November 1989

Language Constructs for Distributed Real-Time Consistency

Victor Wolfe
University of Pennsylvania

Susan 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-89-78.

Language Constructs for Distributed Real-Time Consistency

Abstract

In this paper, we present a model and language constructs for a distributed real-time system with the goal of allowing the structured specification of functional and timing constraints, along with explicit, early error recovery from timing faults. To do this, we draw on ideas from (non-distributed) real-time programming and distributed transaction-based systems [81]. A complete language is not specified; the constructs described are assumed to be embedded in a block-structured procedural host programming language such as C [9] or C++ [10] (our current preliminary implementation is in C). The model consists of *resources, processes*, and a *global scheduler*. Resources are abstractions that export operations to processes, and specify acceptable concurrency of operations to the scheduler. Processes manipulate resources using the exported operations, and specify synchronization and restrictions on concurrency (at the exported operation level) to the scheduler. Examples of the types of information given to the scheduler are that a set of operations should be performed "simultaneously", or that a sequence of operations should be performed without interference by another process. The global scheduler embodies the entity or entities that schedule the CPU, memory, devices and other resources in the system. It performs preemptive scheduling of all resources based on dynamic priorities associated with the processes, preserves restrictions on concurrency stated by resources and processes, and is capable of giving "guarantees" to processes that they will receive resources during a specified future time interval.

The remainder of the paper is structured as follows. In the next section, we present language constructs for an expression of timing constraints called temporal scopes, and described resources and processes. Section 3 describes what is required of the global scheduler to support these constructs, and what is entailed in guaranteeing functional consistency.' We conclude in Section 4.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-78.

Language Constructs For Distributed Real-Time Consistency

MS-CIS-89-78
GRASP LAB 199

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

December 1989

Language Constructs for Distributed Real-Time Consistency

Victor Wolfe, Susan Davidson, and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

November 27, 1989

1 Introduction

Real-time applications such as robotics, industrial control and avionics, operate in environments that impose complex functional and timing constraints with severe consequences if violated. Frequently, they are distributed to match the topology of the application devices, to provide better performance through concurrency, and to improve system fault tolerance. For example, consider an application in which two robot arms must lift a container of chemicals from a moving conveyer belt [1]. The belt and each robot arm is controlled by separate process; these processes must synchronize to grab the container as it passes by the arms. To prevent spills, either both arms should lift or neither arm should lift, the arms should lift simultaneously, and no other use of the arms should be allowed while the lift is being performed. Furthermore, the lift should be performed under timing constraints to adhere to the dynamics of the moving belt and inherent properties of robot control algorithms.

Language support for these distributed real-time applications is crucial. Since the applications are extremely complex, programs must be easy to write, verify, and modify. The language must be capable of expressing the functional, synchronization and timing constraints inherent in the underlying application in a structured manner in the program. Programs should be *modular* so that resources and processes can be decomposed and constraints can be locally specified in modules with limited knowledge of the behavior of other modules. *Abstraction* should be supported so that interaction between modules is based on a simple, well-defined interface. The abstract characterization of modules and constraints should be a simple, natural reflection of information that the programmer has available. Currently, however, very little has been done to provide structured support for distributed real-time programming.

One approach that has been taken is to extend traditional cyclic executive scheduling to distributed real-time systems [2]. However, this fixed scheduling approach is inadequate for dynamic real-time applications since it becomes intractable for a fixed schedule to account for the number of possible scenarios posed by a changing environment. Even if it were possible to account for dynamic behavior, creating distributed synchronized executives is complicated and modification requires re-scheduling the system.

Another approach has been to write a concurrent program that is logically correct and then add run-time scheduling primitives to satisfy real-time constraints. For instance, in Ada [3] the programmer must translate timing constraints into static priorities so that a priority-based scheduler schedules tasks to meet the timing constraints. In

Modula-2 [4] the programmer must explicitly add transfer commands so that co-routines coordinate with each other to meet timing constraints. However, using scheduling primitives to "express" timing constraints results in programs that are difficult to write, verify, and modify. Hiding timing constraints in complicated synchronization patterns hinders error recovery from timing violations since it is difficult or impossible to determine which timing constraint was violated. Furthermore, since the scheduling primitives are added at compile-time, coping with a dynamic run-time environments is complicated. Another problem is that the scheduling primitives are only concerned with scheduling the CPU and not other shared resources that are needed to meet timing constraints, such as memory and devices. The result is that priority inversions may occur, and timing constraints violated.

