October 1987

Motivating Time as a First Class Entity

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

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

Victor Fay-Wolfe
University of Pennsylvania, wolfe@cs.uri.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-87-54.

Motivating Time as a First Class Entity

Abstract

In hard real-time applications, programs must not only be functionally correct but must also meet timing constraints. Unfortunately, little work has been done to allow a high-level incorporation of timing constraints into distributed real-time programs. Instead the programmer is required to ensure system timing through a complicated synchronization process or through low-level programming, making it difficult to create and modify programs. In this report, we describe six features that must be integrated into a high level language and underlying support system in order to promote time to a first class position in distributed real-time programming systems: expressibility of time, real-time communication, enforcement of timing constraints, fault tolerance to violations of constraints, ensuring distributed system state consistency in the time domain, and static timing verification. For each feature we describe what is required, what related work had been performed, and why this work does not adequately provide sufficient capabilities for distributed real-time programming. We then briefly outline an integrated approach to provide these six features using a high-level distributed programming language and system tools such as compilers, operating systems, and timing analyzers to enforce and verify timing constraints.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-87-54.

MOTIVATING TIME AS A FIRST CLASS ENTITY

Insup Lee
Susan Davidson
Victor Wolfe

MS-CIS-87-54

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

(revised October 1987)

Acknowledgements: This research was supported in part by NSF grants DCR 8501482, DMC 8512838, MCS 8219196-CER, U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027 and a grant from AT&T's Telecommunications Program at the University of Pennsylvania.

Motivating Time As A First Class Entity

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

October 30, 1987

Abstract

In hard real-time applications, programs must not only be functionally correct but must also meet timing constraints. Unfortunately, little work has been done to allow a high-level incorporation of timing constraints into distributed real-time programs. Instead the programmer is required to ensure system timing through a complicated synchronization process or through low-level programming, making it difficult to create and modify programs. In this report, we describe six features that must be integrated into a high level language and underlying support system in order to promote time to a first class position in distributed real-time programming systems: expressibility of time, real-time communication, enforcement of timing constraints, fault tolerance to violations of constraints, ensuring distributed system state consistency in the time domain, and static timing verification. For each feature we describe what is required, what related work has been performed, and why this work does not adequately provide sufficient capabilities for distributed real-time programming. We then briefly outline an integrated approach to provide these six features using a high-level distributed programming language and system tools such as compilers, operating systems, and timing analyzers to enforce and verify timing constraints.

Contents

1 Introduction

In many applications such as robotics or industrial control, programs must not only be functionally correct, but must also meet timing constraints. These applications are referred to as *hard real-time* applications. Many real-time control systems are implemented as *distributed* computing systems to match the distributed topology of the devices controlling the application, provide better performance through concurrency and improve system fault tolerance. Distributed *real-time* programming, however, has an additional complication over distributed *non-real-time* programming in that timing constraints must be adhered to for correct performance.

The problem we address in this paper is that the incorporation of timing constraints is not adequately supported in distributed real-time programming. Although work has been done to provide high-level constructs that make it easier to create functionally correct distributed programs (e.g. strong typing, abstract data types, exception handling, atomic actions, recovery blocks), little work has been done to allow a high-level incorporation of timing constraints into distributed real-time programs. Current real-time languages [GL83,BMR83,US 83,KS86,Wir83] lack in their ability to express timing constraints since not all possible timing constraints can be expressed explicitly or naturally as they are posed in the application and no high-level constructs are provided to incorporate time into traditional distributed programming practices. Consequently, most of the proposed distributed real-time system support tools [CMMS79,DMV82,FR86,GLR82,RS82,SBWT85,TH86] still require the programmer to ensure that timing is met by low-level systems programming through constructs such as alarm events, watchdog processes, signals, and interrupts. Recent proposals for an integrated system of high-level languages and run-time-system support tools, such as [BMM87,Yan86], have made progress in easing the burden on the real-time programmer, but further work on issues such as variable and nested timing constraints, system tolerance to missed constraints, and incorporation of time into distributed state consistency, is required.

In this report we outline an integrated approach to elevating time to a first class entity in real-time programming by expressing timing constraints explicitly in a high-level distributed programming language and using system tools such as compilers, operating systems, and timing analyzers to enforce and verify timing constraints. The next section presents a model of the real-time control system we assume, and describes, as an example application, a real-time robot manipulator control application developed at the University of Pennsylvania. Section 3 describes six features that must be integrated to provide for a high-level treatment of time in distributed programming. Along with each feature, related work in the area is discussed. Section 4 briefly overviews our approach to providing an integration of these six features.

2 Assumptions and Model

This paper identifies necessary language constructs and tools for developing *hard real-time* applications on a *distributed* computer system controlling physical devices such as a robot manipulator and associated sensors [SBWT85,PZ85,SE85]. Control applications are characterized by input which comes from the environment and output which affects the environment. In hard real-time control systems, input, output and computation are time constrained (various types of time constraints are discussed in Section 3.1.1) and these timing constraints must be met for correct performance.

2.1 Assumptions About the Application

This section describes a real-time robotics application that illustrates concepts in this paper. A detailed description of the current implementation is presented in [PZ85,PZ84]. Assumptions about the distributed computation system including its physical structure, software structure, and faults that can occur, are also presented.

2.1.1 An Example Application

As an illustration of a hard-real-time control system we describe a robotic control application developed at the University of Pennsylvania: a multi-sensor, six joint Puma robot [PZ85,PZ84,PDM86]. The system control is performed by the processes shown in Figure 1. The coordinator process executes a user program that includes two kinds of operations: operations that define a world model and operations that define a task plan. The world model is described by positions and transformations for the robot. Input determining the world model comes from integrated sensor data. The task plan is a set of high-level instructions commanding the robot. Commands and a description of the current world model are passed from the coordinator process to the supervisor process through messages. The supervisor process interprets each command and plans a progression of Cartesian coordinates that the robot end effecter (hand) must reach to carry out the command. These Cartesian coordinates, called a *set point*, are sent to the joint processes which compute corresponding joint coordinates and control the actuation. The *sampling period* is defined as the period for the generation of new set points. During each sampling period, a servo process for each joint process accepts the current joint position and the desired joint position and performs an interpolation to determine the actuation required for the joint. The servo processes execute at a period four times faster than the sampling period to allow more frequent actuation signals to be sent to the joints than the sampling period allows.

2.1.2 Assumed Computation System

The assumed distributed computation system is a collection of nodes that communicate with each other through a network. A node is a collection of one or more processors that share a common memory. The processor(s) of a node are multiprogrammed and work under the control of a resident operating system such as the TIMIX real-time kernel currently being developed at the University

