the option of defining such a handler for each synchronizer; upon completion the handler may optionally terminate of the current node or the whole system.

Static timing analysis (and in general conformance testing) is more useful than run-time detection and is our main, active research interest. At this time static analysis produces a timed call graph for method calls and then computes its reflexive and transitive closure in order to take global constraints into consideration. The process is restricted to constant time restrictions, and does not take into consideration the run time of unrestricted code inside actors. It is also a somehow ad-hoc process, having the role of a stub for future development rather than being a good implementation.

3.1 Actors

Actors are the active agents of the system: they execute code and generate messages for communication ("method calls"); the other RTSYNC entities merely serve as support and co-ordination. Messages are thus formed and eventually decoded and acted upon solely at the actor level (whenever a "method call" is performed within an actor, referencing a method of a foreign actor, a message is generated). Note that since messages only occur in inter-actor communication, and since only messages are subject to time constraints, activities occurring within an actor cannot be time constrained.

actor *name* {
    C- variable declarations
    init { C- code }
    method-name (*parameters*) { C- code }
    method-name (*parameters*) { C- code }
    . . .
}

Figure 5: The structure of an actor block; keywords are shown in bold face.

An actor is defined in RTSYNC as a block whose structure is shown in Figure 5. An actor is identified by its name specified after the **actor** keyword. It contains a set of declarations for the local variables, followed by a series of blocks of code. The first such a block (the initialization code, introduced by the keyword **init**) is executed when the system starts up. The next blocks of code are methods triggered by incoming messages. Such blocks are introduced by the name of the method, followed by a list of formal parameters, followed by an arbitrary block of code that implements the respective method.

Actors are written in a general purpose language such as C. In the current, proof of concept implementation we use a subset of C dubbed C-. The language (described by a rather readable YACC source) is immaterial to the functionality of the system, and it can be easily enhanced (or even changed altogether).

One special statement of the language is the one that implements the call of a method (of another actor). This syntax of such a statement is $a.m(p)$, with a the name of the actor whose method m is to be executed with p as list of arguments. Once the statement is executed a message is sent to actor a. Such a message is intercepted and processed by a synchronizer, so the actual message passing mechanism is described in the next section.

The default non-blocking semantics can be changed by prefixing the method call with the keyword **sync**. In such a case the caller blocks until the callee completes the execution of the method specified by the message.

3.2 Synchronizers

The syntax of a synchronizer is shown in Figure 6. A synchronizer consists in four parts: a set of instantiation parameters, instantiation code, and declaration of local variables; a set of constraints; a set of triggers; and an exception handler. Messages of interest are identified by patterns, which have in the current implementation the form $a.m$ where a specifies the destination actor and m is the name of the method to be called as a consequence of the message.

As for the actors, the initialization code is introduced by the keyword **init**. Unlike the actors, the **init** keyword is followed by a construct of the form **(** *actor1*<->*actor2*) **)**. This construct specifies that the current synchronizer intercepts messages exchanged between actors *actor1* and *actor2*.

Only one synchronizer may be the designated coordinator of any given actor pair, and the relation is commutative. The presence of an explicitly defined synchronizer at any such relation is optional; relationships may be unconstrained, in which case the system automatically provides a null synchronizer; no assertions regarding timing can be made in such a case.

Recall that a constraint can have two forms: $p_1 \implies p_2 \prec y$ and $p_1 \implies p_2 \succ y$, where p_1, p_2 are message patterns, and y is a positive, real valued variable or constant. These two forms are introduced in the current implementation by the keywords **min** and **max**, respectively. A message intercepted by the synchronizer is continuously "polled." The synchronizer examines the message's time constraints to determine whether the message is a valid candidate for execution (i.e., the minimum time before execution has passed) and whether the message has violated any constraints (i.e. whether its "must run before" time has passed), and taking the proper action. In the first case, the synchronizer holds a message until it is a valid candidate for execution, at which point the synchronizer relays it to its proper destination. In the case of a message violating timing constraints, the synchronizer throws an exception and drops the message (which is thus never delivered).

```
synchronizer name {
    C- variable declarations
    init ( actor1<->actor2) ) { C- code }
    constraint {
        min actor.method -> actor'.method' = expr ;
        max actor.method -> actor'.method' = expr ;
        . . .
    }
    trigger {
        enable ( actor.method ) when expr ;
        disable ( actor.method ) when expr ;
        action ( actor.method ) { C- code }
        . . .
    }
    exception { C- code with return statement }
}
```

Figure 6: The structure of a synchronizer block.

