RTC: Language Support For Real-Time Concurrency[0]

Victor Wolfe

Computer Science and Statistics
The University of Rhode Island
Kingston, RI 02881

Susan Davidson and Insup Lee

Computer and Information Science
The University of Pennsylvania
Philadelphia, PA 19104

Abstract

This paper presents language constructs for the expression of timing and concurrency requirements in distributed real-time programs. The programming paradigm combines an object-based paradigm for the specification of shared resources, a distributed transaction-based paradigm for the specification of application processes, and explicit timing constraint expression. An implementation of the language constructs with real-time scheduling and locking for concurrency control is also described.

1 Introduction

In real-time applications such as robotics, industrial control and avionics, programs must meet timing and concurrency constraints to be correct. As an example, consider a simplified robotics application where two robot arms must lift a container of chemicals from a moving conveyer belt. The arms are shared among the lifting task and other tasks that execute concurrently in the application. To prevent spills when lifting, the following constraints on the operation of the arms must be expressed in its control program: the arms should lift simultaneously, no other use of the arms should be allowed while the lift is being performed, and either both arms should lift or neither arm should lift. The lifting should also meet timing constraints that arise from the dynamics of the moving belt and inherent properties of robot control algorithms. Furthermore, recovery should be specified for violations of any of these constraints.

To support such concurrent real-time applications, a programming language and its run-time system should have the following characteristics. First, the language should facilitate the expression of *real-time concurrency* constraints, *i.e.* the functional and timing constraints imposed by the application. Many of these constraints are illustrated in the example, including absolute timing constraints, exclusive execution, simultaneous execution, all-or-nothing execution, and predictable execution. Second, its run-time system should enforce as many constraints as possible. Third, the language should support the specification of recovery since some constraints may be violated at run-time. Fourth, the language should support the modular decomposition of complex concurrent real-time systems.

Some concurrent real-time languages, such as Ada and Modula-2, require that scheduling primitives be added to programs to meet constraints. In Ada, the programmer must determine the static priorities of tasks from these constraints so that priority-based scheduling of the tasks meets the constraints. In Modula-2, the programmer must explicitly add transfer commands so that co-routines coordinate to meet their constraints. Since the constraints are not explicitly stated, but hidden in scheduling, programs are difficult to write, verify and modify. Detecting and recovering from constraint violations is also complicated by the constraints being hidden. Furthermore, since the scheduling primitives are added at compile-time, their ability to cope with dynamic environments is limited.

Recent real-time languages such as Flex [1] and Real-time Euclid [2] allow explicit expression of some timing constraints. However, the constraints are only used for scheduling the CPU; mutual exclusion is used to control concurrent access to other resources. This has two disadvantages: First, access to resources is first-come, first-served; no timing information is used. Second, no concurrent access is allowed, even if it does not violate the consistency of the resource. Hence, the run-time system may not be able to meet the stated timing constraints, although they could be met using

[0]This work is supported in part by the following grants: ARO DAAG-29-84-k-0061, ONR N000014-89-J-1131 and NSF CCR87-16975.

0-8186-2450-7/91 $1.00©1991 IEEE

other techniques.

Consistent concurrent execution is often supported by *transactions* [3]. Unfortunately, traditional transactions do not support timing constraints, and the notion of "independence" of transactions disallows explicit precedence orderings among them. For instance, it is not possible to specify that a transaction for lifting the container must always execute after a transaction that detects the container.

Our approach to concurrent real-time programming is to explicitly express real-time concurrency constraints in a program and allow the run-time system to enforce them. To define these constraints precisely, we develop a real-time concurrency model that combines an object-based paradigm for the specification of shared resources, a distributed transaction-based paradigm for the specification of application processes, support for timing constraints, and support for precedence orderings.

The rest of this paper is structured as follows: Section 2 presents our real-time concurrency model. In Section 3, we use the model to define the Real-Time Concurrency (*RTC*) language constructs. A complete language is not developed. Rather, the *RTC* constructs are designed to be embedded in any block-structured procedural host language; our current implementation is in C. Section 4 discusses how we use real-time scheduling and two-level locking to implement the constructs. Section 5 summarizes and compares our work to other real-time languages.

2 Model

Our model of distributed real-time computing combines an object-based paradigm with a transaction-based paradigm and adds provisions for timing and precedence constraints. The model consists of *resources* and *processes*. Resources capture an object-based paradigm by providing abstract views of shared system entities, such as devices and data structures. Each resource has a state and defines a set of transactions, called *actions*, that can be invoked by processes to examine or change the resource's state. A resource also specifies in a "compatibility predicate" which actions are compatible, *i.e.*, which actions can have overlapping executions and still preserve its state's consistency. Processes specify a set of action invocations along with precedence orderings, transaction-based consistency constraints, and timing constraints.

