January 1991

RTC: Language Support for Real-Time Concurrency

Victor Wolfe
University of Pennsylvania

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-35.

A shorter version of this paper can also be found at: http://repository.upenn.edu/cis_papers/350.

RTC: Language Support for Real-Time Concurrency

Abstract

This paper presents language constructs for the expression of timing and concurrency requirements in distributed real-time programs. Our programming paradigm combines an object-based paradigm for the specification of shared resources, and a distributed transaction-based paradigm for the specification of application processes. Resources provide abstract views of shared system entities, such as devices and data structures. Each resource has a state and defines a set of *actions* that can be invoked by processes to examine or change its state. A resource also specifies scheduling constraints on the execution of its actions to ensure the maintenance of its state's consistency. Processes access resources by invoking actions and express precedence, consistency. Processes access resources by invoking actions and express precedence, consistency and timing constraints on action invocations. The implementation of our language constructs with real-time scheduling and locking for concurrency control is also described.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-35.

RTC: Language Support For Real-Time Concurrency

MS-CIS-91-35
GRASP LAB 260

Victor Wolfe
Susan Davidson
Insup Lee

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

Revised
March 1992

RTC: Language Support For Real-Time Concurrency*

Victor Wolfe

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

Susan Davidson and Insup Lee

Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

January 1992

Abstract

This paper presents a model and language constructs for expressing timing and concurrency requirements in distributed real-time programs. Our approach combines an abstract data type paradigm for the specification of shared resources and a distributed transaction-based paradigm for the specification of application processes. Resources provide abstract views of shared system entities, such as devices and data structures. Each resource has a state and defines a set of *actions* that can be invoked by processes to examine or change its state. A resource also specifies scheduling constraints on the execution of its actions to ensure its consistency. Processes access resources by invoking actions and by expressing precedence, execution and timing constraints on action invocations. The implementation of our language constructs and the use of this system to control the simulation of a distributed robotics application is also described.

1 Introduction

In real-time applications such as robotics, industrial control and avionics, the correctness of a system depends on the satisfaction of timing constraints and maintenance of resource consistency constraints. However, techniques to enforce these two forms of constraints have not traditionally been integrated. For instance, priority-driven preemptive scheduling for tasks is used to meet the timing constraints of processes, but ignores the consistency constraints of resources. On the other hand, mutual exclusion techniques are used to ensure the consistency of resources, but typically ignore timing constraints. Although there have

*This work is support in part by the following grants: ARO DAAG-29-84-k-0061, ONR N00014-89-J-1131, and NSF CCR90-14621.

recently been efforts to combine the knowledge of timing constraints with consistent access to shared resources [1, 2, 3], the application domain has been restricted to real-time databases. This paper develops programming concepts and language constructs for expressing timing and resource consistency constraints as well as exception handlers for recovery from timing faults, and discusses their implementation.

As an example of a system with timing and resource consistency constraints, 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, which are functional consistency constraints and timing constraints imposed by the application. Many of these constraints are illustrated in the example, including start times and deadlines, exclusive execution, simultaneous execution, all-or-nothing execution, and predictable execution. Second, the run-time system should enforce the constraints. Third, the language should support the specification of exception handlers for recovery from timing faults, since some timing constraints may be violated at run-time. Fourth, since many real-time applications are distributed, the run-time system should be distributed.

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. These and other weaknesses of real-time languages are described in [4]; some of these deficiencies appear to be addressed in proposals for Ada9X [5].

Recent real-time languages such as Flex [6] and Real-time Euclid [7] allow explicit expression of some timing constraints. However, timing 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 often first-come, first-served; no timing information is used. Second, no concurrent access to resources is allowed, even if it does not violate their consistency. Hence, the run-time system may not be able to meet the stated timing constraints, although they could be met using other queuing techniques or techniques that allow concurrency.

Consistent concurrent execution is often supported by *transactions.* A particularly clear model of a transaction-based system can be found in [8], where a transaction is defined as a partial order of actions terminated by a *commit* or *abort.* A transaction is also defined to be *independent*: it does not directly communicate or synchronize with other transactions. A transaction's actions are typically understood as simple read and write operations, although other work has extended the notion to include more complex operations [8, 9]. A transaction-based system must guarantee that:

1. Each transaction accesses shared data without interference from other transactions;

2. If a transaction terminates normally, then *all* of its effects are realized; otherwise, *none* of its effects are realized.

Unfortunately, the traditional notion of a transaction does not include timing constraints. Also, since transactions are independent, we cannot specify the relative order of transactions. For instance, it impossible to specify that a transaction for lifting the container must always execute after a transaction for detecting 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 or raise an exception when they are violated. To define these constraints precisely, we develop a real-time concurrency paradigm that combines an abstract data type approach for the specification of shared resources with a distributed transaction-based approach for the specification of application processes. In addition we develop a notion of timing constraints. To express these constraints, we define the Real-Time Concurrency (*RTC*) language constructs. The constructs are designed to be embedded in C, although any block-structured procedural host language could be used. We then describe how to implement the constructs using an operating system with traditional dynamic priority-based CPU scheduling and the ability to block interrupts. Note that our system is designed for run-time enforcement of real-time concurrency constraints; we do not guarantee that all timing constraints will be met, but

allow the specification of recovery from timing faults. Hence we are not addressing *a priori* schedulability analysis [7, 10].

The rest of this paper is structured as follows. Section 2 presents requirements for real-time concurrent programming systems. We describe our paradigm for such a system including a basic set of constraints that should be expressed. Section 3 presents the *RTC* language constructs and illustrates how they can be used to express the constraints found in the robot lifting example; their implementation is discussed in Section 4. Section 5 describes related work that formed the basis for our system and compares our work to other real-time languages. Section 6 summarizes and discusses future work towards dynamic support of concurrent real-time programming.

2 The Real-Time Concurrency Model

Our paradigm for distributed real-time computing combines an abstract data type paradigm with a transaction-based paradigm and adds provisions for timing and precedence constraints using the notions of resources and processes. *Resources* provide abstract views of shared system entities such as devices and data structures. Each resource has a state and defines a set of *actions* that can be invoked by processes to examine or change the resource's state. A resource also specifies permissible overlapping of execution of its actions that preserves its state's consistency. *Processes* capture a transaction-based paradigm by specifying a set of action invocations along with precedence orderings, execution constraints, and timing constraints.

More formally, a system consists of a set of resources, $\mathcal{R}$, a set of processors, $\mathcal{P}$, and a set of processes, $\mathcal{Q}$. Resources provide actions, which are invoked by processes. The execution of an action is called an *action invocation*, and is characterized by a *start time*, a *complete time*, and a processor $p \in \mathcal{P}$ on which the action is executed. For action invocation a, the start time will be referred to as $start(a)$, the complete time as $complete(a)$, and the processor as $proc(a)$. Two action invocations a_1 and a_2 are said to be *overlapped* if $start(a_1) < complete(a_2)$, and $complete(a_1) > start(a_2)$. Action invocations may overlap either because they execute concurrently on different processors or because they are interleaved on the same processor. A process is characterized by a set of *behaviors*, each of which is a set of action invocations. Both resources and behaviors provide *constraints* on action invocations; resources constrain which actions can have overlapping invocations, and behaviors constrain the start times and complete times of action invocations. The *execution of a system* defines an instantiation of action invocations, i.e., for each action invocation a, an execution defines $start(a)$, $complete(a)$ and $proc(a)$ (where $start(a) \leq complete(a)$). If action invocation a

does not occur in an execution of a system then $start(a) = complete(a) = \infty$. The execution of a system is said to be *correct* if it satisfies all constraints expressed by resources, and it satisfies all constraints of some behavior for each process.