Many of these problems can be avoided or mitigated by explicitly expressing timing constraints and exception handlers for timing violations, and allowing the run-time system to schedule system resources to best meet them. This is similar to the approach taken in programming languages such as Pearl [5], Real-Time Euclid [6], and Flex [7]. These languages, however, do not provide structured programming for *distributed* systems. In particular, they do not support the synchronization of two actions that must be either simultaneous, exclusive, or all-or-nothing. Furthermore, these languages are not concerned with expressing timing constraints on resources other than the CPU.

In this paper, we present a model and language constructs for a distributed real-time system with the goal of allowing the structured specification of functional and timing constraints, along with explicit, early error recovery from timing faults. To do this, we draw on ideas from (non-distributed) real-time programming and distributed transaction-based systems [8]. A complete language is not specified; the constructs described are assumed to be embedded in a block-structured procedural host programming language such as C [9] or C++ [10] (our current preliminary implementation is in C). The model consists of *resources*, *processes*, and a *global scheduler*. Resources are abstractions that export operations to processes, and specify acceptable concurrency of operations to the scheduler. Processes manipulate resources using the exported operations, and specify synchronization and restrictions on concurrency (at the exported operation level) to the scheduler. Examples of the types of information given to the scheduler are that a set of operations should be performed "simultaneously", or that a sequence of operations should be performed without interference by another process. The global scheduler embodies the entity or entities that schedule the CPU, memory, devices and other resources in the system. It performs preemptive scheduling of all resources based on dynamic priorities associated with the processes, preserves restrictions on concurrency stated by resources and processes, and is capable of giving "guarantees" to processes that they will receive resources during a specified future time interval.

The remainder of the paper is structured as follows. In the next section, we present language constructs for an expression of timing constraints called *temporal scopes*, and described resources and processes. Section 3 describes what is required of the global scheduler to support these constructs, and what is entailed in guaranteeing functional consistency.[1] We conclude in Section 4.

2 Language Constructs

What timing expression is needed in a distributed environment where there are multiple threads of execution? Firstly, constraints on single threads of execution must be expressed. These include bounds on *start times*, *deadlines*, and

[1] A complete explanation of these ideas is presented in [tech89?].

total execution times [11]. Secondly, constraints on the coordinated execution of multiple threads must be expressed. In traditional distributed environments where only functional consistency must be reasoned about, there are notions of "synchronizing" the order of execution of threads using synchronous communication or a rendezvous construct [3], there are also notions of *serializable* execution and a guarantee of *atomicity*, both of which are bound up in the notion of a transaction [8,12]. In a time-constrained environment, we sometimes need something stronger than synchrony or serializability: *simultaneity* so that threads may execute at the same time and *exclusive execution* so that threads can be assured of no interference during specific time intervals. For example, in the coordinating robots example, the two arms must lift the container simultaneously to avoid spilling. Exclusive execution is also required: each arm must execute a "grasp" and series "lifts" without interference from other processes attempting to use the arm. This lifting must also be done atomically, that is, either both arms should "grasp" and "lift", or neither should. Furthermore, the entire lifting task is time-constrained since there is a deadline, imposed by the dynamics of the belt, by which the lift must either be completed, or the coordinating process notified that it did not occur.

In this section, we give constructs for timing expression, starting with *temporal scopes* for single threads of execution. We then define *resources* as single threads of execution that export operations to *processes*. Processes can fork concurrent execution of exported operations, and therefore can require that a set or a sequence of operations be *exclusive* or *simultaneous*. Finally, there is a notion of *timed atomic commitment*, which allows a set of operations to be performed if and only if there is a guarantee that, barring faults, the operations can all be completed by a global deadline.