Once the timing constraints managed by the synchronizer are violated, the code from the exception section is called. This is the only section in which the **return** keyword is allowed, with the argument `TERMINATE` (with the effect that the whole RTSync system is terminated), `KILLNODE` (with the effect that the current node is terminated), or `CONTINUE` (which specifies that the exception has been taken care of and execution can continue).

Synchronizers may also perform non-coordinating tasks by responding to arbitrary events ("events" being specific methods calls) by means of triggers. These can be action triggers, which are arbitrary code blocks executed upon notification of certain events, or enable/disable triggers, which globally enable or disable individual methods of actors subsequent to particular events.

Global synchronizers In addition to the point-to-point timing constraints specified by the RT-Synchronizers$^-$ [9] semantics (and implemented by synchronizers), ROOM's *transaction constraints* (also referred to as "global constraints" [9]) are implemented by *global synchronizers*. The syntax of global synchronizers is similar to the syntax of (point-to-point) synchronizers (shown in Figure 6), except that they are introduced by the keyword **global** rather than **synchronizer**.

Transaction constraints must be synchronized by the global synchronizer. Any constraint expression over relations that are not point-to-point in terms of method calls, although syntactically valid, are not allowed in normal synchronizers; synchronizers are validated against the timed call graph.

3.3 The postmaster

The postmaster serves as the universal relay and dispatch engine for an RTSync system. All calls originating from actors or synchronizers route through the postmaster so that the operation of actors and synchronizers is fully network transparent. The postmaster holds data structures that contain information about all RTSync entities within the local node, and essential information about all remote nodes.

Each node contains its own postmaster which allows a distributed RTSync program to operate as a peer-to-peer system. As such, there is no server, rather clients fulfill both roles as needed throughout the lifespan of the system. The general process is as follows:

For a single node system (or the first node of a distributed system), the node simply activates, sets up a server port, and then goes about running (to the extent that it can if it is waiting for other nodes).

For a node joining an already running system the process is a little different. While it will still set up a server port, it will (in the constructor) also connect to the host that it has been provided with. This initial interaction between client and "server" serves to populate the new client with information about all of the other nodes (postmasters) in the network. From there, the new client will initiate communication with each of the other nodes of the network to exchange information with them about their respective setups. This is referred to as the population stage. Both client and server complete this stage when they have each received a snapshot of the other's structure (members, methods, relationships...)

Once (initial) population is complete, the postmaster will continue execution of the `run` method which is to say that it will either go into an infinite poll loop or begin system execution by passing an initial message. The network messaging protocol uses a simple, plain-text, format for transmission.

4 RTSYNCinternals

Note that this section is in the process of being expanded, so its current incarnation shall not be considered final by any means.

4.1 Actors

Every actor in the system inherits from the `actor` class and for the purpose of the modular scheduler also inherits from `schedulable`. The parent `actor` class defines the necessary functions such that every actor can setup a polling thread after construction. The purpose of an actor's polling thread is to request messages from synchronizers (whenever it has message registrations in `m_msg_requests`) and to execute the methods contained in those messages. Note that execution is done in the polling thread and therefore only one message is acted upon at a time.

4.2 Messages in an RTSYNC system

Messages go through a (necessarily) complex handshake process as they travel through an RTSYNC system. A message is thus subject to the following process (controlled by postmasters) from inception to destruction:

1. Message creation is done by source actor; creation timestamp is set.
2. The message is then passed to coordinating synchronizer by invoking `sm_receive_msg` of the postmaster; the dispatch timestamp is set by synchronizer when it receives the message.
3. The synchronizer holds the message in its incoming queue until time constraints allow for its sending to the destination actor.
4. The message is then moved to outgoing queue and the destination actor is notified that synchronizer has a message holding for it (via `am_register_msg`).
5. When the actor's polling thread is ready for this message (all messages are received in order of their timestamp), it calls `sm_request_msg` of the postmaster; the return value of this call is the message; the received timestamp is set upon message arrival at the actor.
6. The actor executes the method specified by the message using the parameters contained within; execution timestamp is set when execution is complete.
7. The actor calls `sm_ack_msg` of the postmaster; the result of this call is a passing of the message to `ack_msg` of the synchronizer; this function's purpose is limited to status messages and to the implementation of synchronous messages.
8. Actor notifies all of its registered listeners that it completed execution of the method specified by the message.