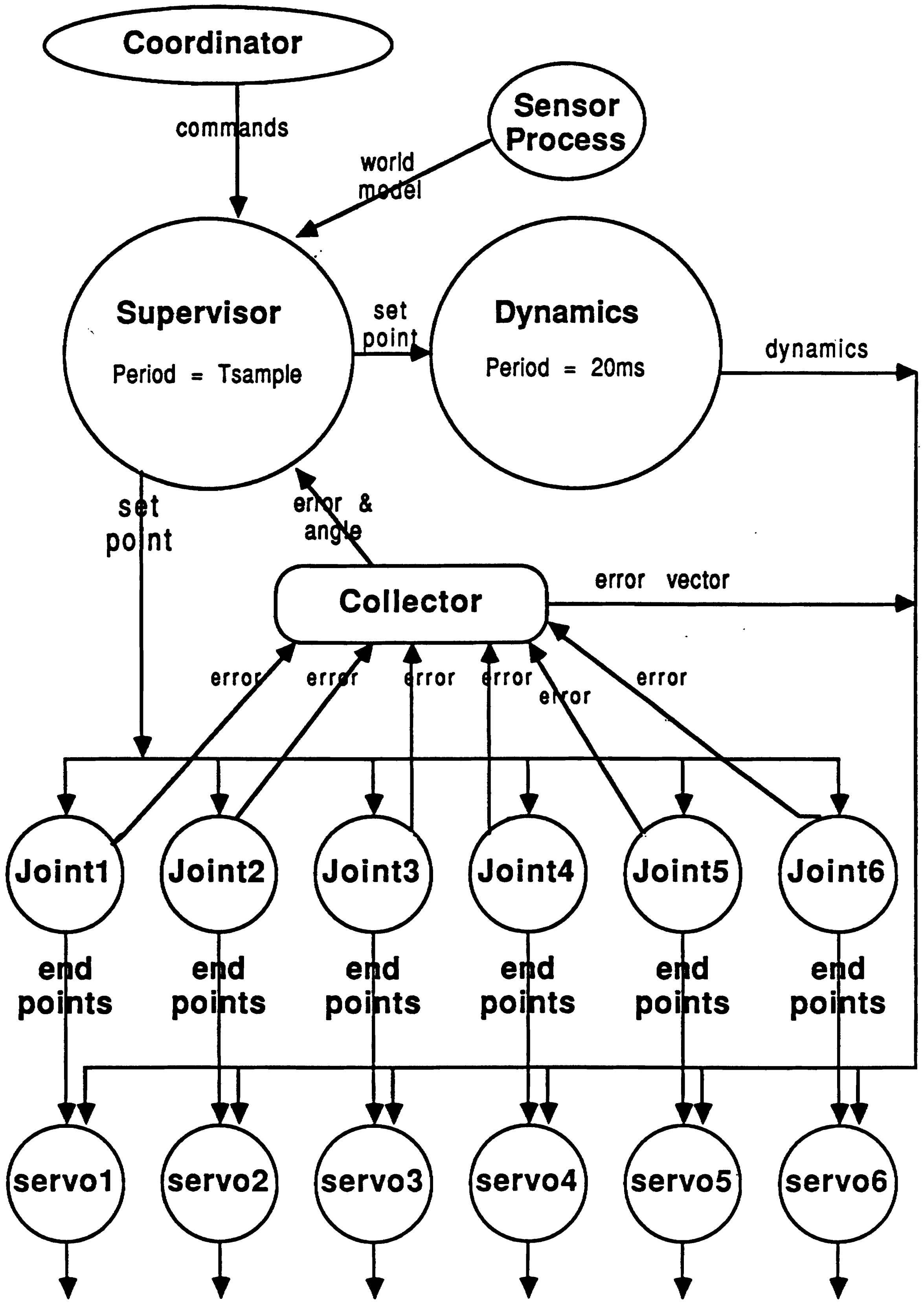

FIGURE 1

ROBOT MANIPULATOR CONTROL PROCESSES

of Pennsylvania [Kin87]. Each node possesses a local clock that is synchronized within a small fixed time interval of all other node clocks to provide a consistent notion of time throughout the system. Nodes communicate by means of messages through a network; no physical memory is shared between nodes[1].

It is assumed the application program is composed of concurrent processes that are autonomous, asynchronous computation tasks that only occasionally synchronize and communicate with each other through messages. The processes are written in a high-level applications-oriented language independent of the system topology, configured offline, and started at system boot time; there are no dynamically created processes. It is assumed process placement on nodes and communication port addresses are specified in a configuration specification language such as DICON [Lee84]. Nothing is assumed about how and why processes are placed at a node, only that they remain at a node and do not migrate. Once the system is running, the operating system resident on each node is responsible for scheduling resources such as the node processors and communication medium.

It is assumed *faults* may occur in the distributed system due to hardware failures and *errors* due to software design flaws. Only detectable faults and errors are assumed to occur and they do not completely prohibit system operation, that is, nodes fail cleanly. It is further assumed communication failures do not partition the network, implying there is always a communication path from one node to another.

3 Required Features For High-Level Real-Time Programming

To elevate time to a first class position in real-time programming, we feel that (minimally) the following six features must be integrated into a high-level language and underlying support system:

1. expressibility of time;

[1]Building message passing is possible with a shared memory implementation *within* a node. This implementation is transparent to the processes, which appear only to communicate through messages.

2. timed communication;
3. system enforcement of timing constraints;
4. tolerance to violations of constraints;
5. ensuring distributed system state consistency in the time domain; and
6. static timing verification.

For each of these six features we describe what is required, what related work has been done in the area, and what further work is needed to incorporate time to distributed real-time programming. Other typical desirable characteristics such as program modularity, system expandability, portability, etc., are addressed elsewhere by Gligor and Luckenbaugh [GL83] and not addressed here.

3.1 Expression of Timing

3.1.1 Timing Constraints to be Expressed

The first required feature is to make the expression of timing constraints in the high-level distributed programming language an explicit, natural reflection of the application specifications. In real-time processes, timing constraints are found on execution, input, output, and communication. Each constraint has an interval of validity denoted by a start time interval and a deadline interval for completion. In addition, processes may have a period when they are to be executed.

A *start time interval* of timing constraints is denoted by a constraint stating when the process is *allowed* to start and one stating when it *must* be started by. Similarly, a *deadline interval* is denoted by two times: when the process is *allowed* to be terminated, and when it *must* be terminated. Each of these kinds of timing constraints may be expressed either absolutely in specific wall-clock time or relatively with respect to the current time the process is executing. Not all tasks in a real-time application have to express all of these timing constraints. For example, the supervisor processes in the robot controller specifies only a period; the start time is inferred by the end of the previous period and the deadline by the end of the present period.

System input is environmental and thus constrained by the environment. Since the environment is constantly changing, input may only be valid for a certain interval of time and must be accepted in that time to be correct. In the robot control example, the sensor data is only valid for an interval depending on the rate at which an object, which is being tracked by the robot, is moving. System output is also time constrained. The joint actuators in the Puma manipulator, for instance, require commands every 1 ms. Interprocess communication may also be time constrained as discussed in Section 3.2.2.

Expected Execution Constraints Another form of timing constraint is the maximum expected execution time of a process. The maximum expected execution time of a process is different than its deadline. A deadline is an execution constraint set by the application specification stating when the process must finish to be correct. Waiting until the deadline to detect incorrect performance can be inefficient because the process may have gone into an error state well before the deadline is reached. If a process's maximum execution time (without errors) can be bound, then this bound can be used to detect erroneous performance often well before the process' actual deadline. For instance, if the robot dynamics process has a period of 20ms and a worst case expected execution time of 5ms, then there is a 15ms interval between when the process should finish and its actual deadline for finishing. Waiting to detect the violation at the 20ms deadline, when it is known the process is in error after 5ms, wastes time that is often valuable in real-time systems. The expected execution time should not necessarily be programmer specified, techniques are available for compiler-generated expected execution times (see related work section)

Variable Time Constraints. Timing constraints may be variable with their value depending on the state of the environment. Paul and Zhang [PZ85] describe a class of processes called *kinematic* processes which have variable periods that change with the environment. The sensor processes in our example all have variable periods; the actual period value depends on the speed of a moving object being tracked by the robot. A faster object requires a faster period for sensor updates.

Nested Time Constraints Timing constraints may be nested. A task in a real-time application may be time constrained and composed of nested subtasks that are themselves (further) time constrained. When a subtask is executing, the overall task is under by the subtask's constraints as well as its own constraints. An example of nested timing constraints are time constrained actions such as communication that may appear within processes that are also time constrained. When the process is executing the time constrained communication it will have tighter timing constraints than during the other segments of its execution.

Nested timing constraints may take the form of timing constraints on concurrent nested actions or serial nested actions. Concurrent parallel activities are possible in a distributed system so it may be necessary to express timing constraints on a collection of concurrent actions. For instance in the robot controller all six servo processes that actuate the joints must do so in the same 1ms period. These six processes may be distributed among six processors and executed concurrently with timing constraints expressed on the execution of the group of all six concurrent processes.