2.1 Temporal Scopes

In a real-time applications, operations frequently must start at a specific time or complete by a deadline. *Temporal scopes* [11] explicitly express these timing constraints as follows:

```
after ⟨sa⟩ before ⟨sb⟩ execute ⟨e⟩ within ⟨d⟩ do
    ⟨statements_1⟩
except
    when E_START do ⟨statements_2⟩ end when
    when E_EXECUTE do ⟨statements_3⟩ end when
    when E_DEADLINE do ⟨statements_4⟩ end when
end before
```

⟨statements_1⟩ must not start until after ⟨sa⟩. If ⟨statements_1⟩ are not started by ⟨sb⟩, then ⟨statements_2⟩ are executed. If ⟨statements_1⟩ or ⟨statements_2⟩ take over ⟨e⟩ execution time, then execution is terminated and ⟨statements_3⟩ are executed. If the the action is not completed by ⟨d⟩, then execution is terminated and ⟨statements_4⟩ are executed. Each constraint and exception handler clause is optional. A discussion of *nested* temporal scopes and a summary of the temporal scope and periodic temporal scope language constructs is given in [tech89?].

2.2 Resources

Resources are designed as abstract data types where each resource has a local state and a set of exported operations. A resource is declared as follows:

```
resource ⟨resourceID⟩
```

```
        ⟨data structure declarations⟩
        ⟨local procedure declarations⟩
        ⟨action declarations⟩
        ⟨statements⟩
    end resource
```

The resource's state parameters and local procedures are declared using data structure and procedure declarations of the host language. For example, state parameters of the robot arm resource might include declaration of data structures for its Cartesian coordinates and the position of its hand (grasp/ungrasp). The resource's body consists of ⟨statements⟩ that are executed when the resource is created. The resource's exported operations, called *actions*, are declared as follows:

```
    action ⟨actionID⟩ ⟨arguments⟩;
        compatible { ⟨actionID_1⟩, ..., ⟨actionID_n⟩ }
        ⟨statements⟩
    end action
```

⟨arguments⟩ are value parameters. The *compatible* declaration is used to inform the scheduler which actions may preempt this action without causing functional inconsistency (see Section 3). The action's body is a sequence of ⟨statements⟩ designed to meet functional constraints when executed in isolation to completion. Its original deadline is *inherited* from the process that calls it. Temporal scopes within the action expressing timing constraints that are *inherent* to the resource are *nested* within the inherited temporal scope. Consider a robot arm resource as an example; it could have the following actions: *lift*, *lower* and *grasp*; as well as others. The coordinating process may impose a 10 second deadline on a *lift* action of a robot arm when the action is called. The *lift* action inherits this deadline, and any constraints inherent to the action due to the lift control algorithm are nested within this 10 second deadline.

2.3 Processes

Process are declared as follows:

```
    process ⟨processID⟩
        ⟨local data structure declarations⟩
        ⟨local procedure declarations⟩
        ⟨timed action declarations⟩
        ⟨process body⟩
    end process
```

Local data structures and procedures are declared in the host language and may be used only within the process. *Timed actions* are a special kind of procedure for use in *timed atomic commitment*, and are described later. A *process body* consists of a sequence of either host language statements, *action calls*, *concurrency blocks*, *reservation blocks*, or *tac blocks*. The temporal scope construct described in Section 2.1 is used to express timing constraints on sequences of blocks in processes. We now describe the components of a process body.

Action calls are specified as:

⟨resourceID.actionID⟩ (⟨params⟩).

Concurrency blocks are expressed using **cobegin – coend** constructs. Only action calls are allowed in this block. The actions are called concurrently and all actions must complete in order for the concurrency block to complete.

Reservation Blocks. Reservation blocks allow a process to detect potential timing constraint violations before they occur by determining whether resources will be available to meet timing constraints. A single reserve statement requests a guarantee from the scheduler that an action will receive e units of execution time somewhere in a time interval $[s, d]$; e is not necessarily consecutive. It can be used as a statement in a process:

$[sr, er]$ = **reserve** e, $[s, d]$ ⟨resourceID.actionID⟩

The return value indicates the lower and upper bounds of the time that the action is reserved for, or *NULL* indicating that the guarantee could not be made. These bounds may be used to synchronize subsequent reservations requests; for example er from one reservation can be used as s in a later reservation to ensure that second reservation starts after the first reserved interval completes.