2.1 Resources

All shared data structures in the system are specified as *resources*. A resource, r, is characterized as $\langle S_r, A_r, \mathcal{P}_r, C_r \rangle$, where S_r is a set of states, A_r is a set of *actions*, $\mathcal{P}_r$ is a set of processors on which actions of A_r can be executed, and C_r is the compatibility predicate. We assume that a resource can be implemented as a multi-processor component, and therefore that its actions can be executed simultaneously on multiple processors in $\mathcal{P}_r$; however, the resource's state is shared. A process can only use a resource by invoking its actions. When an action invocation completes normally, it changes the resource to a *consistent* state (*i.e.*, a state that meets application requirements) and returns values to the process. An action may be *aborted* by its calling process. An aborted action *recovers* by restoring the resource to a consistent state. In our example, a robot arm is a resource. Its state includes the Cartesian position of its arm and the position of its hand (grasp/ungrasp). Its actions include: *lift*, which increments the z-coordinate of the arm's state; *lower* which decrements the z-coordinate; *grasp*, which affects the hand; and *read* which does not change the state, but returns its values.

The *schedule* of a resource r is $Sch_r = \langle H_r, start_r, complete_r \rangle$, where H_r is the set of all action invocations on the resource, $start_r : H_r \rightarrow time$ maps each action invocation in H_r to the absolute time at which it started executing, and $complete_r : H_r \rightarrow time$ maps each action invocation in H_r to the absolute time at which the action terminated. A resource's schedule defines a partial order, $\prec_r$, on the action invocations of H_r such that, for two invocations $a_1 \in H_r$ and $a_2 \in H_r$, $a_1 \prec_r a_2 \Rightarrow complete_r(a_1) \leq start_r(a_2)$. The ordering $\prec_r$ is partial because the execution of actions may be *overlapped* in the schedule and thus neither $complete_r(a_1) \leq start_r(a_2)$ nor $complete_r(a_2) \leq start_r(a_1)$. Action invocations may either overlap because they execute concurrently on different processors of $\mathcal{P}_r$ or because they are interleaved on the same processor.

C_r defines a *compatibility predicate* that describes conflicts between actions, and is used to define acceptable overlapped executions of actions. That is, if $a_1, a_2 \in A_r$ and $C_r(a_1, a_2) = TRUE$, then all overlapped executions of a_1 and a_2 in a schedule for resource r produce the same state for r and the same return values for a_1 and a_2. In the example the compatibility predicate for an arm includes: $C_{arm}(lift, grasp) = TRUE$ because the actions affect different parts of the state; $C_{arm}(read, read) = TRUE$ because the state is not affected; and

$C_{arm}(lift, lower) = FALSE$ because overlapping these actions could leave the state inconsistent. To ensure that resource r's state remains consistent, we require that all schedules for r be *serializable*, *i.e.*, equivalent to some schedule in which there are no overlapped action executions. Two schedules Sch_1 and Sch_2 for r are *equivalent* iff r has the same final state in Sch_1 and Sch_2, and every action invocation of Sch_1 has the same return value as the corresponding action invocation in Sch_2.

2.2 Processes

A process p is defined as $p = \langle AI_p, P_p, C_p, T_p \rangle$, where AI_p is a set of action invocations, P_p is a set of precedence constraints on AI_p, C_p is a set of consistency constraints on AI_p, and T_p is a set of timing constraints on AI_p. All constraints in a process are expressed on sets of action invocations.

Precedence Ordering. A process expresses two forms of precedence orderings for action invocations: *intra-process orderings* on the action invocations of AI_p, and *inter-process orderings* on the action invocations of AI_p relative to action invocations in other processes.

Process p's intra-process precedence ordering, $\prec_p$, is an irreflexive partial ordering on AI_p. That is, let a_i, a_j be action invocations of actions of resources r and s, respectively, and let Sch_r, Sch_s be schedules containing those action invocations (*i.e.*, $a_i \in H_r$ and $a_j \in H_s$). If $a_i, a_j \in AI_p$ such that $a_i \prec_p a_j$, then $complete_r(a_i) < start_s(a_j)$ (*i.e.*, a_i must complete executing before a_j starts executing). Since $\prec_p$ is a partial order, it may allow certain action invocations within the same process to execute concurrently. Using our example, the action invocations for the lifting process, P_{lift}, would be: $AI_{lift} = \{read_{arm1}, grasp_{arm1}, lift_{arm1}, read_{arm2}, grasp_{arm2}, lift_{arm2}\}$.
P_{lift}'s ordering, $\prec_{lift}$, could be defined follows:

$$\begin{array}{ccccc} read_{arm1} & \longrightarrow & grasp_{arm1} & \longrightarrow & lift_{arm1} \\ & \times & & \times & \\ read_{arm2} & \longrightarrow & grasp_{arm2} & \longrightarrow & lift_{arm2} \end{array}$$

In this ordering the *read* action invocation on arm1 must complete before each of the *grasp* action invocations and also before each of the *lift* action invocations start, but the two *read* action invocations may be concurrent (as may the two *grasp* action invocations and the two *lift* action invocations).