4.3 The network application protocol

The network messaging protocol uses a simple, plain-text, format for transmission. The general format of a transmission packet consists of the following tokens: *size of packet* (excluding the size itself and its trailing space), *protocol directive* (enumerated found in `postmaster.h`), and *optional parameters* (directive dependent).

A sample message is provided below. This message is of type `RT_REQUEST` and would be sent to a node that contains a synchronizer named `mysync`. After parsing, that synchronizer would be informed that actor `someactor` would like a message from it.

```
18 3 mysync someactor
```

In general the system uses the following protocol directives:

- `RT_MESSAGE`: normal RTSYNC message to be acted upon (i.e. send to receiver); parameters: the message.
- `RT_NOTIF`: notification of method completion (send to listener); parameters: syncref, actorref, rttime_t, method.
- `RT_REQUEST`: notifies a synchronizer that an actor would like its next message; parameters: syncref, actorref.
- `RT_REQRESPONSE`: reponse to an `RT_REQUEST` (contains a message); parameters: the message.
- `RT_REGMSG`: notifies an actor that a message is waiting; parameters: actorref, syncref, rttime_t.
- `RT_ENABLE`: enable an actor's method; parameters: actorref, method.
- `RT_DISABLE`: disables an actor's method; parameters: actorref, method.
- `RT_REG_LISTENER`: register synchronizer as listener to actor; parameters: actorref, syncref.
- `RT_RECVMSG`: synchronizer received message; parameters: syncref, message.
- `RT_ACKMSG`: acknowledge reception of message from synchronizer; parameters: syncref, message.
- `RT_POST_DISABLE`: send notification to another postmaster that method was disabled; parameters: actorref, method.
- `RT_POST_ENABLE`: send notification to another postmaster that method was enabled; parameters: actorref, method.
- `RT_GET_NETIDS`: request the list of net id-s from another postmaster; parameters: none.
- `RT_POPULATE`: request a postmaster's list of local actors and synchronizers; parameters: none.
- `RT_POPULATION`: response to a populate request (or send by new postmaster); parameters: full population data from a node (see below).
- `RT_TERMINATE`: signal a postmaster sends to other postmasters to tell them that it is going to terminate; parameters: none.
- `RT_KILLALL`: once received by a postmaster, it should immediately kill itself; the whole system is going down.
- `RT_TIMEUPDATE`: notification that a method has been run with new lastrun timestamp; parameters: actorref, method, rttime_t.
- `RT_SYNCNOTE`: notification to actor locked in synchronous call that remote party finished execution; parameters: message.

The following is the structure of a `RT_POPULATION` packet (all tokens in each section are separated by spaces, and all sections of the `RT_POPULATION` packet are separated from previous sections by a dollar sign as shown):

list of actors (local to the remote node) **$**
list of synchronizers (local to the remote node) **$**
list of (2 actors, followed by a synchronizer (synchronizing pairs)) (global) **$**
list of (actor, method, boolean enabled flag) (global list of all methods in the system) **$**

5 Conclusions

Recall that our goals is developing the RTSYNC system were the study of a distributed, real-time programming language that should place the normal programmer in a familiar environment, that should produce code capable of running on commodity systems, that should offer separation between time restrictions and code (thus facilitating code re-use), and that should offer support for conformance testing.

We described here an incipient implementation of such a system. The implementation should be regarded as a proof of concept rather than a functional application. We have addressed the first three goals by providing a familiar C-like language to the programmer (still to be enhanced to a production language) and by offering an implementation based on dynamic priorities that should satisfy all the feasible real-time tasks. Our implementation uses internet domain sockets and is thus suitable for both nodes distributed over a network and nodes running concurrently on one machine (the latter kind of nodes communicating over the loopback subnet).

The system is also based on a sound semantical model. This offers a good starting point for the last (and most important) goal. The present concept offers support for testing real-time systems, or to be more precise for interfacing a running system with one or more testing systems. Indeed, beside the usual input/output interface provided by the operating systems, a testing system is able to connect to the system under test using the message passing interface already provided. A tester would be thus able to test various time properties (not necessarily limited to existent time constraints since the tester could introduce more time restrictions of its own). We believe this to be a powerful tool and so a significant contribution.