An example of serial nested constraints is the action plan of a robot on an assembly line. It may be necessary to constrain a series of actions to happen within a specified time in order for the robot to conform to the dynamics of the assembly line. Each of the actions in the series may be individually time constrained by the dynamics of the robot. This series of robot-imposed time constraints are serially nested within the assembly-line-imposed time constraints.

It is desirable that the incorporation of all timing constraints into the control program be done with high-level constructs. Incorporating program timing using low level constructs such as alarm events or external "watchdog" processes requires a knowledge of system implementation details that ideally the programmer should not be concerned with. Instead, the programmer should have the ability to simply express the constraints that are part of the application specification and allow the operating system to coordinate execution to meet those constraints. This high-level expression of timing reduces the complexity of program development and makes modification of timing easier.

3.1.2 Related Work on Expression of Timing Constraints

We now examine work related to the expression of timing constraints in real-time programs.

Alarms and Low-Level Programming. The most widely used technique of incorporating timing constraints into real-time programming is by alarm clocks or delays. The software for the robot controller, described by Paul and Zhang in [PZ85], is coded in C with low-level system calls implementing alarms to specify timing constraints. The alarm method uses a "watchdog process" that treats alarms as events, watches for the time of events, and signals processes when their time events occur. Each process sets alarms in the watchdog process through low-level system calls. The alarms are signaled by means of interrupts from the watchdog process and handled in assembly language. The result is the application program timing is not evident in the program itself, but instead is hidden in low-level code and other processes, making verification and modification of the timing difficult.

Real-time High-Level Languages. We now examine the ability to express timing constraints in four high-level real-time languages:

Ada. Although it uses the notion of time, Ada [US 83] has no constructs for explicitly expressing execution timing constraints. The expression of time is limited to a **delay** primitive which allows execution suspension for at least a specified time. The delay primitive is also used to specify message transmission deadlines which allows for communication timeouts. There is, however, no explicit support for specifying absolute start times, deadlines, and periods of tasks; no direct way of expressing nested timing constraints; and no direct way to handle variable constraints.

Modula-2. Modula-2 [Wir83] also supports the notion of time by allowing the programmer to explicitly specify scheduling through the use of "co-routines" which implicitly incorporate timing and do not explicitly express timing constraints in the program. This use of co-routines requires the programmer to maintain system timing using scheduling and precedence specifications to synchronize processes to meet timing constraints. This added burden on the programmer of having to schedule execution complicates real-time program development and modification.

PEARL. PEARL [Mar78] is a widely used language for real-time control applications which allows the expression of time. Absolute and relative start times, deadlines, and periods may be expressed. Pearl has an excellent facility for variable constraints using an abstract data type called "duration" with well defined rules for setting constraint values. Timing constraints may be nested, but since the scheduler is not part of the language, the semantics of the enforcement of these nested constraints are not clear. Pearl is not designed for distributed systems and thus lacks distributed synchronization and communication.

Real-Time Euclid. Real-Time Euclid [KS86] is a recent real-time language which incorporates the notion of schedulability into the language. The expression of process periods is handled through activation information provided with each process that specifies a period, and an absolute start time. The period specification mechanism may be used for a limited notion of deadlines assuming the deadline is the same as the period. It does not, however, provide explicitly for deadlines, start time intervals, nested constraints, or variable constraints. The language is also not designed for distributed systems.

Specification Language Timing Constraints Barbacci and Wing [BW86] describe a task level specification language which describes high level properties of tasks by integrating functional descriptions with timing constraints. The Larch specification language is used to describe the names, interfaces, pre-conditions and post-conditions of blocks in the language. Timing constraints including earliest and latest start time, event triggering, and durations of execution, both minimum and maximum, are specified. Their timing expression is complete for execution constraints, but since the timing expression is for a static specification language, not for a programming language, it does not provide for variable or nested constraints. Although their timing expression is designed for a specification language, it is expressive and does capture execution timing constraints well. Communication, input, and output constraints are not directly expressible.

Work on Expected Execution Times Worst case expected execution time is a form of timing constraint that should be enforced to efficiently detect errors. It should be possible for the programmer to specify worst case execution times, but not necessary. In the event of no specified expected execution time the compiler should generate a worst case estimate.

None of the above languages allow for programmer specified expected execution time and none of them enforce expected execution time as a constraint.

Real-Time Euclid does allow for calculation of worst case execution times for analyzing schedulizability of processes. The language places worst case bounds on execution times by eliminating some programming constructs and adding others. Recursion and dynamic data structures are eliminated because they make estimating execution time difficult. A construct is added to time bound loops so that arbitrary time of event-controlled loops is limited. Processes are also time bounded through period specifications that denote the end of the period as a worst case time bound for the process.

The refinements are valuable for calculating expected execution times and could easily be altered to be a form of timing constraint in addition to a schedulizability tool.

Yang [Yan86] and Wei [Wei81] also propose the use of compiler-generated expected execution time to aide in scheduling decisions. The expected execution time is arrived at using worst case estimates on loops (which are time bounded), conditionals, subprogram calls, branching, and I/O. The estimates are arrived at using execution times of the host processor and pessimistic worst case assumptions about operations, such as conditionals, that may have a varying number of operations. This work is also useful and can easily be altered to form an execution constraint bound.

In both Real-Time Euclid and the work of Yang and Wei the estimated execution time is often a very pessimistic worst case, possibly too pessimistic to be useful as a constraint. Although their methods do allow estimation of execution times, more realistic, possibly probablistic, expected execution times need to be derivable from the program in order for expected execution time to be a useful constraint.

We have seen that none of these languages treat the explicit expression of timing constraints completely, so as to make their incorporation into real-time programs an easy, natural reflection of the application specifications. Real-time Euclid and Pearl have made progress, but still lack capabilities in addition to not being designed for distributed applications.

3.2 Timed Communication

This section presents requirements of real-time distributed interprocess communication. We derive real-time communication requirements and examine how the work on distributed communication must be augmented to support real-time.

3.2.1 Timed Communication Requirements

Since real time systems must respond to external stimuli, capabilities must exist for fast communication among processes. Often more important than actual speed of communication is *predictably fast* communication. Predictably fast communication implies that the communication must adhere to some timing constraints that were assumed when the system was designed. In most cases these communication timing constraints are tight so that physically fast communication may be required, but as long as the communication is predictable within the assumed timing constraints the system will function as designed.

Predictable Communication. Since real-time systems must adhere to timing constraints, unbounded communication delays are not acceptable. It is desirable to have a communication medium which has predictable data transmission delays and communication primitives that bound delays in message queues. Predictably fast communication is supported by asynchronous messages and by concurrent shared data. Both forms of communication are low overhead and thus physically fast. Neither form requires waiting in a queue or buffer (or at most a small buffer for asynchronous messages) thus the delay of messages is predictable. Asynchronous messages may get lost and are thus unreliable, and concurrent shared data may not be consistent. Other forms of communication such as half-synchronous messages, full synchronous messages, and remote procedure calls can be made predictable by adding timing constraints to their protocols.

Time Constrained Communication. Meeting timing constraints is a topic not addressed by the traditional distributed communication primitives. Adding timing constraints to communication is important to add predictability to communication primitives. Also the dependency of hard real-time applications on time implies that messages often have a specific interval when they are valid. In the robot control processes for example, the set point which is communicated from the supervisor process to the joint processes is only valid during the sampling period in which it was generated. For the **send** primitive the ability to stamp a message with a validity time interval is required; for the **receive** primitive the ability to specify a deadline for waiting for a message is required.

3.2.2 Related Work on Time Constrained Communication

There has been some work done toward providing for time constrained communication. Lee and Gehlot [LG85] propose a set of real-time distributed communication primitives that allow for specifying an absolute or relative deadline on message reception. They touch on timed synchronous two-way communication and provide primitives to express it. Lee and Davidson [LD87] present a method for incorporating time to full synchronous communication using deadlines for message and reply reception.