Inter-process precedence orderings are specified in a process by *sync sets*, each of which is a tuple, $\langle a_s, sig_s \rangle$. The action invocation $a_s \in AI_p$ is called a *sync_action*, and must be started after the *corresponding sig_action*, sig_s, has completed, where sig_s is an action invocation in a process other than p. For example, all action invocations in P_{lift} could be ordered as sync_actions after the same corresponding sig_action in another process which detects the object.

Consistency Constraints. Processes express two forms of consistency constraints typically found in transaction-based paradigms: exclusivity and atomicity of sets of actions [3].

Using the notion of conflict provided by the compatibility predicates of resources, processes can specify that a set of action invocations be executed exclusive of interruption from any conflicting action. That is, an *exclusive set* of action invocations must not have any of its execution overlap with the execution of an incompatible action that is not in the set. In our example, the sets of action invocations on each arm are exclusive sets since once the grasp and lift of the object by each arm has started, another process should not be allowed to move an arm in an incompatible way.

To ensure that consistency is not violated due to partial execution, atomicity is expressed by *atomic sets* of action invocations: Either all action invocations of an atomic set must be executed, or none of them must be executed. In the example, the two *lift* action invocations should be an atomic set to prevent one arm from lifting without the other.

Timing Constraints. Processes express three forms of timing constraints: absolute timing constraints, guaranteed execution, and simultaneous execution.

Absolute timing constraints are expressed by *temporal scopes*. A temporal scope is defined as $ts = \langle T, sa, sb, d \rangle$, where $T \subseteq AI_p$ is the set of action invocations to be time constrained, sa is an absolute earliest start time, sb is an absolute latest start time, d is an absolute lastest complete time (deadline) for the action invocations in T, and $sa \leq sb < d$. For instance, if action $a \in T$ is an action invocation of resource r then $sa \leq start_r(a) \leq sb$ and $complete_r(a) \leq d$.

To further constrain the timing of actions, a process may express *guaranteed sets* of action invocations. Each action invocation of a guaranteed set must execute at the earliest time that it is ready, where an action invocation is said to be *ready* at time t iff executing it at t meets precedence and absolute timing constraints. That is, the action invocations of a

guaranteed set must execute without delays caused by contention for resources. In the example, the two *lift* action invocations should be guaranteed. Assume it is known that the lifting will take t time without contention for resources, and that there is a deadline of d to complete the lifting. By including the *lift* action invocations in a guaranteed set and using a latest start time constraint of $sb = d - t$, P_{lift} can guarantee that the lifting starts *only if* it can meet its deadline.

A process may also express *simultaneous sets* of action invocations. The action invocations of a simultaneous set must start executing at the same time. In the example, the two *lift* action invocations should be simultaneous.

3 RTC Language Constructs

Our goal in developing language constructs for distributed real-time programming is to provide a small set of orthogonal constructs that naturally expresses the concepts developed in the model of Section 2. Since it is possible to specify requirements that are impossible or impractical to satisfy, we also provide exception handling capabilities to allow graceful recovery from run-time violations of the requirements.

The *RTC* language constructs consist of *resources*, *processes*, and *statements*. The precedence, consistency and timing constraints described in Section 2 are captured in ***block*** statements; processes request resources to perform actions using *action invocation* statements. We do not describe the exact syntax and semantics of each construct; instead, we describe the constructs using an outline of an *RTC* program for the robot lifting example. In the description of constructs, we pay particular attention to defining the *start* time, *complete* time and *ready* time of statements (see Section 2), since the model is ultimately concerned with precedence orderings and timing properties of programs.

3.1 Resources

The resource construct contains local data declarations, action declarations, and initialization statements. An action specifies parameters for exchanging information with its invoking process, as well as which actions of the resource are compatible with it (*i.e.*, may be overlapped with it). The body of an action is a sequence of host language, timing block, or no_except block statements. For simplicity, we do not allow actions to invoke other actions. In case the calling process aborts the action before it is completed, an exception handler may be used to specify the action's recovery. Details of timing blocks, no_except blocks, and exception handlers are discussed in Section 3.3.

```
resource Arm1
    C data structures for Cartesian coordinates
        and hand position
    action lift (parameters)
        compatible read, grasp;
        :   action body: C code for lifting.
        except /* Process aborts lift */
            when E_ABORT do
                :   C code for exception handling
            end when
    end action
    :   other action declarations (read, grasp, etc.)
    :   C code for arm calibration and initialization
end resource
```

Figure 1: Arm1 Resource in Lifting Program

Figure 1 shows how these constructs are used to specify resource *Arm1* in the lifting application.