Another form of reservation block is a *simultaneous reservation block*, which is used to specify that actions start at the same time and that each action's reserved time is uninterrupted after this common start time. The syntax is

```
begin align
    $[sr_1, er_1]$ = reserve $e_1$, $[s_1, d_1]$ ⟨resourceID.actionID$_1$⟩
    ⋮
    $[sr_n, er_n]$ = reserve $e_n$, $[s_n, d_n]$ ⟨resourceID.actionID$_n$⟩
end align
```

Section 3 describes how the scheduler ensures that $sr_1 = sr_2 = \ldots sr_n$ and that each reservation is uninterrupted. This construct can be used in the example to specify that the arms are to lift simultaneously.

Another use of reservations is to exclude other actions from executing at certain times because, in order to reason about consistency, some sets of actions in a process must appear to be executed without interference from other processes. For instance, while the lift coordinating process is performing a series of incremental *lift* actions on an arm, we do not wish to allow a *lower* action issued by another process to be scheduled. Exclusive actions may not necessarily be on the same resource. For instance, the process may wish to ensure that the passing of an object from one arm to another, which is a sequence of interleaved actions of the two arms, is not interfered with by other processes trying to use either arm. To express these requirements, another form of reservation block called an *exclusive block* is used in a process. An exclusive block has the structure:

```
begin exclusive
    $[sr_1, er_1]$ = reserve $e_1$, $[s_1, d_1]$ ⟨resourceID.actionID$_1$⟩
    ⋮
    $[sr_n, er_n]$ = reserve $e_n$, $[s_n, d_n]$ ⟨resourceID.actionID$_n$⟩
end exclusive
```

The exclusive block specifies that during the interval bounded by the earliest start of a reservation granted in the block to the end of the latest reservation granted in the block, called the *exclusive interval*, no action that could interfere with any action in the block can be executed. Note that this is a stronger requirement than the serializability requirement found in non-real-time distributed systems [8,13,12]. Serializability requires that, although actions which could potentially interfere with a transaction can be scheduled during the transaction, the schedule must be functionally equivalent to one in which no interference occurs. Exclusive blocks further require that no incompatible action appear *anywhere* in the exclusive interval. This stronger requirement is needed because traditional serializability is not concerned with *when* interfering actions occur, only that they do not cause the schedule to be unserializable. However, in a distributed real-time control application where resources interact, the time at which interfering actions occur is important. For instance, consider the passing of an object from one arm to another: Serializing the schedule of each arm is not sufficient because it may allow an incompatible movement of the arm to be executed on arm1 while the passing of the object is being performed. If the incompatible move action is scheduled after all movement of arm1 required by the passing process is done, the passing actions may be serializable, but this incompatible action could cause arm1 to be moved before arm2 is done interacting with it. An exclusive block alleviates this problem by specifying that the interfering movement can not be scheduled while any actions in the block are still executing. An exclusive block also specifies that either all reservations are granted with the above restriction or all of them are denied (indicated by *NULL* return values for all intervals in the exclusive block). Section 3 describes how a scheduler can ensure these requirements.

Simultaneous and exclusive reservation blocks may be nested within each other. A simultaneous block nested within an exclusive block specifies that the actions in the simultaneous block start at the same time, and that their reservations, like all other reservations in the outer exclusive block, are used to compute the exclusive interval. An exclusive block nested within a simultaneous block specifies that the all actions in the simultaneous block, including those within the nested exclusive block, have reservations starting at the same time; those actions in the nested exclusive block further require that no actions that interfere with them be executed until all of their reservations have completed. Nesting exclusive blocks in exclusive blocks or simultaneous blocks in simultaneous blocks is redundant; it adds no further restrictions on the scheduler.