The MARS system [KM85] introduced the idea of "time tagged messages" in a distributed cluster of processors. Each message comes with a time stamp indicating the interval it is valid for. The operating system is assumed to discard messages after their validity period is up. A process may also specify a deadline on message reception. Yang [Yan86] also incorporates time tagged messages to distributed real-time programming and proposes primitives for a non-blocking send and a blocking receive.

RNet [BMM87] provides for time constrained communication ports as its primary method of supporting timing constraints. In RNet, execution is modeled as processes started with message reception and terminated by message sending. All message communication is done to ports. Special ports are provided for sporadic process invocation, periodic process invocation, missed deadlines, and exceptions. All communication may be time constrained primarily through the receiving process. There is no provision for time stamping the validity of messages from a **send** operation.

The language ADA [US 83] provides for a time out on its synchronous communication "rendezvous". The single time primitive "**delay**" bounds the time an executing process is willing to wait for a rendezvous. Other real-time languages such as Real-Time Euclid, Modula-2, and PEARL do not provide for timed distributed interprocess communication.

3.2.3 Communication Requirements of Real-Time Process Interaction

Shin and Epstein [SE85] present four classifications of real time processes:

1. **Independent** - Independent process run independent of other processes in the system and almost never need to be synchronized. Communication, both sending and receiving, is dominated by asynchronous periodic updates. Messages are usually periodic and not critical. In the robot control example the dynamics process is independent. It runs on its own period, asynchronously receiving set points from the supervisor process and asynchronously sending updates to the servo processes.

2. **Loosely coupled** - Looslely coupled processes are characterized by small volume communication that often requires synchronization. Since the communication is low volume, efficiency considerations are usually not important, but communication must satisfy time constraints. The collector process in our example is loosely coupled with the joint process. It synchronizes the joint processes to collect the joint error information that is sent to the servo processes.

3. **Tightly coupled** - Tightly coupled processes are characterized by infrequent high volume communication. Communication is usually critical and must be quick. An example is the interaction between the supervisor process and a critical sensor process at a critical period of the application. During a critical period, such as grasping a moving object, the sensor process will require a fast period. This fast period implies that the communication with the supervisor is fast, and high volume.

4. **Serialized** - Serialized processes imply that one process depends on the completion and results of another. These processes require synchronization and communication which must be reliable, but not necessarily time critical. The supervisor process and joint processes are serial in our example. The joint processes must wait for the supervisor to generate a set point before performing their computation.

The class of independent processes don't necessarily require reliable or consistent communication. For independent processes, the predictable speed of concurrent shared memory and asynchronous message communication outweigh their disadvantages of reliability and consistency. If a message does not arrive or is erroneous, the system might lose performance, but as long as the next independent process update arrives within the receiver's timing constraints the application still functions correctly.

For loosely coupled processes a reliable and consistent form of communication is needed such as a half synchronous message passing scheme, where the receiver waits and the sender does not. Half-synchronous communication is more reliable than an asynchronous scheme because the receiver, waiting for a message, can detect that the message did not arrive by timing out; in asynchronous communication neither process waits, thus neither can detect that a message did not arrive. The advantage of half-synchronous over synchronous communication is that the sending process continues executing and has no arbitrarily long waits which effect predictability. Depending on the level of consistency required and the placement of the processes on the same node, concurrent shared memory may also be acceptable.

Some form of synchronization must be provided by the primitives of any distributed system and real time is no exception. Serialized processes require exact synchronization to ensure the order of process execution. Other classes of processes require some form of limited synchronization. Copper [CK85], however, claims that real-time applications are dominated by asynchronous communication.

Since synchronization may be achieved without explicit synchronous primitives, he feels that it is possible to exclude full synchronous primitives from real time operating systems.

From the discussion of real time requirements and the wide range of primitives provided by proposed real time systems, we see that no one form of communication is sufficient. The four classes of real time processes each have different communication requirements. Some form of asynchronous communication, either asynchronous messages or concurrent shared memory, is needed to provide the predictable speed needed in real time. In addition every distributed system needs some form of synchronization. All primitives should also provide predictable performance and must also have the ability to be time constrained to reflect the validity interval of the communication.

3.3 Enforcement of Timing Constraints

3.3.1 Goals for Enforcement of Timing Constraints

In order to relieve the programmer of the burden of ensuring that timing constraints are met in real-time programs, it is necessary to provide system support to enforce timing constraints. System support includes:

- The underlying system detecting violations of timing constraints and invoking appropriate action (see Section 3.4) when they are violated;
- The run-time system changing timing constraints due to nesting of constraints, propagation of constraints from a sending process based on the urgency of messages, and the setting of values for variable constraints;
- The run-time scheduler scheduling processes based on their current timing constraints so that all system constraints can be met.

Thus, to properly enforce timing constraints processes must be scheduled based on a system view of dynamic timing constraints, including nested and propagated timing constraints.

3.3.2 Related Work on System Enforcement of Timing Constraints

Several real-time operating system scheduling strategies have been proposed that are useful for timing constraint enforcement [LL73,Mar82,Nag80,ZRS87,RS84,SLR86]. These scheduling methods all use process deadlines and expected execution time for meeting timing constraints through scheduling based on their deadlines.

Naghibzadeh [Nag80] has proven that Earliest-Deadline-First-With-Preemption is the optimal scheduling strategy for uni-processor systems with dynamic priorities. His work is useful with our assumed model of a system with distributed multi-programmed nodes, dynamic constraints, and no process migration. Zhao, Ramamritham and Stankovic [ZRS87] discuss implications of the non-preemptive and preemptive scheduling and conclude that preemptive scheduling is better for meeting system constraints, but is harder to implement and incurs high overhead. They also describe a weighting function for determining the next process to run which incorporates other factors such as resource allocation in addition to straight deadlines.

Real-Time Euclid [KS86] incorporates the notion of system enforcement by making schedulability a main goal of their language. To achieve schedulability (their method is described in Section 3.6.2) they have added constructs to time bound most computation and removed features such as dynamic data types and recursion so that the system can get an accurate estimate of expected execution time of processes (their expected execution time was described in Sectin 3.1.2).

No work has been done explicitly to expand the area of traditional scheduling algorithms to include the handling of nested and propagated timing constraints. It is possible to add this notion to scheduling strategies that support variable timing constraints by allowing timing constraints of a process to be altered by a nested or propagated timing constraint. With a run-time system that is capable of setting variable constraints based on nested or propagated constraints, real-time scheduling methods such as those presented by Ramamritham and Stankovic [RS84] and Zhao et al. [ZRS87] can handle nested and propagated timing constraints.

3.3.3 Related Work on Programmer-Specified Scheduling

Some of the real-time languages mentioned in Section 3.1.2 integrate the notion of scheduling into high-level languages. These languages allow the programmer to explicitly control scheduling of processes.

Ada. Ada [US 83] requires very limited run-time support for scheduling. Task synchronization is done through "rendezvous" where tasks wait for each other before communicating. Task priorities may be specified. Since Ada has no method of specifying timing constraints, scheduling is not driven by timing constraints and thus is not used to enforce timing constraints.

SSS Co-Routines Berry et al. [BGMT82] describe a method called SSS which uses "co-routines" to allow the programmer to control scheduling. Co-routines are sequential processes which time-share a processor and synchronize through explicit synchronization primitives. Berry et al. contend that system-provided scheduling is non-deterministic and can not guarantee that timing constraints of real-time processes can be met. They allow the programmer to specify scheduling in a separate module of the program. This separation of code and scheduling concerns allows the code to be developed without reguard to timing constraints; timing constraint enforcement is then added through the scheduling module. Scheduling can be modified to allow for message reception. The scheduling is done through synchronization primitives that allow waiting and signaling conditions. A process may also resume other processes.