3.2 Processes

An *RTC* process contains local data structure and procedure declarations, and a sequence of statements. In addition to host language statements, *RTC* statements include *action invocations* and *blocks* that capture the constraints described in Section 2.

Action Invocations Statements. An action invocation statement may be *synchronous*, denoted by:

action ⟨resourceID⟩.⟨actionID⟩ (⟨arguments⟩)

or *asynchronous* denoted by:

action& (⟨event⟩) ⟨resourceID⟩.⟨actionID⟩ (⟨arguments⟩).

With a synchronous action invocation statement, the calling process waits for the invoked action to complete; the calling process does *not* wait for an asynchronously invoked action to complete. Completion of an asynchronously invoked action may be detected using an *event variable* (see Section 3.3), which is signaled by the run-time system upon completion of the invoked action.

The start time of any action invocation statement is when the first primitive instruction for invoking the action starts executing. A synchronous action invocation statement's complete time is when the run-time system has been notified of the completion of the invoked action. An asynchronous invocation state-

ment's complete time is when the action invocation has been requested.

Block Statements. *RTC* provides *timing block, guaranteed block, simultaneous block, exclusive block* and *no_except block* statements. A block statement is a sequence of statements, and may have an associated exception handler. The start time of a block is the minimum of the start times of its enclosed statements; the complete time of a block is when the processor finishes executing the last primitive instruction in the block. The last primitive instruction may complete after the sequence of statements in the block have completed (*e.g.*, when the process executes the last primitive instruction to release locks used in the block). However, if an exception was raised, the last primitive instruction of a block may complete after the exception handling statements finish executing.

When an exception is raised, the process *aborts* the statements in the block for which the exception was raised. When a process aborts a block, the next statement in the block does not become ready; instead, the exception handler of the block becomes ready. The process aborts a block at the completion of the current primitive instruction, except in two cases: during a *no_except block* statement and when waiting for a synchronous action to complete. A no_except block, indicated by **no_except** – **end no_except**, delays all exceptions from timing blocks in which it appears until after the no_except block completes. If the exception occurs while the process is waiting for a synchronous action invocation to complete, the process aborts before the wait completes.

When a calling process aborts an action invocation statement, the system raises an E_ABORT exception in the invoked action invocation, the invoked action aborts its body (if it has not yet completed), and the statements of the invoked action's E_ABORT exception handler become ready.

Figure 2 shows the outline of *RTC* constructs for the lifting process of the two arm example; details of the block statements will be given in the next subsection.

```
event detected /* Global event */

process P_lift
   event read1, read2, grasp1, grasp2, lift1, lift2;
   ...other declarations.
   after detected by (detected+10sec) do
      exclusive
         action&(read1) Arm1.read (position);
         action&(read2) Arm2.read (position);
         after max(read1,read2)
         action&(grasp1) Arm1.grasp (position);
         action&(grasp2) Arm2.grasp (position);
         after max(grasp1,grasp2)
         before (detected+6sec) do
            guaranteed no_except simultaneous
               action&(lift1) Arm1.lift()
               action&(lift2) Arm2.lift()
            end simultaneous no_except
               guaranteed;
            after max(lift1,lift2)
         except /* start time violation */
            when E_START do stop application.
            end when
         end do
      end exclusive
   except /* detected + 10sec deadline violation */
      when E_DEADLINE do
         stop application, emergency actions.
      end when
   end do
end process
```

Figure 2: Two-arm lifting Example

3.3 Expression of Constraints

Timing Constraints. Temporal scope timing constraints are specified in a program using the *timing block* construct, which explicitly constrains the earliest start time, latest start time, maximum execution time, and completion time of statements in the block. The *timing expressions* used to express these constraints have operands of the following three types: *abs_time* for representing absolute time (*e.g.*, 10:00 am in EST); *rel_time* for representing relative time (*e.g.*, 10 seconds); and *event* for representing either absolute time or a special value called *DNO* (Did Not Occur). There is also a read-only global absolute time variable called *NOW* whose value is the current absolute time. Variables of type abs_time and rel_time may be declared in programs and are assigned values using host language assignment statement. Variables of type event may be declared in processes or may be global to all processes. Event values may be assigned in one of three ways: 1) A process executes a *signal* statement, which assigns to each event in its specified list the absolute time that the signal statement starts executing on a processor; 2) A process executes a *clear* statement, which resets the value of each event variable in its specified list to *DNO*; 3) The system "signals" an event variable associated with the completion of an asynchronous action invocation by assigning the event variable's value to the complete time of the action invocation. Until it is signaled, the value of the event variable is *DNO*. Timing expressions can be formed

using arithmetic operations, maximum functions, and minimum functions involving time values (see [4]).