2.1 Resources

A resource, r, is characterized as $\langle S_r, A_r, C_r \rangle$, where S_r is a set of consistent states, A_r is a set of *actions*, and C_r is a compatibility relation on actions in A_r. An action invocation that overlaps with no other action invocations transforms any state in S_r to another state in S_r and returns values to the process. However, the state of the resource during the action invocation is not guaranteed to be consistent, i.e. its intermediate state may not be in S_r. The *compatibility relation* C_r is a symmetric relation on A_r that constrains which actions can have overlapping executions. That is, if $(a_1, a_2) \in C_r$ then invocations of a_1 and a_2 can be overlapped and will always result in a consistent state for resource r after their completion. Note that the designer of the system must ensure that C_r is defined correctly.

In our example, a robot arm is a resource. Part of its state information includes the Cartesian position of its arm and the position of its hand (grasp/ungrasp); consistent states are those that are within the reachable space of the arm and do not violate any of its mechanics. Actions of the arm 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 compatibility relation C_{arm} includes: $(lift, grasp)$, because the actions affect different parts of the state, and $(read, read)$, because the state is not affected. It does not contain $(lift, lower)$ because interleaving the executions of $lift$ and *lower* may not have the same effect on the state as executing them one after the other (e.g. executing $lift$ completely then executing *lower*); hence the state resulting from an interleaved execution may not be in S_{arm}.

2.2 Processes

A process is defined by its set of possible behaviors; each behavior has constraints on the time at which action invocations can occur in an execution. That is, each behavior b of a process q is defined as $\langle AI_b, \prec_b^l, \prec_b^g, XE_b, AE_b, AT_b, ST_b, GT_b \rangle$, where AI_b is a set of invocations of actions on resources in $\mathcal{R}$. The start and complete times of action invocations in b are constrained by precedence constraints ($\prec_b^l$ and $\prec_b^g$), execution constraints (XE_b and AE_b), and timing constraints (AT_b, ST_b, GT_b).

Precedence Constraints. A behavior b expresses two forms of precedence constraints for action invocations: local precedence constraints, $\prec_b^l$, and global precedence constraints, $\prec_b^g$.

Behavior b's *local precedence* constraints is an irreflexive partial ordering on AI_b. That is, if $a_i, a_j \in AI_b$ such that $a_i \prec_b^l a_j$, then either a_i completes before a_j starts, or a_i starts executing but a_j does not, or neither action invocation is executed ($complete(a_i) < start(a_j)$ or $complete(a_i) = start(a_j) = \infty$). Since $\prec_b^l$ is a partial order, it may allow certain action invocations to overlap. For instance, we could define a behavior b of the lifting process q_{lift} in our example by: $AI_b = \{read_{arm1}, grasp_{arm1}, lift_{arm1}, read_{arm2}, grasp_{arm2}, lift_{arm2}\}$. The precedence constraints of b could be defined follows:

$$\begin{array}{ccc} read_{arm1} & \longrightarrow & grasp_{arm1} \longrightarrow lift_{arm1} \\ & \searrow \quad \searrow & \\ read_{arm2} & \longrightarrow & grasp_{arm2} \longrightarrow lift_{arm2} \end{array}$$

(where "$\longrightarrow$" indicates "$\prec_b^l$"). In this behavior 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).

For a given process q, behavior b's *global precedence* constraints, $\prec_b^g$, express precedence orderings between action invocations in AI_b and action invocations in behaviors of other processes in $\mathcal{Q}$, i.e. it is a relation on $(AI_b \times AI_c) \cup (AI_c \times AI_b)$, where AI_c is a behavior of any process $q' \neq q$. For example, the robotics application which lifts a container could be extended to include another sensor process q_{sense} which monitors the conveyor belt to detect the next container. We might specify that the lifting of a container occur after its detection for each pair of possible behaviors of q_{sense} and q_{lift}, i.e. that all action invocations in q_{lift} used to lift the i'th container be ordered after the invocation of the i'th *detect* action in q_{sense}.

Execution Constraints. Two forms of execution constraints can be specified on a behavior b: exclusivity, XE_b; and all-or-nothing behavior, AE_b. Each constraint is a set of sets of action invocations from AI_b ($XE_b, AE_b \subseteq 2^{AI_b}$).

Using the notion of conflict provided by the compatibility relations of resources, an *exclusive execution* constraint specifies that sets of action invocations must be executed exclusive of interruption from any conflicting action. That is, no action invocation in set $x \in XE_b$ can have an overlapped execution with any action invocation that is incompatible with some element of x. In our example, for each behavior b of q_{lift}, XE_b contains the sets $\{grasp_{arm1}, lift_{arm1}\}$ and $\{grasp_{arm2}, lift_{arm2}\}$ since once the grasp and lift of the container by each arm has started, another process should not be allowed to move an arm.

Behaviors can also specify that sets of action invocations have an *all-or-nothing execution*. That is, if $s \in AE_b$ then either every action invocation in s was executed (for every $a \in s$,

$complete(a) \neq \infty$), or no action invocation in s was executed (for every $a \in s$, $start(a) = \infty$). In the example, the two *lift* action invocations should be constrained to have all-or-nothing execution to prevent one arm from lifting without the other.

Timing Constraints. A behavior b can specify three forms of timing constraints: absolute timing constraints, AT_b; simultaneity constraints, ST_b; and guaranteed constraints, GT_b.

Absolute timing constraints are expressed by a set of *temporal scopes*. A temporal scope $ts \in AT_b$ is defined as $ts = \langle A, sa, sb, d \rangle$, where $A \subseteq AI_b$ 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 latest complete time (deadline) for the action invocations in A, and $sa \leq sb < d$. That is, for $a \in A$, $sa \leq start(a) \leq sb$ and $complete(a) \leq d$.

Simultaneity constraints are a set of sets of action invocations ($ST_b \subseteq 2^{AI_b}$), constraining certain action invocations to start executing at the same time. That is, if $s \in ST_b$ then for each pair of action invocations $a_i, a_j \in s$, $start(a_i) = start(a_j)$. In the example, $\{lift_{arm1}, lift_{arm2}\}$ is a simultaneous set for each behavior of q_{lift}.

Guaranteed constraints are a set of action invocations ($GT_b \subseteq AI_b$), where each action invocation in the set must execute at the earliest time that it is ready. An action invocation is said to be *ready* at time t iff executing it at t meets all precedence and absolute timing constraints. More precisely, an action invocation a is ready at time t if all precedence constraints have been met ($t > max\{complete(c) |\ c \prec_b^l a\ \}$ and $t > max\{complete(c) |\ c \prec_b^g a\ \}$), all absolute timing constraints have been met ($t > max\{\ sa_i(c) |\ ts_i \in AT_b \wedge c \in A_i\ \}$), and all simultaneous execution constraints have been met (for every action invocation c such that $\{a, c\} \subseteq s \in ST_b$, the precedence and absolute timing constraints of c have been met). Furthermore, a must continue to execute without interruption until it is completed. That is, no other action c may execute on $proc(a)$ between $start(a)$ and $complete(a)$.