Berry's description does not discuss the use of actual timing constraints. Instead they assume that timing constraints will be met implicitly through synchronization of processes. Enforcing timing through synchronization complicates real-time programming by requiring the programmer to translate explicit constraints that are part of the application to implied synchronization primitives. Modifying constraints and verifying correctness in time is also difficult. Making the programmer responsible for all scheduling also further complicates program development.

Modula-2. Modula-2 [Wir83] also places a large part of the scheduling burden on the programmer by requiring him to explicitly schedule all co-routines through scheduling primitives. Modula-2 was designed for single processor systems so that processes are made quasi-concurrent by time sharing. Scheduling is handled in the processes through the use of the **transfer** primitive which explicitly transfers control of the processor to another process. As with the co-routines of SSS, Modula-2's co-routines complicate real-time program development, verification, and modification by forcing the programmer to translate explicit constraints to implicit scheduling.

None of these languages is acceptable for system enforcement of timing constraints because constraints are not explicitly expressed. If constraints are expressed, then scheduling algorithms such as those proposed in [RS84,CSK86], properly augmented to support dynamic constraints, can be useful in relieving the programmer from having to enforce timing constraints.

3.4 Time Fault Tolerance

All real-time control systems should be capable of handling all events, including faults and unusual environmental input, without loss of control. *Faults* are expected but highly undesirable outcomes that, if untreated, cause incorrect system behavior. Violation of timing constraints should be viewed as expected outcomes that must be handled to avoid incorrect system performance. Violations of timing constraints are called *time faults.* In order to provide complete control, the system must be *time fault tolerant* by handling missed timing constraints.

3.4.1 Goals for Time Fault Tolerance

If a time fault occurs for any of the time constraints present in the application, an action appropriate to the particular time fault should be invoked. For instance, if an input constraint of one of the sensor processes is missed, perhaps due to an extreme change in speed of a moving object being tracked, a possible action would be to send old or partially updated information. If an output constraint of the set point process is missed, then the robot should be stopped immediately.

It is possible to extend the notion of time fault tolerance to incorporate tolerance of hardware failures and software errors as well. The traditional approach to detecting these types of faults has been to identify data with bad values, either by having redundant processes vote on values or testing for known values. Values in real time systems have a time interval of validity associated with them and thus it is possible to detect faults not just by bad values but by bad timing. A timeout is used to detect faults in many existing systems. For instance if the supervisor process fails it would not send a set point to the joint processes. The joint processes would then miss their receive deadline and signal a fault by the supervisor process. By providing for time fault tolerance, a powerful method of treating not only time faults, but faults that can be detected by timeouts, is also provided.

3.4.2 Related Work on Exception Handling

There has been work done on exception handling in sequential programming that is applicable to handling time faults. The languages Ada [US 83], CLU [LS79], and Real-Time Euclid [KS86] are examples of languages that provide some form of exception handling capability for traditional software exceptions.

Both Goodenough [Goo75] and Yemini [YB85] discuss relevant issues concerning syntax and semantics for raising, handling, and propagating exceptions. They call for the following actions to be provided: resume, terminate or retry the signaling process, propagate the exception, or transfer control – all of which are viable actions for treating time faults.

Ada has an excellent facility for handling exceptions. Exceptions may be enumerated, control flow is switched from the process to the exception handler, and exceptions may be propagated from the process in which they occurred to a calling process. Ada's exception mechanism does not, however, help with time fault tolerance since timing exceptions (except limited timeout capability on communication) are not part of the language.

Klingerman and Stoyenko [KS86] present an exception handler in Real-time Euclid that provides for handling timeout exceptions. They identify the form of the timing constraint that was violated as part of the exception signal. Due to the limited timing constraints that are expressed by the language, the number of timing exceptions is also limited therefore limiting the flexibility of treating different timing violations. They do not provide actions specificly for dealing with time faults. Their exception handling, however, is a good mechanism for providing time fault tolerance since actions appropriate to all time faults they allow can be treated in a user-specified way.

Other than limited work in the RNet system [BMM87] and by Yang [Yan86] discussed later, no work has been done in handling timing exceptions in a distributed real-time environment where it might be necessary to propagate exceptions to other nodes.

3.4.3 Related Work on Process Redundancy

Systems such as the real-time kernel described by Natarajan and Jian [NJ] use redundant processes to detect faults and provide fault tolerance. The processes perform parallel, identical computation and vote on the result. The advantages of redundancy is that there is a higher probability of a good process surviving and recovery is quick. The disadvantage is the high overhead due to replication of computation and coordination of the redundant task execution in a distributed environment.

Natarajan and Jian address some of these disadvantages in their work. The issue of the meaning of timing constraints on the redundant processes is not addressed, and this work is not directly applicable to time fault tolerance.

Time fault tolerance is important for ensuring a reliable real-time system. No current real-time languages support high-level constructs for providing time fault capability. What is required is an investigation of adding time fault handling to traditional fault tolerant techniques such as exception handlers and process redundancy.

3.5 Maintaining Consistency in Distributed Real-Time Systems

Another feature needed for high-level treatment of time in a distributed environment is the ability to incorporate time into the process of ensuring that system states are consistent. An *invariant constraint* is a predicate, required as part of the application, which must be true for correct performance. A system state which satisfies all invariant constraints is said to be *consistent.* In distributed systems these invariant constraints usually take the form of *integrity constraints* which place restrictions on the data stored in the system. In distributed real-time systems, there are additional constraints – timing constraints – which are invariant constraints that must be adhered to for a system to perform correctly. Thus, state consistency in distributed real-time systems must be expanded with the notion that a state must be reached within its timing constraints to be consistent. Consistency that incorporates timing constraints is called *consistency in the time domain.*

3.5.1 Timed Atomic Actions

The state of a distributed system depends on the states of individual nodes. It is possible that, when changing the system state, a node may take the system to an inconsistent state before completing its action and creating a new consistent state. To guarantee a consistent global state, some changes to the system state must be carried out completely or not carried out at all so as not to leave the system in an inconsistent state. The traditional solution is to provide *atomic actions* with the property that the actions either complete entirely or have no effect. Atomic actions which appear to have no effect in the event of a fault are said to be *atomic with respect to faults.* Adding atomicity with respect to time faults leads to the notion of a *timed atomic action* for ensuring state consistency in the time domain for distributed real-time program development.

A timed atomic action either performs completely within its timing constraints or appears as if it never executed. For instance, the joint actuation of the robot manipulator has the invariant constraint that all six joints must be actuated correctly within the same 1 ms deadline. It is therefore desirable to have all of the joint actuation be a compound timed atomic action so that either all six joints actuate correctly within their common deadline or none actuate. This use of a timed atomic action avoids the problem of five joints executing correctly within their deadlines, but having the sixth violate its invariant timing constraint creating a state which is inconsistent in the time domain. The result of the inconsistent state could cause the manipulator to unexpectedly strike an object that the sixth joint would have made it avoid.

Although it may be possible to implement timed atomic actions at the application level, the program structure becomes complex and incurs high overhead. The notion of timed atomic action should be provided as a basic primitive for supporting a time fault tolerant system.

3.5.2 Related Work on Distributed Consistency

No work has been done on incorporating the notion of time into distributed state consistency and no real-time languages provide any form of high-level constructs related to distributed state consistency in the time domain. Work which has been done on traditional distributed consistency without the notion of time does, however, provide insight into the goals of obtaining consistency in the time domain. We now examine some of that work. Due to our assumed model we omit discussion of methods that involve special or redundant hardware and process migration.

Atomic Transactions. Liskov, [Lis82], Lampson [LPS86] and Bennett [Ben84] discuss the notion of *atomic transactions* for distributed state consistency. An atomic transaction (or *atomic action*) is composed of one of more actions, with the transaction having the property that the transaction is executed entirely or not at all. This atomic property ensures that the system is transformed from one consistent state to another. They assume that atomic actions may be "undone", which is not the case in real-time control applications where output affects the environment. The notion, however, is still useful for timed atomic actions when time is incorporated into the definition of consistency.