A timing block can also provide exception handlers for latest start time, maximum execution, and completion time violations. If multiple exceptions are raised simultaneously in a timing block or an exception is raised while another exception handler of the same timing block is executing, a preemption ordering of E_START < E_EXECUTE < E_DEADLINE is used, where only a higher ordered exception handler can preempt a lower one.

In the example of Figure 2, the line:

after detected by (detected +10sec) **do**

is a timing block header that constrains the statements enclosed by it and its associated **end do** to start after the event *detected* is signaled, and to complete by 10 seconds after event *detected* is signaled. If the statements do not complete by the deadline, they are aborted and the associated E_DEADLINE exception handler becomes ready. This exception handler stops the application and takes emergency actions.

A second timing block is expressed by:

after max(grasp1,grasp2) **before** (detected+6sec) **do**.

This timing block constrains its enclosed statements to start executing after both events *grasp1* and *grasp2* have been signaled and before 6 seconds past the time that event *detected* was signaled. If the statements have not started by this latest start time, they are not started and the E_START exception handler becomes ready. Note that this second timing block is *nested* within the first timing block. This nesting causes the statements of the second timing block to be constrained by both timing blocks (*e.g.*, the deadline of the first timing block still applies in the second timing block). Nested blocks are discussed in more detail at the end of this section; the implementation of nested timing blocks is discussed in Section 4.2.

Timing blocks also provide two features not demonstrated in the example: the expression of start time, period, termination condition, and exception handling for periodic behavior; and the expression of maximum execution time (an **execute** clause and E_EXECUTE exception handler). The notion of maximum execution time is useful for supporting schedulability analysis [2].

To specify a guaranteed set timing constraint in a process, a *guaranteed block*, denoted by **guaranteed – end guaranteed**, is used. Once a guaranteed block starts, its enclosed sequence of statements must be executed as soon as they are ready. In addition, all action invocations requested in the guaranteed block must be executed on their processors as soon as they are ready, which is when the action invocation request is received by the run-time system. That is, no delays due to contention for resources may occur in the process or the actions that it invokes while it is in the guaranteed block. In the example of Figure 2, P_{lift} uses a guaranteed block to specify that once the two *lift* actions start, they may not be delayed by contention with other processes for use of the arms.

To specify a simultaneous set timing constraint in a process, a *simultaneous block*, denoted by **simultaneous – end simultaneous**, is used (see Figure 2). The action invocations of a simultaneous block are requested concurrently by the process and must be started within a bounded time from each other on their processors. This bound is a system-dependent interval called the *simultaneity bound*, ϵ (the simultaneity bound for our current implementation is discussed in Section 4.4).

Precedence Orderings. Intra-process precedence orderings are naturally supported by the sequential nature of statements, as well as by asynchronous action invocations and timing blocks. In the example of Figure 2, the two *grasp* actions are invoked concurrently as asynchronous action invocations with associated event variables *grasp1* and *grasp2* respectively. Since events *grasp1* and *grasp2* are signaled by the system when the *grasp* action invocations have completed, the second timing block ensures that both *lift* action invocations are executed after both *grasp* action invocations have finished. Using traditional concurrency terminology, a process "forks" asynchronous action invocations and uses timing block *after* clauses to "join" combinations of these action invocations at later points in its execution.

Inter-process precedence orderings are supported using global events and timing blocks. For example, the first timing block in Figure 2 specifies that all of its statements execute after the event *detected* has been signaled. We assume that another process (not shown in Figure 2) detects the container and then executes a *signal* statement on the global event variable *detected*. Therefore, all of process P_{lift}'s statements execute after the detection of the container.

Consistency Constraints. To specify exclusive set consistency constraints in a process, an *exclusive block*, denoted **exclusive – end exclusive**, is used. In the example of Figure 2, process P_{lift} uses an exclusive block to specify that once process P_{lift} starts using the arms, no incompatible actions may be executed on the arms by other processes until P_{lift} completes

lifting.

The notion of an atomic set in the model is supported by no_except blocks, timing blocks, and guaranteed blocks. No_except blocks ensure that the enclosed statements complete once they start by delaying timing exceptions until after the statements complete. To minimize the number of exceptions that are delayed, the atomic set of statements should be placed inside a guaranteed block as the first statement in a timing block. By using the timing block to constrain the latest start time of the guaranteed block to be "sufficiently far in advance" of the deadline, the guaranteed statements must complete by the deadline under normal operating conditions. For example, in Figure 2 we assume that the *lift* actions each take a maximum of 4 seconds including message delays when there is no contention for resources. The *before* clause is used to ensure that either the atomic set comprised of the *lift* actions starts within 4 seconds of its deadline, or its is not started and exception handling is performed. If the *lift* action invocations are started, the no_except block prevents abortion due to a deadline violation.