In the example, the two *lift* action invocations should be guaranteed in each behavior of q_{lift}. Assume it is known that the lifting will take τ time without contention for resources and there is a deadline of d to complete the lifting. By including the *lift* action invocations in the set of guaranteed action invocations and by using a latest start time constraint of $sb = d - \tau$, each behavior of q_{lift} will force the lift of each arm to start *only if* it can meet its deadline.

Inter-resource Constraints. Note that consistency constraints cannot span multiple resources. In our example, if each arm is its own resource we cannot explicitly represent the consistency constraint that the arms should not collide (i.e. the Cartesian coordinate of *arm1* and *arm2* must be disjoint). If we want to specify such a constraint, then *arm1* and

arm2 must be a single resource; the move actions of each arm must be designed to check the Cartesian coordinates of the other arm, and the move action of each arm be specified as incompatible with the other.

3 Language Constructs

The *RTC* language constructs are embedded in the C language, and consist of a small set of orthogonal primitives that naturally and explicitly express the constraints of the *RTC* model. In Section 4, we describe a run-time system that ensures correct executions of the system. However, since users may specify unsatisfiable timing constraints, or failures may cause timing constraints to be violated, we include timing-constraint exception handling in the language constructs so that users can express various forms of recovery, including compensating actions [11, 12, 13], imprecise computations [6], or other forms of roll-back or roll-forward techniques.

The *RTC* language constructs consist of *resources*, *processes*, and *statements*. The precedence, execution, and timing constraints described in Section 2 are captured in *block* statements. Processes request resource 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 a *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 since the model is ultimately concerned with precedence orderings and timing properties of programs.

An outline of the syntax of *RTC* constructs appears in Figure 1, where ⟨ ⟩ indicates non-terminal symbols, bold face indicates terminal symbols, | indicates alternatives, and [ ] indicates optional syntax.

3.1 Resources

The resource construct contains local data and procedure declarations, action declarations, and initialization statements. The local declarations and initialization statements are written in C. An action declaration 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 C language statements, as well as signal, clear, timing block, and no_except block statements. For simplicity, we do not allow actions to invoke other actions; the extension of the model and language to allow actions to invoke other actions is discussed as future work. In case the calling process aborts the action before it is completed, an exception handler should be used to specify the action's

⟨resource⟩ ::= **resource** ⟨ident⟩ [⟨host lang. declarations⟩]
[⟨actions⟩] [⟨host lang. stmts.⟩] **end resource**

⟨action⟩ ::= **action** ⟨ident⟩ ([⟨parameters⟩]) [⟨host lang. declarations⟩]
[**compatible** ⟨ident_list⟩] ⟨action statements⟩ **end action**

⟨action statement⟩ ::= ⟨host lang. stmt.⟩ | ⟨signal⟩ | ⟨clear⟩ | ⟨timing block⟩ |
⟨no_except block⟩

⟨process⟩ ::= **process** ⟨ident⟩ [⟨host lang declarations⟩]] ⟨statements⟩
end process

⟨statement⟩ ::= ⟨host lang. stmt.⟩ | ⟨action invocation⟩ | ⟨signal⟩ | ⟨clear⟩ |
⟨block⟩

⟨action invocation⟩ ::= **action** ⟨actionID⟩.⟨resourceID⟩ ([⟨arguments⟩]) |
action& [(⟨event⟩)] ⟨resourceID⟩.⟨actionID⟩ ([⟨arguments⟩])

⟨signal⟩ ::= **signal** (⟨event_list⟩)

⟨clear⟩ ::= **clear** (⟨event_list⟩)

⟨block⟩ ::= ⟨timing block⟩ | ⟨guaranteed block⟩ | ⟨exclusive block⟩ |
⟨no_except block⟩ | ⟨simultaneous block⟩

⟨timing block⟩ ::= ⟨tb_head⟩ ⟨statements⟩ [⟨tb_ehandler⟩] **end do**

⟨tb_head⟩ ::= [**after** ⟨abs_expr⟩] [**before** ⟨abs_expr⟩] [**execute** ⟨rel_expr⟩]
[**by** ⟨abs_expr⟩] **do** |
[**from** ⟨abs_expr⟩] **every** ⟨rel_expr⟩ [**while** ⟨condition⟩] **do**

⟨tb_ehandler⟩ ::= **except**
[**when** E_START **do** ⟨statements⟩ **end when**]
[**when** E_EXECUTE **do** ⟨statements⟩ **end when**]
[**when** E_DEADLINE **do** ⟨statements⟩ **end when**]

⟨abs_expr⟩ ::= ⟨abs_value⟩ | ⟨abs_expr⟩ + ⟨rel_expr⟩ |
⟨abs_fun⟩ (⟨abs_expr_list⟩)

⟨abs_value⟩ ::= ⟨abs_constant⟩ | ⟨abs_variable⟩ | ⟨event⟩

⟨rel_expr⟩ ::= ⟨rel_constant⟩ | ⟨rel_variable⟩ | (⟨rel_expr⟩) |
⟨rel_expr⟩ ⟨rel_op⟩ ⟨rel_expr⟩

⟨rel_op⟩ ::= + | − | * | /

⟨abs_fun⟩ ::= **max** | **min**

⟨guaranteed block ⟩ ::= **guaranteed** ⟨statements⟩
[**except when** E_GUARANTEED **do**
⟨statements⟩ **end when**]
end guaranteed

⟨simultaneous block ⟩ ::= **simultaneous by** ⟨abs_expr⟩ ⟨action invocations⟩
[**except when** E_SIMULTANEOUS **do**
⟨statements⟩ **end when**]
end simultaneous

⟨exclusive block ⟩ ::= **exclusive** ⟨statements⟩ **end exclusive**

⟨no_except block ⟩ ::= **no_except** ⟨statements⟩ **end no_except**

Figure 1: Outline of *RTC* Syntax

```
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 (stop arm, etc.)
            end when
    end action
    ⋮   other action declarations (lower, read, grasp, etc.)
    ⋮   C code for arm calibration and initialization
end resource
```

Figure 2: Arm1 Resource in Lifting Program

recovery. Details of these statements and exception handlers are discussed in Section 3.3

Figure 2 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 that are written in the C language, and a sequence of statements. In addition to C language statements, processes include *RTC* statements for *action invocations* and *blocks* that capture the constraints described in Section 2.

3.2.1 Action Invocation Statements

An action invocation statement may be *synchronous*, denoted by:

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

or *asynchronous* denoted by:

action& (⟨event_variable⟩) ⟨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 can be indicated by 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 requesting the action invocation starts executing. A synchronous action invocation statement's complete time is when the process has been notified of the completion of the action invocation. An asynchronous invocation statement's complete time is when the action invocation has been requested.

3.2.2 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 that 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. If no exception is raised, the last primitive instruction completes 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); otherwise, the last primitive instruction completes 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 (indicated by **no_except – end no_except**) and when waiting for a synchronous action to complete. In the first case, all exceptions from timing blocks in which the no_except block appears are delayed until after the no_except block completes. In the second case, the process aborts without waiting for the synchronous action invocation to complete.

When a calling process aborts an action invocation statement, the system raises an E_ABORT exception in the invoked action, 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 3 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 section.