Timed Atomic Commitment. Another constraint posed by processes is that some sets of actions have the property that either all actions in the set be performed correctly or none should be performed. Reaching a decision to perform under an all-or-none constraint is known as *atomic commitment* [13]. When the decision and the performance of the decided-upon action must be done under timing constraints, it is called *timed atomic commitment* (TAC) [1]. In the example, the lifting of the two arms is a TAC because if one arm can not grasp the container or it appears that one arm can not complete lifting within timing constraints, then neither arm should lift to prevent spills of the container. A more formal definition of the TAC problem and its constructs is given in [1].

An action in a TAC is a special form of process procedure called a *timed action* that coordinates with other timed actions by executing an underlying protocol, such as a *centralized timed two-phase commit protocol* [1] to ensure all-or-nothing behavior. Timed actions have three stages to detect timing constraint violations as early as possible: a *reservation stage* which detects potential violations before a timed action starts; a *vote stage* which detects violations before a timed action performs any actions that affect consistency; and a *performance stage* during which consistency-altering actions are performed. The structure of the timed action construct is:

```
timed action ⟨action-name⟩ ( ⟨parameters⟩ )
    { ⟨reservation_block⟩ }
begin
    ⟨statements_1⟩ /* decide vote: YES or NO */
    vote (YES/NO)
    await
        when COMMIT do ⟨statements_2⟩ end when
        when ABORT do ⟨statements$_3$⟩ end when
    except
        when E_DEADLINE do ⟨statements$_4$⟩ end when
end action
```

The timed action's reservation stage is indicated by { ⟨reservation_block⟩ } which lists a set of reservation blocks (described above). If any of the specified reservations are denied, then ⟨statements$_3$⟩ are executed. ⟨statements$_1$⟩ constitute the vote stage, which ends with the **vote** statement. If the decision arrived at by the underlying protocol is to commit in the performance stage, then ⟨statements$_2$⟩ are executed; if the decision is not to perform (abort), ⟨statements$_3$⟩ are executed. If the timed action's inherited deadline is violated, then current execution is terminated and ⟨statements$_4$⟩ are executed.

In our example, a timed action for lifting arm1 would first attempt to reserve use of arm1 in time to meet its deadline. In its vote stage, it would call the grasp action of arm1 to attempt to grasp the container before lifting. If the initial reservation is denied or it can not grasp, then it will vote NO and not lift. Otherwise, it executes the underlying TAC protocol to determine if it should lift.

To invoke a TAC, a process starts a set of concurrent timed actions and then waits for their results. The structure of the TAC block is:

```
begin tac
    $V_1$:= ⟨Timed_actionID$_1$⟩ (⟨args⟩)
    ⋮
    $V_n$:= ⟨Timed_actionID$_N$⟩ (⟨args⟩)
end tac;
```

The return values, V_i, in the TAC block indicate whether the timed action committed, aborted, or has not returned. When all timed actions have returned, the TAC completes. To establish a deadline for TAC, the TAC block is enclosed within a temporal scope.

3 Scheduling

We include the scheduler in our language construct presentation because the scheduler must enforce the consistency that the program expresses. Proposing a new scheduling algorithm or advocating a particular existing scheduling algorithm is beyond the scope of this paper; instead, we discuss the requirements posed to a scheduling algorithm by our language constructs and outline possible implementation techniques.

Meeting Timing Constraints. To meet timing constraints, a scheduler should use the timing constraints expressed by temporal scopes to schedule the actions. This can be done by incorporating timing constraint information into a dynamic priority associated with a process and scheduling based on this priority. For example, earliest-deadline-first scheduling determines dynamic priority as a function of the deadline alone and has been shown to be optimal for meeting deadlines under certain conditions [14]. A more complex example is the Spring kernel [15], which incorporates slack time, static priority, and resource use into it dynamic priority value.