Moss [Mos85] does an excellent job of extending the notion of atomic transactions to nested transactions where atomic transactions are nested within other atomic transactions. He presents a version of the *two phase commit protocol* which is useful for compound concurrent atomic transactions across a distributed system and discusses how restoration to a consistent state can be done when nested transactions are used. Since timing constraints can be nested, Moss's work is applicable to timed atomic actions once the notion of time is incorporated into the definition of a system's consistent state.

Ramamritham and Stankovic [RS84,CSK86] have investigated schedulers which determine the node to run a process on (assuming process migration is possible). They assume an expected execution time is available and determine if a given process can execute within that time on a particular node. This work is useful for implementing timed atomic actions where it is necessary to determine if an action can perform within its deadline before executing irrevocable operations.

Recovery Blocks. A *recovery block* is a construct for aligning computation to ensure software consistency. In a recovery block, blocks of code are ended with a software acceptance test that must be passed to ensure that the computation adheres to consistency constraints. If the test fails, the process falls back to the state before the recovery block and a different method of computing the same result is tried. Only after the acceptance test is passed can the process(es) proceed out of the block. Recovery blocks [RLT78,Lom77] for uniprocessor machines have been extended to distributed computing in TCN [Kim84] and CMS [Wel83] using variations on this scheme. Randell [Ran75,Shr85] and Yang [Yan86] also conceptually extend the notion of recovery blocks to distributed systems by using the idea of a *conversation scheme* which links recovery blocks to ensure that related blocks either all pass the acceptance test or all are retried.

Time in recovery blocks is mentioned briefly by Kim [Kim84] who states that a watchdog process may be used along with recovery blocks. However it is not a built in feature of recovery blocks and is not considered an attribute of the state consistency which recovery blocks aim to achieve.

Recovery blocks can used to implement atomic actions, as well as supplement them with additional capabilities. If the recovery block fails to pass its acceptance test it falls back to a previous state, thus it either executes correctly or does not execute. This is semantically equivalent to an atomic action. In addition to not executing the action, the recovery block provides for alternate actions that are tried upon failure, a capability that atomic actions do not provide. This extra capability, however, adds restrictions to distributed programming since the only alternative is to try another method of computing the new state. This is more restrictive than allowing an exception handler to handle the case that an atomic action does not execute. Randell [Shr85] gives a comparison between exception handling and the recovery block approach to software consistency. His discussion shows that exception handlers are a more general form of providing software consistency. Recovery blocks, he points out, are a more formal and structured construct and more natural to use, but provide only a proper subset of the power of exception handlers.

Ensuring consistency is an important part of programming in a distributed environment where independent processes are acting concurrently. Constructs such as atomic actions and recovery blocks have provided a high-level, structured way of dealing with consistency in traditional distributed programs. As we saw from the example of the actuation of the six joints, timing constraints must also be considered in determining consistent states. For this reason the incorporation of time into traditional techniques such as atomic actions and recovery blocks is needed to produce reliable, consistent distributed real-time programs.

3.6 Static Timing Verification

3.6.1 Goals For Static Timing Verification

As a final feature we would like to be able to provide offline static verification of real-time programs. It is difficult to debug a real-time control system on the actual application because the loss of control that might result from system bugs could cause damage. Also the program may appear to be correct, but due to an unanticipated timing error, will perform incorrectly after testing. The robot arm, for instance, could be damaged by striking an object as the result of a control program timing error that was not detected by traditional debugging tools. Often real-time control systems must be tested and debugged before attempting to use them in an actual application. Since hard-real-time requires timing to be correct for correct system performance, it is desirable to be able to perform *static timing verification.* Static timing verification tools examine the timing that was expressed and determine if the timing constraints can be met. Since timing depends on system conditions, such as the environment of the robot, it is useful for the timing verification tool to provide information explaining what (if any) of a set of system conditions causes timing to be violated. For instance, the verification tool should be able to help to place a bound on the maximum speed of a moving object that the robot can track based on the fastest allowable period that the sensor processes can operate at. With information concerning what environmental conditions cause time faults, time fault tolerance may be added to the system to handle the conditions, or the conditions can be designed out of the application.

3.6.2 Related Work on Static Timing Analysis

There has only been limited work in real-time static verification (see [GL83] for a brief overview of early verification work). The two principal methods for performing verification are analysis, where assertions about the relationships among constrained processes are tested; and simulation, where an abstract model of the real-time process interaction is simulated to detect violations.

Simulation-based Analysis The RNet system [BMM87] (described further in section 3.7.2) has implemented a simulation-based static verification tool that checks timing of a specification language. First the worst case real-time behavior of every periodic and sporadic task is defined. The scheduling algorithm is then simulated by performing every task activation relationship, with violations of the constraints being reported. This static verification technique often makes pessimistic assumptions, particularly for sporadic processes. RNet's exhaustive search method is also complex for systems with many process interactions. There is no provision for variable or nested constraint verification or identification of conditions that cause violations.

Analytical Techniques Some analysis has been done concerning assertions about real-time behavior [Yan86,Wei81,LY86,Haa81]. This type of analysis requires a known, often worst case, bound on the execution time of processes. Yang [Yan86], Wei [Wei81], and Real-time Euclid [KS86] have all made significant advances in eliminating or bounding programming constructs of unknown time as discussed in Section 3.1.2. These bounding techniques are usually either limiting, such as Real-time Euclid's elimination of recursion and dynamic data structures; or pessimistic such the bounds placed on loop statements; but necessary for creating analyzable programs.

Real-time Euclid uses the information provided by expected execution times to analyze schedulizability of real-time programs. They employ an equation which relates CPU requirements of interruptable and non-interruptable parts of program segments, communication and I/O time, blocking and waiting time, process periodic timing constraints , and relative speed of the processes. Allowable process periods can then be determined given the rest of the information which can be derived from the bounded language and hardware characteristics. The equation can determine schedulizability assuming that blocking can indeed be estimated. The analysis may, however, suffer from the worst case assumptions necessary in time bounding computation. They do not extend their analysis to distributed programming.

Yang [Yan86] first develops rules for bounding execution times of language constructs in Concurrent-Pascal and then proposes a set of assertions concerning deadlines and expected execution times that the verification tool checks. These assertions, however, are not rigorous, only do simple checking, can not do checking in a distributed environment, and make no provision for checking which system conditions cause time violations.

Jahanian and Mok [JM86] address the problem of formalizing analysis of determining if timing specifications meet safety assertions in real-time programs. They model computation in an event-action model where events cause actions to transform the system state. This model is then transformed to first order logic called Real-time Logic (RTL) which has time incorporated. RTL has a clean syntax for expressing system specifications of time, events, and nesting of actions. In a later paper [JM87] Jahanian and Mok discuss a graph-theoretic approach for analyzing RTL specifications to determine if they meet safety assertions. Although the event-action model is expessive for system specification, it does not reflect system operation and the result is that the associated analysis techniques are not directly applicable to offline verification of timing constraints. Due to the expressibility of RTL, an extension to formal verification of timing constraints is possible. Barbacci and Wing [BW86] translate specifications in their task-level specification language (described in section 3.1.2) into RTL so that specifications can be analyzed.

3.7 Integration of Features

3.7.1 Goals of Integrating Features

The characteristics of the previous six sections provide a distributed real-time programming environment where program development is easier, with programs being verifiable, fault tolerant, and able to ensure consistent behavior – all in the time domain as well as the typical functional domain. In order to provide an environment which allows a high-level incorporation of timing, the six characteristics must be integrated in a system which provides:

- A high-level language which expresses timing constraints, naturally and explicitly. The language should also specify explicitly what action to take when timing constraints are violated. Along with the language, an associated compiler is needed to interpret the high-level constructs, extract timing constraints, and determine expected execution time.

- An operating system, capable of providing timed communication, detection of time faults, enforcement of timing constraints, and timed atomic actions.