3.3 Expression of Constraints

Absolute Timing Constraints. Absolute 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

```
event detected /* Global event signaled by another process */
process Q_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 by (1sec)
                    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 3: Two-arm lifting Example

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 infinite absolute time 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 C language assignment statement. Variables of type *event* may be declared in processes and actions, or may be global to all processes. Event values may be assigned in one of three ways:

- A process executes a *signal* statement, which assigns the absolute time that the signal statement starts executing on a processor to each event in the signal statement's specified list.
- A process executes a *clear* statement, which resets the value of each event variable in the clear statement's specified list to *DNO*;
- 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 Figure 1).

A timing block constrains the start time of its statements to be after the absolute time specified by **after** ⟨abs_expr⟩. If some statement in the timing block has not started by the time specified by **before** ⟨abs_expr⟩, the statements of the timing block are not started, and the E_START exception handler becomes ready. If the statements of the timing block or E_START exception handler execute on a processor for longer than the relative time specified by **execute** ⟨rel_expr⟩, the current execution is aborted, and the E_EXECUTE exception handler becomes ready. If the timing block has not completed by the time specified in **by** ⟨abs_time⟩, then the current execution is aborted and the E_DEADLINE exception handler becomes ready. Note that these semantics imply that if exceptions occur simultaneously, their precedence is E_START < E_EXECUTE < E_DEADLINE. Also note that many parts of a timing block are optional (see Figure 1).

To specify periodic behavior, a timing block establishes a series of time intervals called *period frames*, where the beginning of period frame i is the end of period frame $i - 1$. The statements of the timing block become ready at the beginning of each frame. If the

statements have not completed by the end of a period frame, then the system aborts the statements and the E_DEADLINE exception handler becomes ready. The first period frame begins at the time specified by **start** ⟨abs_time⟩. The duration of each period frame is the relative time specified by **every** ⟨rel_expr⟩. Negative durations result in an E_DEADLINE exception for the first frame. If there are no exceptions, the periodic timing block completes at the completion of the first frame during which the boolean expression specified in **while** ⟨condition⟩ evaluates to *FALSE*. Note that in addition to C language relational predicates, the terminating condition, ⟨condition⟩, may contain relational predicates involving ⟨timing_expr⟩ values to determine the number of period frames generated.

RTC syntax restricts timing blocks to be either nested or disjoint; that is, a statement may appear in two timing blocks only if the two timing blocks are nested. Timing blocks can be nested by placing a timing block in another timing block's body or exception handler.

In the example of Figure 3, the line:

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

is a timing block header that constrains its statements 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, causing 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).

Guaranteed Constraints. To specify a guaranteed 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

3, Q_{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.

Simultaneity Constraints. Simultaneous execution of statements is indicated by a *simultaneous block*, denoted by **simultaneous by** t **– end simultaneous**. The action invocations of a simultaneous block are requested concurrently by the process and must be started within the stated time t from the time at which the simultaneous block is entered. If the actions are not started within t, perhaps due to contention for the resource or its processors or due to a fault, the action invocation statements are aborted and an E_SIMULTANEOUS exception handler becomes ready. In the example of Figure 3, a simultaneous block is used to specify that the two *lift* actions start within 1 second.

Local Precedence Constraints. Local precedence orderings are naturally specified by the sequential composition operator (";" in C), as well as by asynchronous action invocations and timing blocks. In the example of Figure 3, the two *grasp* actions are invoked as (concurrent) 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's **after** construct specifies that both *lift* action invocations are to be executed after both *grasp* action invocations have finished. Using traditional concurrency terminology, a process "forks" asynchronous action invocations and uses *after* clauses of a timing block to "join" combinations of these action invocations at later points in its execution.

Global Precedence Constraints. Global precedence orderings are specified using global events and timing blocks. For example, the first timing block in Figure 3 specifies that all of its statements execute after the event *detected* has been signaled. We assume that another process Q_{sense} (not shown in Figure 3) detects the container and then executes a *signal* statement on the global event variable *detected*. Therefore, all of process Q_{lift}'s statements execute after the detection of the container by Q_{sense}.

Exclusive Execution Constraints. Exclusive execution constraints are indicated by an *exclusive block*, denoted **exclusive – end exclusive**. The set of exclusive action invocations specified by the exclusive block is comprised of all of the action invocations in the block. Therefore, after the exclusive block starts and before it completes, no action that is incompatible with any action invocation in the exclusive block may be executed by another process. In the example of Figure 3, process Q_{lift} uses an exclusive block to specify that

once process Q_{lift} starts using the arms, no incompatible actions may be executed on the arms by other processes until Q_{lift} completes lifting.

All-or-Nothing Execution Constraints. To specify that all statements in a block complete, a no_except block can be used to delay exceptions until after the statements complete. Specifying the "nothing" alternative involves ensuring that no actions are executed if exceptions are possible during the no_except block. This is done by nesting the no_except block inside a guaranteed block as the first statement of a timing block. The timing block specifies a latest start time that is sufficiently far in advance of the deadline to allow the statements to complete under normal operating conditions. Note that the programmer must know the maximum execution time of the statements to be guaranteed in order to establish this latest start time. For example, in Figure 3 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 the *lift* actions start prior to 4 seconds before their deadline, or they are not started and the E_START exception is handled. 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 [8, 14] impossible. For example, if an action moves an arm from a starting position, a compensating action [13, 12] 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 of the constrained statements complete once they are started, or that none of them start.

Multiple Constraints. Multiple constraints are expressed by nesting blocks. 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. If exceptions are raised simultaneously in several nested blocks, only the "outermost" violated block handles the exception. For example, in Figure 3, the two *lift* action invocations are placed in a simultaneous block, nested within a no_except block, nested within a guaranteed block. These nested blocks specify that the two *lift* action invocations must start at the same time, and once started they may not be interrupted by other action invocations or by timing constraint violations. These nested blocks also appear within an exclusive block, so that even if one lift finishes before the other, no incompatible action, such as another movement of the arm, may execute until the other lift finishes. Furthermore, these statements are nested in the two timing blocks that specify their earliest start time, latest start time, and deadline.