This being said, we are interested in model-based testing. At this time we offer some tools that could be used to accomplish this, but we fall short of actually performing any kind of model checking. We base our implementation on a well defined semantics, so one would be able to express the model semantically and then derive both the test cases and the application itself starting from the semantical model. Once this is accomplished, the actual conformance testing of the end product is easily done as mentioned above. However this falls well short of the stated goal that our system is a tool suitable for programmers not necessarily comfortable in a formal environment. We are currently interested in studying the possibility of deriving a model and test cases starting from the RTSYNC code itself. The mentioned timed call graph currently computed by the system should be viewed as a stub that is to be replaced in the future by the results of our current research.

At this time—i.e., without a proper model checking in place—our language should be regarded as producing soft real-time systems; we intend to strengthen the language so that it is able to produce hard real-time systems at least for particular structures of such a system. Even so, the unpredictability of network communication is expected to pose additional problems. The issue of network communication needs further consideration.

We note finally the relation between our current work outlined here and timed ω-languages, the previously developed complexity-theoretic model of parallel real-time computations [3, 4]. Timed ω-languages are in particular suited for describing (or being described by) the interface between nodes in an RTSYNC system. Establishing the semantical model presented here as an abstract machine for timed ω-languages and deriving further complexity results for real time is also one of our long term interests.

In all, this paper presents a preliminary report of what we believe to be a powerful tool in communizing real-time, distributed programming. This is one small step towards this goal; future developments are in the works theoretically, and once developed their implementation should pose little problems given the support already built into the system. The second goal of the paper is to gauge the interest of the applied community in such a tool; this interest will be a factor in the speediness of future developments of the RTSYNC system.

The RTSYNC system is available on the Web at http://turing.ubishops.ca/home/bruda/rtsync and can be distributed freely under the terms of the GNU General Public License [6]. An extended version of this document (including programming details that were left out of this document due to space restrictions) is also provided. Given the preliminary character of this paper (and also the space limitations) and the system being under active development we do not provide programming examples or testing of the system, though these should be available in the near future.

References

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

[2] BOOST C++ libraries. http://www.boost.org/.

[3] S. D. Bruda and S. G. Akl. Pursuit and evasion on a ring: An infinite hierarchy for parallel real-time systems. *Theory of Computing Systems*, 34(6):565–576, 2001. For an extended version see http://turing.ubishops.ca/home/-bruda/papers/pursuit.

[4] S. D. Bruda and S. G. Akl. Real-time computation: A formal definition and its applications. *International Journal of Computers and Applications*, 25(4):247–257, 2003. For a preliminary version see http://turing.ubishops.ca/home/-bruda/papers/timed-langs.

[5] M. Franz, P. H. Fröhlich, and T. Kistler. Towards language support for component-oriented real-time programming. In *Proceedings of the International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, Monterey, CA, Nov. 1999. IEEE Press.

[6] Free Software Foundation. The GNU general public license. http://www.gnu.org/copyleft/gpl.html.

[7] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Jan. 1991.

[8] Y. Ishikawa and H. Tokuda. Distributed real-time programming language: RTC++. *Computer Software*, 9(2):28–47, Mar. 1992.

[9] B. Nielsen, S. Ren, and G. Agha. Specification of real-time interaction constraints. In *The First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 206–214, Kyoto, Japan, Apr. 1998. IEEE Computer Society Press.

[10] V. M. Nirkhe, S. K. Tripathi, and A. K. Agrawala. Language support for Maruti real-time system. Technical report, University of Maryland at College Park, 1990. Univ. of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-90-76.

[11] A. P. Pons. Temporal abstract classes and virtual temporal specifications for real-time systems. *ACM Transactions on Software Engineering and Methodology*, 11(3):291–308, 2002.

[12] A. P. Pons. An object-oriented language for real-time systems. *International Jornal of Computers and Applications*, 26(1):31–37, 2004.

[13] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for automated implementation of executable object oriented models for real-time embedded control systems. In *The 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 240–252. IEEE, Dec. 1997.

[14] S. Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. John Willey and Sons, 2000.

[15] B. Selic, G. Gullekson, , and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[16] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of IEEE*, 82(1):68–82, Jan. 1994.

[17] K. Takashio and M. Tokoro. DROL: An object-oriented programming language for distributed real-time systems. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 276–294, 1992.