While this expression of "atomicity" is somewhat unconventional, the fact that real-time control applications directly affect the environment and are time-constrained makes traditional atomic rollback impossible. For example, if an action moves an arm from a starting position, a compensating action can bring it back to the starting position, but not erase the fact that the move was performed or that the move took time. Thus, to achieve atomicity in a real-time environment, we require that either all actions of the atomic set complete once they are started, or that none of them start.

Since *RTC* statements can be *RTC* blocks that themselves contain *RTC* statements, the syntax allows blocks to be nested. The semantics of nested blocks is a composition of the semantics of the individual blocks, thus allowing the expression of multiple constraints on parts of processes.

4 Implementation

In our implementation, a preprocessor translates programs written in C + *RTC* into C programs that interact with the operating environment and run-time system. The operating environment is a distributed collection of processors and devices (such as robot arms) that communicate asynchronously with each other via messages over a network. Each processor has a collection of *tasks*, which are executable code on that processor. A real-time kernel resides on each processor to perform services such as low-level resource allocation to tasks, message communication between tasks, and detection/notification of exceptions such as timing violations. The kernel allocates memory to each task, and only that task may access the local memory unless the task explicitly grants access to other tasks. The kernel determines when a task executes on its processor based on a *scheduling policy*. The scheduling policy assigns a dynamic priority to each task, and whenever possible allows the highest priority task to execute on each processor. Timing constraint information should be incorporated into the dynamic priority value to improve performance, although the implementation of the *RTC* constructs does not require it. We currently use *earliest-deadline-first scheduling* (EDF) of tasks, *i.e.*, we determine dynamic priority as a function of the deadline alone.

4.1 Run-Time Support for Timing Blocks

Timing blocks are implemented using a stack of temporal scopes for each process (or action invocation) to keep track of its current timing constraints. The timing constraints on the top of the stack are used by the kernel to set alarms and determine the scheduling priority of the process or action invocation. As nested timing blocks are entered during execution, the run-time system pushes modified timing constraints for that block onto the stack. That is, the run-time system compares the timing constraints specified by the block to those on the top of the stack, and pushes the "tighter" timing constraints. For instance, if the current deadline of a process is 10:00 and a nested timing block specifies a deadline of 11:00, the current deadline of 10:00 is pushed on the temporal scope stack; therefore, the process continues to operate under the 10:00 deadline. This adjustment of timing constraints is performed so that statements meet the timing constraints of all temporal scopes in which they appear.

When the kernel notifies a process (or action invocation) that a timing constraint was violated, the run-time system first aborts the current execution of the process (or action invocation). It then pops the temporal scope stack until the timing constraints of the timing block surrounding the violated timing block are on the top of the stack. These constraints are then used by the system as it executes the violated timing block's exception handler.

4.2 Run-Time Management

System resources are managed by tasks: a *resource manager task* (RMT) for each user-defined resource

r (RMT_r), and a *processor manager task* (PMT) for each processor p (PMT_p). Processes request access to system resources through the appropriate manager tasks. Requests to the manager tasks are arbitrated in order of the priorities of requesting processes.

Due to the nature of our consistency and timing constraints, however, a simple preemptive, priority-driven scheduling paradigm is not sufficient. For example, consider the use of preemptive priority-driven processor scheduling in the application of Figure 2: While P_{lift} is performing its lifting, a higher-priority process sharing a processor with P_{lift} may preempt P_{lift} to use Arm_1. If this happens, P_{lift}'s exclusive set, simultaneous set, and guaranteed set will be violated. Furthermore, the action requested by the higher-priority process may conflict with the action being performed by P_{lift}, violating the serializability of Arm_1.

We therefore add locks to arbitrate the use of resources: *Resource locks* are used to preserve the consistency of user-defined resources and to implement exclusive blocks; *processor locks* are used to guarantee execution to implement simultaneous and guaranteed blocks. Although processes may request resource locks directly from the RMTs, processor locks are indirectly requested from PMTs by including the request for a processor lock in a resource lock request to an RMT. The RMT then forwards the lock request to its associated processors. This indirection is used so that processes do not have to be aware of the mapping of processors to resources.

Resource Manger Tasks. RMT_r handles requests from processes to invoke actions on resource r, grant resource locks, and release resource locks. It also forwards processor lock requests from processes to the processors associated with r.