4 Run-Time System and 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 are connected by a network. The *TimixV2* real-time kernel [15] resides on each processor to perform services including thread management, low-level device management, asynchronous message communication between threads, and signaling of errors and alarms. The kernel provides a dynamic, priority-based scheduler for threads; currently, the dynamic priority is computed as a function of the deadline alone. In general, timing constraint information should be incorporated into the dynamic priority value to improve performance, although *RTC* run-time system does not require it. Currently, the operating environment consists of MicroVax II computers connected via an Ethernet.

The *RTC* run-time system consists of a set of *manager tasks*, each of which is implemented using one or more threads. Each *RTC* process is managed by its own *process manager task* (QMT). Each *RTC* resource may be distributed, but has a single *resource manager task* (RMT); that is, each resource is assumed to have an associated *set* of processors on which its actions may be executed, although access to actions is controlled by a single task. Each processor in the system has a *processor manager task* (PMT) which is used to reserve processors on behalf of processes for guaranteed executions of actions. There is also a centralized *event manager task* (EMT) in the system which interacts with process and resource managers to implement global *RTC* events.

4.1 Run-Time Support for Timing Blocks

To enforce the **after** construct of a timing block in process q, the process manager task for q, QMT_q, suspends q and sets an alarm for the earliest start time. When the alarm signal arrives, process manager QMT_q re-activates process q. To enforce the **before** construct of a timing block in process q, QMT_q sets an alarm for the latest start time. If the alarm is signaled before process q executes the statments of the timing block, process manager QMT_q causes process q to jump to the timing block's E_START exception handler. Otherwise, process manager QMT_q removes the alarm and process q continues to execute the statements in the timing block.

The **by** and **execute** constructs of a timing block in process q are implemented using a stack of temporal scopes to keep track of current timing constraints. Process manager QMT_q notifies the real-time kernel of the deadline and execution timing constraints on the top of the stack. The kernel uses these constraints to determine the scheduling priority of the process,

and to notify QMT_q of deadline and execution time violations. As nested timing blocks are entered during process q's execution, process manager QMT_q pushes modified timing constraints for that block onto the stack. That is, QMT_q 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 o'clock and a nested timing block specifies a deadline of 11:00 o'clock, the current deadline of 10:00 o'clock is pushed on the temporal scope stack; therefore, the process continues to operate under the 10:00 o'clock deadline. This adjustment of timing constraints is performed so that statements meet the timing constraints of all temporal scopes in which they appear. When process q completes all statements in a timing block, QMT_q pops the timing block's temporal scope from the stack.

When the kernel signals process manager QMT_q that a deadline or execution timing constraint was violated, QMT_q first pops the temporal scope stack until the timing constraints of the timing block that surrounds the violated timing block are on the top of the stack. QMT_q then causes process q to jump to the violated timing block's exception handler. The new constraints on the top of the temporal scope stack are used by the kernel as process q executes the exception handler.

Although we described the timing block implementation only for processes, a similar implementation is used for action invocations.

4.2 Resource and Processor Management

To ensure correct execution of an action, we need the ability to block incompatible action invocations. To guarantee the timely execution of an action, we need the ability to reserve execution time on an associated processor. We therefore use exclusive locks (without pre-emption) at both the resource and processor level: At the resource level, if an action is locked, then no actions that are incompatible with the locked action may be executed until the lock is released. At the processor level, if a processor is locked on behalf of a process q, then an action invocation of q will be executed as soon as it is requested, pre-empting any other activity on the processor. Since execution is guaranteed, a lock for a given processor may be held on behalf of at most one process. Note that there are many action locks for a given resource, but that there is a single processor lock per processor.

A process manager task requests *action locks* from the appropriate RMT; if it also requires a *processor lock*, then the RMT forwards the request to the associated PMTs on behalf of the process. Pending requests to RMTs and PMTs are served in order of the priority of the requesting process. This indirection is used so that processes do not have to be aware of the association between processors and resources.

4.2.1 Resource Manager Tasks

A resource manager RMT_r for resource r gets requests from process managers to acquire action locks, acquire a processor lock, invoke actions, and release locks. An action can either be invoked directly, or first locked and then later invoked. The latter is done to guarantee execution of an action.

Acquiring Action Locks. Process manager QMT_q requests action locks from resource manager RMT_r by specifying the set of actions that process q wishes to lock on resource r, $\{a_1, \ldots, a_n\}$. RMT_r grants the request only if each action in $\{a_1, \ldots, a_n\}$ is compatible with resource r's currently held action locks and pending requests of higher priority. If RMT_r does not grant a resource lock request, it queues the request based on process q's priority.

Acquiring a Processor Lock. Process manager QMT_q may also include a processor lock request with the action lock requests. If such a request is received, RMT_r forwards the request to its associated PMTs. If some PMT grants process manager QMT_q's request, the PMT notifies RMT_r, which then informs QMT_q that a processor lock has been granted. Note that process manager QMT_q does not need to know *which* PMT has granted the lock, only that *some* processor associated with resource r has been locked. When process manager QMT_q receives a processor lock, process q's 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 guaranteed blocks.

Invoking Actions. When RMT_r gets an action invocation request a from QMT_q that has not been locked, it must first lock the action; when the invocation completes, it must also unlock the action. Once a lock is held for the action, RMT_r creates a thread, t, for a and grants t access to the data of resource r. If process manager QMT_q holds a processor lock, RMT_r assigns thread t to the locked processor and assigns highest priority to t; otherwise, RMT_r assigns t to any one of the processors associated with resource r and gives the requesting process q's priority to t.

If process q's action invocation is synchronous, process manager QMT_q suspends q while waiting for return parameters; if the action invocation is asynchronous, q is not suspended. When an action invocation completes, the resource manager forwards the action's return parameters to process manager QMT_q in a message. Process manager QMT_q accepts the returned parameters and updates their values in process q's local state.

The QMT requesting an action invocation may also request an acknowledgement on the start of the action, as will be discussed under simultaneous blocks in the next subsection. In

this case, the thread for the action invocation will send an acknowledgement to the requesting QMT when it starts executing.

Releasing Locks. When a process manager releases an action lock for r, RMT_r grants action locks to the highest priority set of non-conflicting action lock requests. When a process manager releases a processor lock, RMT_r notifies the appropriate PMT to release the lock.

4.2.2 Processor Manager Tasks