- A verification tool capable of statically verifying system timing and evaluating conditions that would cause violated timing.

3.7.2 Related Work on Integrated Systems

We now examine two proposed integrated approaches to real-time programming which involve adding timing expression to existing distributed programming languages and developing system-level support tools.

RNet Programming System. RNet [BMM87] is a system for developing hard real-time programs for distributed systems, which provides an integration of high-level expression of timing constraints and system level support tools to handle the enforcement and verification of the constraints. Applications are developed in Real-time EUCLID augmented by RNet constructs for expressing timing constraints. The system provides run-time scheduling based on timing constraints and static timing verification.

In RNet, execution is modeled as processes started with message reception and terminated by message sending. All message communication is done via ports. Special ports are provided for sporadic process invocation, periodic process invocation, missed deadlines, and exceptions. All communication may be explicitly time constrained. Computation is implicitly time constrained by modeling process execution as computation between a **receive** and a **send**. When the **receive** is constrained to happen by a certain time, the computation is implicitly constrained to start by that time. Similarly, when the **send** must happen by a certain time, the computation has an implied deadline. RNet's model naturally supports data flow methods of programming. Data flow programming, however, requires work on the part of the programmer to translate the real-time execution constraints specified in the application to implied computation time constraints associated with the communication. It is also not possible to express nested constraints, constraint intervals, or variable constraints because they are not provided for in RNet's constructs.

RNet provides a deadline driven scheduler with a modified Earliest-Deadline-First-With-Preemption scheduling algorithm. RNet's scheduler takes advantage of the explicitly expressed constraints and is satisfactory at providing one-level constraint enforcement. Since RNet's constructs do not provide for nested, propagated, or variable timing constraints, it is not clear that the RNet scheduler can enforce them.

Time fault tolerance is provided by a timing exception port. The exception port receives a time-out signal from the kernel and invokes a sporadic process to handle the exception. Although the port-based exception mechanism is useful, by having another process to handle the exception, important state information of the executing process is lost. State information must be retained by handling the exception in the process itself. RNet's exception mechanism can not determine which constraint caused the port to be signaled, which makes invoking action appropriate to that constraint difficult.

RNet makes no provision for preserving state consistency nor for traditional fault tolerance.

Another feature of RNet is a static timing verification tool. It uses worst case execution times, the explicit constraints, and applies a simulated scheduling algorithm. The static verification tool is a useful addition to the system, it is currently being refined to provide more accurate analysis.

Yang's Approach. In his dissertation Yang [Yan86] describes an integrated system capable of handling real-time program development. He adds timing specification constructs to Concurrent Pascal, describes real-time scheduling support, fault tolerance, and static timing verification.

Yang's description of timing constraints is similar to that of Lee and Gehlot[LG85] (which provided preliminary work for this report). He arrives at language constructs to explicitly express start-time constraints, termination deadlines, and periods. He reduces all possible primitives to a set of primitives (equivalent to those given in [LG85]) that completely express all of timing constraints described in Section 3.1.1. He shows that the set is orthogonal. He also provides for time constrained communication. His description does not elaborate on the semantics of nested constraints, nor variable constraints.

Two scheduling hueristics are described, one based on static priorities and another on dynamic conditions. The static scheduler is not useful in a real time environment with variable constraints. The dynamic scheduler uses Earliest-Deadline-First-With-Preemption and is a good enforcement tool. Yang does not mention how his scheduling strategy handles nested constraints, propagated constraints, or variable constraints.

Although fault tolerance and consistency is thoroughly discussed by presenting a "conversation" scheme that ensures software consistency, time fault tolerance is not treated. Time is not used as a factor in determining consistency in the conversation scheme. Atomic actions are possible with the conversation scheme, but timed atomic actions are not explicitly provided for. Timing exceptions are handled by an "else" clause of a timed Pascal statement; this "else" clause is barely described and does not have the robustness of a full exception handler with the ability to invoke state recovery action, propagate exceptions, signal other exceptions, or handle exceptions by default.

Yang also presents methods for determining worst case execution time of processes and using it to analyze timing properties. The work is summarized in Sections 3.1.2 and 3.6.2

4 Current Research Directions

Our current approach to incorporating time as a first class entity consists of extending the *temporal scope* language construct presented by Lee and Gehlot in [LG85] to provide for the features outlined in the previous section. This extension is resulting in the development of the *timed action* concept. We have also been developing a model of distributed real-time programming to identify and describe abstract properties that the eventual language construct should have.

Our model of a distributed real-time process is an initial state, a series of sequential and/or concurrent actions each transforming the system to a consistent state and then to a final (consistent) state. Consistency is determined by integrity constraints imposing physical constraints of the application on state variables, sequencing constraints on the state transitions, and timing constraints on the states (*lifetimes* of state variables) and on the transitions. For complete state transitions, actions must be provided for when consistency is violated.

From the model we have derived the *timed action* language construct. A timed action is a programming module (block) that groups statements based on modularity considerations and explicitly expresses timing constraints on the statements. A timed action reflects the notion of an action in the model with the following characteristics:

- A timed action allows programs to be easily created from application specifications.
- A timed action provides explicit, natural expression of timing information on input, output, execution, and communication. Timing constraints include deadline, period, start time, as well as scheduling information such as priority. The deadline, period, and start times may be implemented as variables with their values set at run-time. A method for declaring the range of the time variable is provided.
- Timed actions provide time fault tolerance capabilities and a structured method for ensuring consistent behavior of the program by providing a transition from one distributed system state that is consistent in the time domain to another. This transition is denoted by the construct expressing (possibly variable) timing constraints on a body of actions. The body of actions may contain nested timed actions. When performed on time, the actions transform the system to a consistent state. Atomic actions, including timed atomic actions are provided as a basic part of the primitive. Each timed action handles time faults through an exception handler similar to those provided for in Ada [US 83]. The exception handler is designed to recognize a series of time related exceptions in addition to usual exceptions. Special consistent state recovery actions are provided to restore consistency and provide time fault tolerance.
- When implemented, the construct will be able to use current real-time scheduling and verification hueristics.

The TIMIX [Kin87] real-time kernel is also being developed to be integrated with the high level language to provide system enforcement of timing constraints as well as traditional distributed kernel support such as communication. TIMIX possesses an alarm package capable of detecting timing events and a provides for signal handlers to specify action in the event of an alarm signal. Time constrained communication is also supported.

Static time verification tools are also being developed to take advantage of the expression of constraints and expected execution time [ZL86,DL87]. The verification tools check which system conditions cause system timing constraints to be violated. Those conditions that cause timing to be missed are handled as exceptions, or the system is redesigned to alleviate timing violations.

5 Conclusion

This report has presented six features:

1. Explicit, natural expression of timing constraints on execution, communication, input, and output; including variable and nested constraints;
2. Provision for timed communication, which is predictable and may be time constrained, to support all classes of real-time processes;
3. System Enforcement of timing constraints including detection of timing violations and scheduling to meet dynamic timing constraints that may be variable, nested, or propagated.
4. Time fault tolerance that provides action appropriate to a particular time fault and the current system state;
5. Distributed state consistency in the time domain; and
6. Static verification of timing constraint compliance.

In examining related work we have found no system that provides for all of these six features.

By creating appropriate language constructs such as timed actions for incorporating time into traditional distributed programming, we can create tools which allow for timing to be treated on a high-level. We are implementing the timed action construct through close interaction with the TIMIX distributed real-time kernel and other run-time support tools. With this integrated approach, the programmer no longer must be concerned with how timing is to be met; he/she simply expresses the constraints naturally as deadlines and periods, which are part of the application specification, and allows the system to handle the timing. This high-level treatment of timing means that the development, verification, and modification of reliable distributed real-time systems is easier.

References

[Ben84] K. Bennet. *Distributed Computing*, chapter Mechanisms For Distributed Control, pages 179–192. Academic Press Publishers, 1984.