RMT_r grants an *action invocation* request for action a_i from process q if and only if a_i is compatible with r's currently executing action invocations, the actions for which resource locks are currently held, and pending requests (both action invocation and resource lock requests) of higher priority. If the request is granted, RMT_r creates a task, t_k, for a_i and grants t_k access to the data of r. If q holds a processor lock, RMT_r assigns t_k to the locked processor and assigns highest priority to t_k; otherwise, RMT_r assigns t_k to any one of the processors associated with r and assigns q's priority to t_k. Note that once t_k is assigned to a processor, it is scheduled by the kernel (based on its priority) so that RMT_r does not directly control how and when action invocation tasks access the data of r. However, because RMT_r checks compatibility before creating an action invocation task, the serializability of r's schedule is assured. If RMT_r determines that a_i does not meet the compatibility requirement, it queues the request based on q's priority. When an action invocation completes, RMT_r traverses the queue of pending action invocation and resource lock requests, in order of priority, and grants those requests that it can.

Process q directly requests a *resource lock* from RMT_r, by specifying the set of actions it wishes to invoke on r, $\{a_1, \ldots, a_n\}$. RMT_r grants the request only if $\{a_1, \ldots, a_n\}$ are compatible with r's active action invocations, the actions for which resource locks are currently held, and pending requests of higher priority. Thus, when RMT_r grants a resource lock to q, RMT_r guarantees that no action that is incompatible with any action in $\{a_1, \ldots, a_n\}$ will be executed while q holds the resource lock, providing an implementation for exclusive blocks. If RMT_r does not grant a resource lock request, it queues the request based on q's priority.

Process q may also include a processor lock request with the resource lock request. If such a request is received, RMT_r forwards the request to its associated PMTs. If some PMT grants q's request, the PMT notifies RMT_r who then informs q that a processor lock has been granted. Note that q does not need to know *which* processor has granted the lock, only that *some* processor associated with r has granted the lock. When q receives a processor lock, its action invocations from $\{a_1, \ldots, a_n\}$ will execute with the highest priority on the locked processor. This "immediate execution" is required to implement simultaneous blocks and guaranteed blocks.

When a process releases r's resource lock, RMT_r traverses its queue of pending action invocation and resource lock requests, in order of priority, and grants those requests that it can. When a process releases a processor lock, RMT_r notifies the appropriate PMT to release the lock.

Processor Manager Tasks. A PMT handles processor lock requests and releases that have been forwarded from RMTs. A PMT grants a forwarded request if and only if there is no processor lock currently held, or the requesting process is the same as the process specified in the currently held lock. Thus, while only one process may hold a lock on a given processor, forwarded requests from several RMT's may be satisfied by a single processor lock if the requests are on behalf of the same process. If the lock cannot be granted,

the PMT queues the request according to the priority of the requesting process. When an RMT notifies the PMT to release the processor lock, the PMT removes the RMT's request from a list of resources that requested the processor lock on behalf of the holding process. If the RMT is the last resource on the list, the PMT releases the processor lock and grants it to the pending request with the highest priority, req_1. The PMT also grants the processor lock to all pending requests that specify the same process as req_1.

4.3 Meeting Constraints

We now show how this run-time system ensures that the requirements of resource serializability, exclusive sets, simultaneous sets and guaranteed sets are met.

Serializability of each resource r is ensured by the fact that no incompatible actions can overlap in Sch_r since a RMT checks compatibility before executing an action invocation or granting a resource lock. If two actions a_i and a_j do overlap in Sch_r, then they must compatible; hence by the definition of compatibility (Section 2), their overlapped execution produces the same state and same return values as an execution in which a_i completes before a_j starts. Hence, Sch_r is serializable.

To ensure the guaranteed set requirement of the guaranteed block construct is met, it is sufficient for the run-time support of a process to obtain resource locks and an associated processor lock for all resources used in the guaranteed block before it requests any of the action invocations in the block. Each lock is held until all action invocations in the guaranteed block that use its resource have completed. The resource locks ensure that no action invocation of the guaranteed block is queued by its RMT. The processor locks ensure that the action invocations execute on their assigned processors when the action invocations are ready.

To ensure the exclusive set requirement of the exclusive block construct, it is sufficient for the run-time system to obtain resource locks for all resources used in the exclusive block before any of the action invocations in the block are requested. The resource locks must be held until all action invocations in the block have completed. The resource locks ensure that no action invocation is executed that is incompatible with any action invocation in the exclusive block. Since the locks are held for the entire exclusive block, exclusive sets are maintained.

To implement simultaneous blocks, we assume that the underlying system can give a reasonable worst case bound on the time between a task sending a message and the arrival of the message at the recipient task's message queue. We call this maximum message delivery time δ. Furthermore, we assume a worst case time bound of σ for the action invocation request to be processed by its RMT. Given these assumptions, it is sufficient for the run-time support of a process to obtain a resource lock and associated processor lock for all resources used in the block, and then to broadcast the action invocation requests simultaneously. The resource locks ensure that no action invocation in the block will be queued by a RMT. The processor locks ensure that the action invocation tasks start executing immediately on their processor. To check that the actions were started within the simultaneity bound, the process waits for replies from the RMTs indicating that the actions have started. These replies can take up to δ time to be delivered. The simultaneity bound for our implementation is therefore the time it takes for an action to be started $(\delta + \sigma)$ plus the maximum time it takes the acknowledgement of the start (δ) to be received by the process: $\epsilon = 2\delta + \sigma$. Since a simultaneous block only constrains the start of its action invocations, its locks may be released after the action invocation statements have started.