A PMT handles processor lock requests and releases for guaranteed execution that have been forwarded from RMTs. If there is no processor lock currently held, the PMT grants the request and sets a counter of the number of lock requests for this process to 1. Otherwise, it grants the lock and increments the counter only if the requesting process is the same as the process who is currently holding the 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 by the same process. If the lock cannot be granted, the PMT queues the request. When an RMT notifies the PMT to release the processor lock, the PMT decrements the counter for the processor lock; when the counter becomes 0, the PMT releases the processor lock and grants it to the pending request with the highest priority. The PMT also grants the processor lock to all pending requests that originate from the same process as the one to whom the lock was granted.

Since a process may hold the same processor lock for two different actions, it is possible that the actions will be invoked at the same time. Thus, even though the associated RMTs forward the requests with high priority, at most one of the action invocations can execute on the processor, violating the notion of guaranteed execution. We must therefore ensure that these conflicts are detected and that the QMT for the requesting process is notified so that a E_GUARANTEED exception can be raised. Since at most one process can hold the processor lock, and action invocations are given high priority only if the requesting process holds the processor lock, this can be detected by checking if a request for execution of a high priority action is received while another high priority action is being executed.

4.3 Meeting Constraints

We now discuss how a *RTC* program and its run-time system implement the remaining RTC language constructs: simultaneous blocks, exclusive blocks, guaranteed blocks and no_except blocks.

Simultaneous Blocks. To implement the simultaneous execution of action invocations in a simultaneous block with bound t, the QMT must measure the delay between the time at which the first action invocation request is sent to an RMT, and the time at which the last acknowledgement that an action invocation was started is received. If the delay is greater than t, then the QMT cannot be sure that they were started within t of each other and an E_SIMULTANEOUS exception is raised. Note that is also possible that they were started in time, but that the acknowledgements were not received in time.

Note that the value of t will affect the likelihood of success. If it is less than $2\delta + \rho$, where δ is the minimum message delay time and ρ is the minimum time for the action invocation request to be processed by its RMT, the QMT will always raise an exception. A more reasonable choice is to use the average message delay time and average action invocation time. Note that even if worst case times are used, the exception can still be raised due to contention on resources. Therefore, if the user wants to improve the likelihood of successful execution of the simultaneous block, it should be enclosed in a guaranteed block.

Exclusive Blocks. To ensure exclusive execution, the QMT must obtain action locks for all action invocations in the exclusive block before the process executes any action invocations in the block. The action locks must be held until all action invocations in the block have completed. In this way, no action invocation that is incompatible with any action invocation in the exclusive block will overlap the execution of the block.

Guaranteed Blocks. To ensure guaranteed execution, the QMT must obtain action locks and an associated processor lock for all actions invoked in the guaranteed block before it invokes any action in the block. All locks are released when the guaranteed block completes. The action locks ensure that no action invocation of the guaranteed block is queued by an RMT. The processor locks ensure that the action invocations execute on their assigned processors when the action invocations are ready and that the action invocations are not preempted. However, a process may attempt to execute two guaranteed actions on the same processor at the same time, in which case the PMT of the processor will signal an E_GUARANTEE exception and the guaranteed block will be aborted. We could improve this method for detecting guaranteed block exceptions by attempting a priori determination of which guaranteed actions invocations will be concurrent and then trying to assign them to different processors. However, such solutions appear to be too complicated to justify the marginal improvement.

No_except Blocks. No_except blocks are implemented by the QMT delaying all exception signals until after block. The capability to delay signals is directly provided by the *TimixV2* real-time kernel as well as many other operating systems, including Real-Time Unix [16].

Deadlock Prevention. 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. Deadlock detection and recovery techniques involve preempting resources or aborting process, which make the execution time of processes unpredictable. We therefore feel that these techniques are not suitable for many real-time applications, and use deadlock prevention. Our prevention scheme is an example of the AND-OR request model [17], since an action lock must be acquired for each action invocation, as well as a processor lock on some associated processor; requests for action and processor locks must be acquired in a certain order. To obtain this ordering, processors are first ordered arbitrarily; actions are then ordered based on their highest associated processor. The algorithm and proof of correctness can be found in [18].

4.4 Performance

To determine the overhead imposed by the *RTC* run-time system in our current implementation, we measured the amount of time required to perform some of its basic execution. The results are shown in Figure 4. For these measurements, we used a Codar Technology clock/timer device that has a 4MHz crystal and allows measurements of durations with accuracy on the order of micro-seconds. These measurements were made on MicroVax II nodes controlled by the *TimixV2* real-time kernel [15] and connected by an Ethernet.

Due to space limitations, we discuss only the action invocation case in the performance measurements shown in Figure 4. Lines 1 and 2 show the amount of time from the start of a synchronous (null) action invocation to its return. Line 1 represents the case where the calling process and resource are on the same machine and Line 2 represents the case for them being on different machines. The approximately 8ms difference in the times represents the message delays due to the network communication. However, the estimated message delay on the *TimixV2* MicroVax II system is 9ms [15]. Using this value one would expect the action invocation time on different machines to be at least 18ms more than its time on the same machine (a message delay for the request and reply). The increase of only 8ms is due to concurrent execution of the process and RMT that reside on the two machines. A completed action sends two messages: a return message to the calling process, and a completion message to the RMT so that the RMT can service queued requests. Let A be the time required for the RMT to update its book-keeping information upon the return of an

	Execution	avg. (ms)	min. (ms)	max. (ms)
	action invocations			
1	action invocation round trip, same machine	73.667	67.844	75.856
2	action invocation round trip, diff machine	81.752	77.500	89.708
3	action invocation before sending	10.004	9.996	10.011
4	RMT time to start action (ρ)	45.680	44.340	47.660
5	Asynchronous action return (async thread)	1.147	1.110	1.245
	locking			
6	Alock request round trip, same machine	17.340	17.240	17.340
7	Alock request round trip, diff. machines	37.060	36.992	37.164
8	Alock release, same machine	14.712	14.464	14.832
9	Alock release, different machine	16.166	15.668	19.012
10	Plock request round trip, same machine	22.477	22.376	22.596
11	Plock request round trip, diff. machines	42.477	42.373	42.586
	timing blocks			
12	setting deadline (push)	1.676	1.672	1.684
13	pop temporal scope	2.137	2.132	2.152
14	event query/set/clear, same machine	5.261	5.232	5.892
15	event query with response, same machine	7.557	7.492	8.276
16	event query/set/clear, diff. machine	7.981	7.968	7.992
17	event query with response, diff. machine	27.527	27.288	29.684

Figure 4: Performance Measurements For the *RTC* Run-time System

action, and B be the time required for a process manager to receive the return message and update its arguments. In the case of two machines, A and B are performed in parallel when the respective completion and return messages are received; in the case of the same machine, A is performed before B, which extends the time that the action invocation completes. The execution of A, accounts for the approximate 10ms difference between the expected value and the observed value of action invocation times.