To further support timing constraint enforcement, the scheduler should avoid *priority blocking* that occurs when a lower-priority process keeps a higher-priority process from executing by holding resources required by the higher-priority process. Avoiding priority blocking allows more important processes to meet their timing constraints, and if the dynamic priority value is dependent on timing urgency, then timing consistency is better supported. Furthermore, the scheduler should minimize the time that it allows processes to hold idle resources because this holding may block other processes and contribute to them missing their timing constraints unnecessarily. One way to do avoiding priority blocking and reduce the holding of idle resources is to allow the scheduler to preempt one action for another wherever possible. Preemptive scheduling has been shown to be optimal for meeting timing constraints [14].

Meeting Functional Constraints. The problem with allowing arbitrary preemption is that the functional constraint-preserving property of actions may be lost because the property is predicated on the action's executing without interference from other actions. The scheduler should preempt only if the resulting resource schedule is functionally consistent. A *functionally consistent schedule* is one in which all actions that have completed in the schedule produce consistent states. One way to achieve a functionally consistent schedule is to require that the schedule be *serializable*. A serializable schedule yields the same result as a schedule where all actions are run to completion without concurrency [13]. All serializable schedules are, by the definition of actions, functionally consistent. However, there may be preemptions that produce a functionally consistent schedule that is not serializable.

To determine the functional consistency of a schedule, we use a similar method to that used to determine serializability [12]: a *dependency graph.* Let S_r be a schedule for a resource r and let $DG(S_r)$ be a directed graph. The nodes of $DG(S_r)$ are the actions in S_r. Let $start(a_i)$ be the absolute time that action a_i starts in S_r and $complete(a_i)$ be the absolute time that it completes. There is an edge in $DG(S_r)$ from the node representing a_k to the node representing a_l, iff a_k is not compatible with a_l and $start(a_k) < complete(a_l)$ in S_r. Capturing compatibility semantic information is done using a *compatibility set* [12] associated with each action expressed by the *compatible* declaration found in actions. For instance, assuming the *grasp* action does not affect the z-coordinate of the robot arm and the *lift* and *lower* actions affect only the z-coordinate, *grasp* should be declared as compatible with *lift* and *lower*. In [tech89?] we show that a schedule is functionally consistent if and only if its dependency graph is acyclic by using a proof similar to that for showing serializability from a dependency graph. To ensure a functionally consistent schedule, the scheduler must allow only compatible actions to preempt each other so that cyclic edges in the dependency graph are prevented. Thus, *grasp* may have its operations interleaved with operations from *lift* and *lower* in the robot arm's schedule; however, *lift* and *lower* are not compatible, thus they may not be interleaved.

Reservations. The scheduler must also be capable of handling requests for reservations. To do this, it keeps a *reservation table* that records the interval that actions have been granted reservations. Each time unit in the table is marked as either: *available*, indicating that any action may execute or be given a reservation; *restricted*, indicating that only a certain specified class of compatible actions may execute or be given a reservation; or *reserved*, indicating that no action may execute or be given a reservation. In order to guarantee receiving e execution time the requesting action must be given e *reserved* units of time. The reserved interval does not necessarily have to be consecutive, it may be split into several sub-intervals. However, to maintain functionally consistent schedules, actions possessing reservations in the intervening intervals must be compatible with the requesting action.

To support *simultaneous reservation blocks*, the scheduler must find a common start time for the reserved intervals of the actions in a simultaneous block and must additionly ensure that each of these reserved intervals is not broken into subintervals. To agree on a start time, a centralized scheduler (one scheduler for all resources in the system) could check reservation tables for all resources in the request; a decentralized scheduler (*e.g.* a scheduler for each resource), could employ a distributed agreement protocol [13].

To support *exclusive reservation blocks*, the scheduler must not only mark *e* units of *reserved* time for each reservation request made in the exclusive block, it must also mark all times in the exclusive interval as *restricted* to those actions which are compatible with all actions in the exclusive block. A method for doing this is to first attempt to make independent "conditional reservations" for each request. A conditional reservation is one that, if granted, may be released later. If all conditional reservations are granted, then the reserved interval is determined and the scheduler attempts to restrict it. If incompatible reservations have already been made in the exclusive interval, then all conditional reservations are released and the exclusive block is denied. Otherwise, the exclusive interval is marked as restricted and the exclusive block is granted. We are investigating other methods for enforcing exclusive blocks that reduce the amount of restricted time and improve the probability of granting the requests, such as a *two-phase reservation* protocol, which is similar to a two-phase locking protocol [13].