In each of the block implementations, a process must obtain a set of locks for the block. If processes obtain only some of their required locks while waiting for others, deadlock is possible. We present and prove a deadlock prevention technique for such systems in [4].

5 Conclusion

This paper has described the *RTC* language constructs and run-time system for distributed real-time programming. The constructs allow the explicit expression of real-time concurrency requirements: precedence orderings, absolute timing constraints, simultaneity, exclusiveness, atomicity, and recovery from constraint violations. Our run-time system uses real-time scheduling augmented with locking of resources and processors. This integrated scheduling of shared resources improves performance and predictability in distributed real-time applications. The use of an object and transaction based paradigm supports modularity and abstraction.

The *RTC* action and process constructs are based on the transaction model presented in [3] with several modifications. *RTC* actions are modified transactions that have their notion of conflict defined on the level of actions rather than on the level of read and write

operations [3]. Exclusivity and atomicity are decoupled and enforced on parts of an *RTC* process instead of all of it as is done with a transaction. Furthermore, *RTC* processes are not independent, they synchronize through inter-process precedence orderings. Finally, *RTC* processes are time constrained and transactions typically are not.

Although current real-time languages provide support for subsets of the required constraints described in Section 2, no current language provides support for all of them. An object-based paradigm with concurrency has been employed in ARTC++ [5]. Explicit timing constraints are provided in the temporal scope constructs of [6], Real-Time Euclid [2], Flex[1] and Maruti [7], among others. Temporal Scopes, Real-Time Euclid, and Flex also provide exception handling for constraint violations. The Spring kernel [8] provides guaranteed execution for entire processes rather than a set of actions. Exclusive and atomic sets as well as concurrency based on action-level compatibility are not directly supported in other current real-time languages.

Maruti [7] provides many of the real-time concurrency requirements described in Section 2. The biggest difference between their approach and ours is that Maruti assumes that everything can be prescheduled. On the other hand, we use priority-based run-time scheduling and exception handling to respond to dynamic environments in a more flexible manner. Due to their static approach to scheduling, Maruti does not provide exception handling capabilities or event-relative timing expression for temporal scopes, and has a more restrictive notion of precedence ordering.

The block-structured syntax of the *RTC* constructs allows them to be embedded in any procedural programming language. Since our current target application is robotics and most robotics software at the University of Pennsylvania is developed in C, we chose C for our original host language. We have implemented an *RTC* preprocessor and associated run-time system, and have used them to create the two-arm robot lifting program outlined in this paper [4]. The operating environment consists of three distributed MicroVax II processors that drive a graphic robotics simulation on an IRIS graphics workstation. The control program coordinates two graphic models of Puma 560 robot arms, which have a similar control interface to that of actual Puma 560 robot arms, to simulate picking up a moving object under timing constraints.

For more detail on the *RTC* constructs, their implementation, and this application, see [4]. We are currently improving error checking and fault-tolerance capabilities.

References

[1] K.-J. Lin and S. Natarajan, "Expressing and maintaining timing constraints in FLEX," in *Real-Time Systems Symposium*, pp. 96–105, IEEE Computer Society, 1988.

[2] E. Klingerman and A. Stoyenko, "Real-time Euclid: A language for reliable real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 941–949, Sept. 1986.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* New York: Addison Wesley, 1986.

[4] V. Wolfe, *Supporting Real-Time Concurrency.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1991. Available as Technical Report MS-CIS-91-55.

[5] Y. Ishikawa, H. Tokuda, and C. Mercer, "Object oriented real-time language design: Constructs for timing constraints," Tech. Rep. CMU-CS-90-111, Carnegie Mellon University, March 1990.

[6] I. Lee and V. Gehlot, "Language constructs for distributed real-time programming," in *Proc. IEEE Real-Time Systems Symposium*, IEEE Computer Society, Dec. 1985.

[7] V. Nirkhe, S. Tripathi, and A. Agrawala, "Language support for the Maruti real-time system," in *Real-Time Systems Symposium*, pp. 257–266, IEEE Computer Society, Dec. 1990.

[8] J. Stankovic and K. Ramamritham, "The Spring kernel: A new paradigm for real-time operating systems," *ACM Operating Systems Review*, vol. 23, pp. 54–71, July 1989.