In general, these performance measurements show a disadvantage of our current run-time system: the high overhead imposed by the checking done by managers. However, we believe that in many applications the improved concurrency and use of real-time scheduling for passive resources will make up for performance lost due to the overhead. The *RTC* overhead can also be significantly reduced by using a faster processor than the MicroVax II.

4.5 Application

We have used the *RTC* constructs and run-time system to program control of a graphic simulation of the two arm lifting application. In the application, graphic models of Puma560 robot arms are controlled by a distributed *C + RTC* program executing on a three *Timix V2* Micro Vax II processors. In addition to the lifting process that was described in Section 3,

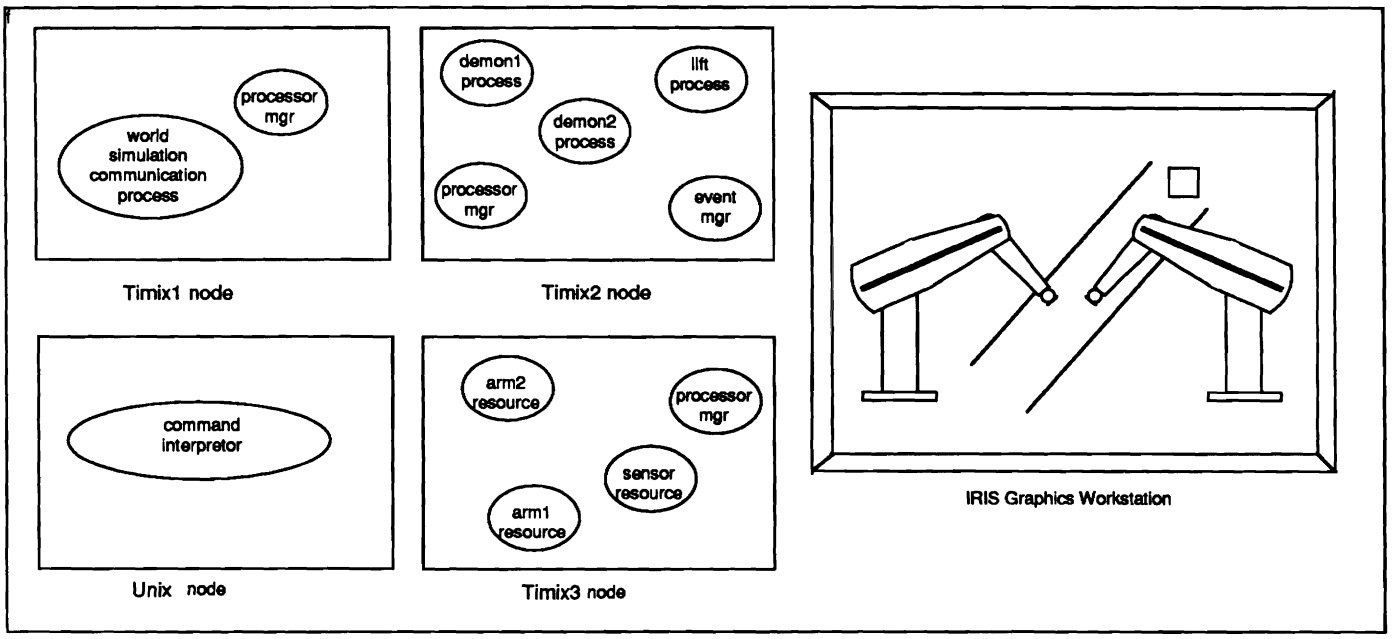

Figure 5: Two-Arm Simulation Software and System

demon processes were introduced to the system to compete for use of the arms.

Simulation Operating Environment. The operating environment (shown in Figure 5) consists of the three MicroVax II computers running the *TimixV2* real-time kernel, a MicroVax II computer running Ultrix, and a 16 MIPS Personal Iris 4D-25 computer with a hardware turbo graphics option, all connected via an Ethernet.

The Iris executes software based on a 3-D modeling package provided by the Computer Graphics Laboratory at the University of Pennsylvania [19]. The package was extended to simulate the control of Puma560 robot manipulators in a kinematic environment [20].

Application Software. The application's *RTC* processes and resources are indicated in in Figure 5. Process *Lift* is essentially the same as the process described in the example of Section 3. It awaits the detection of the moving object, which is signaled by the *Sensor* resource, then requests actions to move to, grasp, and lift the object. The other two processes are used to compete for the resources. Process *Demon1* attempts to move the arms (incompatible actions to the actions in process *Lift*) periodically with a period of 1.4 seconds. The deadline imposed by process *Demon1*'s period causes this process to have higher priority than process *Lift*, which has a 10 second deadline. However, allowing process *Demon1* to overlap execution with process *Lift* would violate process *Lift's* guaranteed and exclusive execution constraints and possibly its simultaneity constraint. The other process, *Demon2*,

reads the position of the arms and writes it to the screen with a period of 1.2 seconds. Process *Demon2's* execution can be allowed to overlap with *Lift* because it is executing compatible actions to those in *Lift*. The locking provided by our run-time support proved sufficient to assure process *Lift's* correct execution.

A more-complete discussion of the implementation, performance evaluation, and application of this implementation of the *RTC* language constructs and run-time system can be found in [21].

5 Related Work

Related work in abstract data type techniques and in transaction theory formed the basis for the programming paradigm that was presented in Section 2. However, since typical abstract data type and transaction-based models ignore timing constraints, we incorporated work in the area of explicit timing constraint expression and system enforcement into our model and language constructs. Also absent from typical transaction models are the requirements of inter-process synchronization and communication found in concurrent real-time systems; therefore we also examined work in these areas. In addition to theoretical work, the features of many current languages that have influenced our work.

Transaction-Theory. The *RTC* action and process constructs are based on Bernstein, Hadzilacos, and Goodman's transaction model [8], 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 [8]. Our notion of compatibility relations has its origins in the semantic compatibility of transactions [22, 9, 11]. Although compatibility relations are often too large and complex for general transaction systems, the limited number of actions within an *RTC* resource make their use possible for defining a resource's actions compatibility. Another difference between actions and transactions is that actions always commit unless instructed to abort externally from the calling process; they "abort" to a user-defined (consistent) state that was not necessarily the original state. Furthermore, actions can be time constrained and their priority is inherited from the caller.

While *RTC* processes borrow the notions of partial ordering, exclusive execution, and atomic execution from transactions, an *RTC* process is not a transaction. First, exclusivity and atomicity are decoupled and enforced on parts of an *RTC* process instead of all of it as is done with a transaction. This design allows the locking implementation of Section 4 to improve resource utilization and concurrency because locks do not have to be held for an

entire process. Second, the action invocations of an *RTC* process are transactions rather than simple operations. This establishes a paradigm similar to a nested transaction paradigm, which Moss [23] describes as an excellent way to enforce complex consistency constraints. Third, *RTC* processes have no explicit commit or abort actions. Furthermore, *RTC* processes are not independent; they synchronize through inter-process precedence orderings. Finally, *RTC* processes are time constrained and transactions typically are not.

Real-Time Databases. Recently the integration of timing and consistency constraints has been addressed in the domain of real-time databases. Most of these approaches incorporate time into traditional concurrency control mechanisms: timestamp methods [24, 25]; optimistic concurrency control methods [26, 27, 28]; multiversion methods [29], and locking methods[1, 2, 24, 30, 31]. All of these methods use serialization of transactions as the correctness criteria for concurrency control. For instance, in [2] and [29], the serialization order is dynamically adjusted to reflect the priorities of transactions. Others have examined relaxing data consistency requirements to better meet timing constraints [32, 33, 34]. Most of these approaches to concurrency control use blocking and/or abortion of transactions that can reduce predictability of real-time scheduling. Like many of these real-time database approaches, the *RTC* run-time system described in Section 4 uses priority-based locking. However, as described in the model of Section 2, the *RTC* system does not require strict serializability, but instead requires a semantic conflict-based concurrency control criteria similar to that proposed in [22]. This relaxed requirement allows the operating system flexibility in employing real-time scheduling of tasks. Also, the *RTC* run-time system does not abort tasks due to conflict.

To limit the adverse real-time effects of *priority inversion* caused by blocking, priority inheritance protocols have recently been proposed [35, 3, 31]. Although we do not currently address priority inversion, these ideas could easily be used in the resource manager tasks to limit priority inversion due to granting compatible locks to low-priority action invocations when there are higher-priority action invocations pending.

Real-Time Languages and Systems. 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. Ada and Modula-2 provide no explicit support for specifying absolute start times, execution durations, deadlines, periods, nested timing constraints and variable timing constraints. Also, as we discussed in Section 1, Ada and Modula-2 use mutual exclusion techniques with FCFS queuing to schedule shared resources. ARTC++ [36] employs an object-based paradigm with concurrency. Explicit timing constraints are pro-

vided in the temporal scope constructs of [37], Real-Time Euclid [7], Flex[6] and Maruti [38], among others. Temporal Scopes, Real-Time Euclid, and Flex also provide exception handling for constraint violations. The Spring kernel [39] provides guaranteed execution for entire processes rather than a set of actions. Esterel [40] provides event-based predecence ordering capability where timing constraints are treated as events. This is a dual notion to *RTC*, which also provides event-based precedence ordering, but treats events as timing constraints. Separate exclusive and atomic sets as well as concurrency based on action-level compatibility are not directly supported in other current real-time languages.