4 Conclusion

This paper has described language constructs for distributed real-time programming that are designed to support timing consistency and functional consistency, including simultaneity, exclusiveness, and atomicity of actions. The constructs also support early recovery, and structured programming techniques.

Functional consistency is supported through actions, which are designed to preserve functional constraints when executed without interference, and by the scheduler which ensures that actions are not interfered with by incompatible actions. Further functional consistency constraints on sets of actions are expressed by the *exclusive block* construct that specifies sets of actions that may not be interfered with, by the simultaneous block that specifies concurrent actions start at the same time and are uninterrupted, and by the *TAC block* that specifies all-or-nothing performance of timed actions. Resources and processes provide information to allow the scheduler to meet these constraints.

Timing consistency is supported by temporal scopes which explicitly express timing constraints for the scheduler to enforce. Allowing preemption also supports timing consistency by reducing priority blocking and holding of idle resources. To inform the scheduler of preemption that maintains functional consistency, we introduced the specification of compatibility sets for actions. The reduction of priority blocking and of holding idle resources is also supported by the use of the *reserve* construct, which allows the functional consistency preservation provided by traditional locking techniques while reducing the idle holding of resources found in traditional locking.

Early detection of constraint violations is supported by the use of reservations that determine whether certain sets of actions will meet their timing constraints before the actions are executed. The exception handler associated with temporal scopes provides recovery in the event that constraints are actually violated. In addition, each of the three stages of a timed action provide an opportunity to check if constraints will be met and to invoke recovery if they will not be met.

The use of an abstract data type paradigm for resource design and a transaction paradigm for action and process design supports modularity and abstraction in our approach. Expression of a resource's functional and timing constraints is isolated within its actions. The actions then provide an abstract interface to processes that hides these constraints. Temporal scopes, exclusive blocks, and TAC blocks modularize and explicitly express constraints making the writing, modification, and verification of the distributed real-time programs easier. In addition, the constructs support the use of a dynamic real-time scheduler, such as that proposed in [15], that abstracts the scheduling of constraints from the processes and actions themselves.

The language constructs are currently being embedded in the C language [9] executing using a real-time kernel [16] being developed at the University of Pennsylvania for distributed real-time control applications.

References

[1] S. Davidson, I. Lee, and V. Wolfe, "Timed atomic commitment," Tech. Rep. MS-CIS-88-80, Department of Computer and Information Science, University of Pennsylvania, Oct. 1988. Submitted for publication.

[2] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Sentf, and R. Zainlinger, "The Mars approach," *IEEE Micro*, vol. 9, pp. 25–40, Feb. 1989.

[3] U.S. Department of Defense, "Ada Programming Language," 1983. ANSI/MIL-STD-1815A-1983.

[4] N. Wirth, *Programming in Modula-2*. New York: Springer-Verlag, 1983.

[5] T. Martin, "Real-time programming language Pearl - concept and characteristics," in *Proc. COMPSAC, Chicago*, pp. 301–306, 1978.

[6] 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, Sep. 1986.

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

[8] B. Liskov, "Distributed programming in Argus," *Communications of the ACM*, vol. 31, pp. 300–312, March 1988.

[9] B. Kernighan and D. Ritchie, *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

[10] B. Stroustrup, *The C++ Programming Language*. Reading, Ma.: Addison-Wesley, 1986.

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

[12] 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.

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

[14] C. Liu and J. Layland, "Scheduling algorithms for multi-programming in a hard-real-time environment," *Journal of the ACM*, pp. 46 - 61, Jan. 1973.

[15] K. Ramamritham and J. Stankovic, "Overview of the spring kernel," Tech. Rep. COINS 87-54, Department of Computer Science, The University of Masseschesetts, 1987.

[16] I. Lee, R. B. King, and R. P. Paul, "A predictable real-time kernel for distributed multi-sensor systems," *IEEE Computer*, vol. 22, pp. 78–83, June 1989.