Maruti [38] 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.

Halang and Stoyenko provide a good evaluation of other real-time languages, such as Pearl [41], in [4].

6 Conclusion

This paper has described the *RTC* model, language constructs, their implementation and application in concurrent real-time programming.

The real-time concurrency model of Section 2 provides a framework to define and reason about real-time concurrency constraints found in distributed real-time applications. It synthesizes aspects of distributed transaction-based programming, an abstract data type paradigm, timing constraint enforcement, and precedence orderings into a paradigm and basic set of requirements for concurrent real-time programming.

The *RTC* language constructs naturally express real-time constraints, and can be embedded in a variety of programming languages. This explicit constraint expression allows system enforcement of constraints instead of burdening the programmer with their enforcement. Furthermore, our run time system can be easily implemented on any operating system that has dynamic priority-based scheduling and the ability to block interrupts.

Our current model has a limitation in that it disallows nested resources and nested action invocations. It may be useful to support nested resources, so that consistency requirements spanning multiple resources can be naturally defined by a resource which encloses the conflicting resources. In addition, nested actions might be useful to allow one resource to use

other resources directly, which has been shown useful in the untimed nested transaction model of [14] and [23]. However, it is not clear that this complexity is useful in a real-time environment and should be investigated.

Our current implementation is also relatively slow due to the fact that the underlying processors are slow. For example, we could not use the current implementation of the *RTC* constructs to perform actual low-level robot control with sampling periods on the order of tens of milli-seconds, although the refresh rate of the graphic simulator was sufficiently slow for our system to work. To further investigate the effectiveness of our approach, we intend to implement the *RTC* constructs and run-time system in other (hopefully faster) real-time operating environments such as those adhering to the POSIX 1003.4 standard [16].

Having examined several different distributed robotic applications, we feel that timing and consistency constraints are needed. However, current paradigms for programming such systems do not adequately support these constraints. We believe that the development of the *RTC* model, the *RTC* constructs and their implementation are an important contribution in providing such support.

Acknowledgements: The authors thank Lui Sha of the Software Engineering Institute for his helpful suggestions, Robert King of IBM's Watson Research Center for his support with the *TimixV2* system, and Janez Funda of IBM's Watson Research Center for support with the robotics application.

References

[1] R. Abbot and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," in *14th VLDB Conference*, Aug. 1988.

[2] Y. Lin and S. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," in *Real-Time Systems Symposium*, IEEE Computer Society, Dec. 1990.

[3] J. Huang and J. Stankovic, "On Using Priority Inheritance in Real-Time Databases," in *Real-Time Systems Symposium*, IEEE Computer Society, Dec. 1991.

[4] W. Halang and A. Stoyenko, "Comparitive Evaluation of High Level Real-Time Programming Languages," *Real-Time Systems Journal*, vol. 2, pp. 365–382, 1990.

[5] T. Baker and O. Pazy, "Real-Time Features for Ada 9x," in *Real-Time Systems Symposium*, IEEE Computer Society, Dec. 1991.

[6] 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.

[7] 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.

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

[9] W. Weihl, "Commutativity-Based Concurrency Control for Abstract Data Types," *IEEE Transactions on Computers*, vol. 37, pp. 1488–1505, Dec. 1988.

[10] W. Zhao, K. Ramamritham, and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 564–577, May 1987.

[11] N. A. Lynch, "Multilevel Concurrency – A New Correctness Criterion for Database Concurrency Control," *ACM Transactions on Database Systems*, vol. 8, pp. 484–502, December 1983.

[12] A. Gheith and K. Schwan, "CHAOSart: Support for Real-Time Atomic Transactions," in *Nineteenth International Symposium on Fault-Tolerant Computing*, pp. 462–469, June 1989.

[13] H. Tokuda, "Compensatable Atomic Objects in Object-oriented Operating Systems," in *Pacific Computer Communication Symposium*, Oct. 1985.

[14] B. Liskov, "Distributed Programming In Argus," *Communications of the ACM*, vol. 31, pp. 300–312, Mar. 1988.

[15] R. King, "Design, Implementation, and Evaluation of a Distributed Real-Time Kernel for Distributed Robotics," Tech. Rep. CIS-90-40, Department of Computer and Information Science, University of Pennsylvania, November 1990.

[16] W. Corwin, C. D. Locke, and K. Gordon, "Overview of the IEEE Posix P1003.4 Real-time Extension to POSIX," *IEEE Real-Time Systems Newsletter*, vol. 6, pp. 9–18, Winter 1990.

[17] E. Knapp, "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, vol. 19, pp. 304–328, Dec. 1987.

[18] V. Wolfe, S. Davidson, and I. Lee, "Deadlock Prevention in Distributed Real-Time Systems." Submitted to The Real-Time Systems Journal, November 1990.

[19] C. Phillips and N. Badler, "Jack: a Toolkit for Manipulating Articulated Figures," in *Proceedings of ACM/SIGGRAPG Symposium on User-Interface Software*, pp. 22–30, 1988.

[20] J. Funda, *Teleprogramming: Overcoming Communication Delays in Remote Manipulation.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1991.

[21] 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.

[22] H. Garcia-Molina, "Using Semantic Knowledge For Transaction Processing in a Distributed Database System," *ACM Transactions on Database Systems*, vol. 8, pp. 186–213, June 1983.

[23] E. Moss, *Nested Transactions, An Approach to Reliable Distributed Computing.* The MIT Press, 1985.

[24] W. S. Kim, T. M. Chan, and J. Srvastava, "Processor Scheduling and Concurrency Control in Real-Time Main Memory Databases," in *Symposium on Applied Computing*, IEEE Computer Society, April 1991.

[25] O. Ulusoy and G. Belford, "Real-Time Concurrency Control in Distributed Database Systems," *IEEE Technical Committee on Real-Time Systems Newsletter*, vol. 3, pp. 1–5, Feb. 1991.

[26] J. Harista, M. Carey, and M. Livny, "On being Optimistic about Real-Time Constraints," in *ACM PODS Symposium*, April 1990.

[27] J. Harista, M. Carey, and M. Livny, "Dynamic Optimistic Concurrency Control," in *Real-Time Systems Symposium*, IEEE Computer Society, Dec. 1990.

[28] J. Huang, J. Stankovic, and D. T. an K. Ramamaritham, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," in *17th VLDB*, April 1991.

[29] W. S. Kim and J. Srvastava, "Enhancing Real-Time DBMS Performance with Multiversion Data and Prioirty Based Disk Scheduling," in *Real-Time Systems Symposium*, IEEE Computer Society, Dec. 1991.

[30] J. Huang, J. Stankovic, and D. T. an K. Ramamaritham, "Experimental Evaluation of Real-Time Transaction Processing," in *Real-Time Systems Symposium*, IEEE Computer Society, Dec. 1989.

[31] L. Sha, R. Rajkumar, S. Son, and C. Chang, "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, vol. 40, pp. 793–800, july 1991.

[32] H. Korth, "Triggered Real-Time Databases," in *16th VLDB Conference*, August 1990.

[33] K. J. Lin, "Consistency Issues in Real-Tie Database Systems," in *22nd Hawaii Internaltion Conference on System Science*, Jan. 1989.

[34] S. Vrbsky and K. J. Lin, "Recovering Imprecise Transactions With Real-Time Constraints," in *Symposium on Reliable Distributed Systems*, Oct. 1988.

[35] R. Rajkumar, *Task Synchronization in Real-Time Systems.* PhD thesis, Carnegie Mellon University, 1989.

[36] 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.

[37] I. Lee and V. Gehlot, "Language Constructs for Distributed Real-Time Programming," in *Proc. IEEE Real-Time Systems Symposium*, IEEE Computer Society, Dec. 1985.

[38] 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.

[39] 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.

[40] G. Berry, S. Moisan, and J.-P. Rigault, "ESTEREL: Towards a Synchronous and Semantically Sound High Level Language for Real Time Applications," in *Real-Time Systems Symposium*, 1983.

[41] T. Martin, "Real-Time Programming Language PEARL - Concept and Characteristics," in *Proc. COMPSAC, Chicago*, pp. 301–306, 1978.