[BGMT82] D.M. Berry, C. Ghezzi, D. Mandrioli, and F. Tisato. Language constructs for real-time distributed systems. *Computer Languages*, 7(1):11–22, 1982.

[BMM87] C. Belzile, G. MacEwen, and G. Marquis. Rnet: a hard real-time distributed programming system. *IEEE Transaction on Computers*, C-36(8), August 1987.

[BMR83] Gerard Berry, Sabine Moisan, and Jean-Paul Rigault. Esterel: towards a synchronous and semantically sound high level language for real time applications. In *Real-Time Systems Symposium*, 1983.

[BW86] M.R. Barbacci and J.M. Wing. *Specifying Functional and Timing Behavior fro Real-Time Applications.* Technical Report ESD-TR-86-208, Carnegie Mellon University, December 1986.

[CK85] Albert Copper III and John Kearns. Real-time distributed control with asynchronous message reception. In *Real-Time Systems Symposium*, IEEE Computer Society, December 1985.

[CMMS79] David Cheriton, Michael Malcolm, Lawrence Melen, and Gary Sager. Thoth, a portable real-time operating system. *Communications of the ACM*, 22(2):105–115, February 1979.

[CSK86] S. Cheng, J. Stankovic, and K.Ramamritham. Dynamic scheduling of groups of tasks with precedence constraints in distributed hard real-time systems. In *Real-Time Systems Symposium*, pages 166–174, IEEE Computer Scicety, 1986.

[DL87] Susan Davidson and Insup Lee. Static timing analysis based on intervals. 1987. in preparation.

[DMV82] Mark Drummond, Clare Mcmullen, and R. Vasudevan. Packrat - a real time kernel for distributed systems. In *Real-Time Systems Symposium*, pages 151–154, IEEE Computer Society, December 1982.

[FR86] R. Fitzgerald and R. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, May 1986.

[GL83] Virgil Gligor and Gary Luckenbaugh. An assessment of the real-time requirements for programming environments and languages. In *Real-Time Systems Symposium*, pages 3–19, IEEE Computer Society, December 1983.

[GLR82] Patricia Garetti, Pietro Laface, and Silvano Rivoira. Modosk: a modular distributed operating system kernel for real-time process control. *Microprocessing and Microprogramming, The EUROMICRO Journal*, 9:201–213, April 1982.

[Goo75] J.B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(2), December 1975.

[Haa81] Volkmar Haase. Real-time behavior of programs. *IEEE Transactions on Software Engineering*, SE-7(5):494–501, September 1981.

[JM86] Farnam Jahanian and Aloysius Mok. Safety analysis of timing properties in real-time systems. *IEEE Transaction on Software Engineering*, SE-12(9):809–904, September 1986.

[JM87] Farnam Jahanian and Aloysius Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36(8):961–975, August 1987.

[Kim84] K. H. Kim. Distributed execution of recovery blocks: an approach to uniform treatment of hardware and software faults. In *4th International Conference on Distributed Computing Systems*, pages 526–532, May 1984.

[Kin87] Robert King. *Design and Implementation of a Distributed Real-Time Kernel.* Technical Report in preparation, Department of Computer and Information Science, University of Pennsylvania, 1987.

[KM85] H. Kopetz and W. Merker. The architecture of mars. In *FTCS-15*, pages 274–279, June 1985.

[KS86] E. Klingerman and A. Stoyenko. Real-time euclid: a language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9), September 1986.

[LD87] I. Lee and S. Davidson. Adding Time to Synchronous Process Communications. *IEEE Transactions on Computers*, August 1987.

[Lee84] Insup Lee. A programming systems for distributed real-time applications. In *Real-Time Systems Symposium*, IEEE Computer Society, December 1984.

[LG85] I. Lee and V. Gehlot. Language Constructs for Distributed Real-Time Programming. In *Proc. IEEE Real-Time Systems Symposium*, Dec. 1985.

[Lis82] B. Liskov. On linguistic support for distributed programs. *IEEE Transactions on Software Engineering*, SE-8(3):203 – 210, May 1982.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multi-programming in a hard-real-time environment. *Journal of the ACM*, 46 – 61, January 1973.

[Lom77] D. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *ACM Conference on Language Design for Reliable Software*, pages 128–137, March 1977.

[LPS86] B. W. Lampson, M. Paul, and H. Siegert, editors. *Distributed Systems Architecture and Implementation*, chapter Atomic Transactions, pages 246 – 265. Springer-Verlag, 1986.

[LS79] B.H. Liskov and A. Snyder. Exception handling in clu. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, November 1979.

[LY86] D. Leinbaugh and M. Yamini. Guaranteed response times in a distributed hard-real-time environment. *IEEE Transactions on Software Engineering*, SE-12(12):1139 – 1144, December 1986.

[Mar78] T. Martin. Real-time programming language pearl - concept and characteristics. In *Proc. COMPSAC, Chicago*, pages 301–306, 1978.

[Mar82] C. Martel. Preemptive scheduling with release times, deadlines and due times. *Journal of the ACM*, 812–829, July 1982.

[Mos85] E. Moss. *Nested Transactions, An Approach to Reliable Distributed Computing.* The MIT Press, 1985.

[Nag80] M. Naghibzadeh. *Analytic Design and Verification of an Overrun-Free Uniporcessor and Multi-processor Real-Time Computing System.* PhD thesis, University of Southern California, 1980.

[NJ] N. Natarajan and T. Jian. Kernel mechanisms for distributed real-time programs. Received in 1986 and presented at IEEE Workshop on Real-time Operating Systems in 1987.

[PDM86] R. P. Paul, H. F. Durrant-Whyte, and M. Mintz. A robust, distributed sensor and actuation robot control system. In *Proc. Third International Symposium on Robotics Research*, pages 93–100, MIT Press, 1986.

[PZ84] Richard Paul and Hong Zhang. Robot motion trajectory specification and generation. In *ISRR Proceedings - Japan*, 1984.

[PZ85] Richard P. Paul and Hong Zhang. Design of a robot force/motion server. In *International Conference on Robotics and Automation*, IEEE, 1985.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, May 1975.

[RLT78] B. Randell, P.A. Lee, and P. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2):123–165, June 1978.

[RS82] S. Rivoira and A. Serra. A multimicro architecture and its distributed operating system for real time control. In *Third International Conference on Distributed Computing Systems*, pages 238–246, October 1982.

[RS84] K. Ramamritham and J.A. Stankovic. Dynamic task scheduling in distributed hard real-time system. *IEEE Software*, 1, July 1984.

[SBWT85] K. Schwan, T. Bihari, B. Weide, and G. Taulbee. Gem: operating system primitives for robots and real-time control systems. In *International Conference on Robotics and Automation*, pages 807–813, IEEE, 1985.

[SE85] Kang G. Shin and Mark Epstein. Communication primitives for a distributed multi-robot system. In *International Conference on Robotics and Automation*, 1985.

[Shr85] S. Shrivastava, editor. *Reliable Computer Systems.* Springer-Verlag, 1985.

[SLR86] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Real-Time Systems Symposium*, pages 181–191, December 1986.

[TH86] F. Tuynman and L.O. Hertzberger. A distributed real-time operating system. *Software–Practice and Experience*, 16(5):425–441, May 1986.

[US 83] U.S. Department of Defense. Ada Programming Language. 1983. ANSI/MIL-STD-1815A-1983.

[Wei81] A. Wei. *Real-Time Programming With Fault Tolerance.* PhD thesis, University Of Illinois, 1981.

[Wel83] H. Welch. Distributed recovery block performance in a real-time control loop. In *Real-Time Systems Symposium*, pages 268 – 276, December 1983.

[Wir83] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, New York, 1983.

[Yan86] S. M. Yang. *Timing Specification and Verification For Fault Tolerant Distributed Computer Systems.* PhD thesis, University of Southern Florida, 1986.

[YB85] S. Yemini and D. Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2):214–243, April 1985.

[ZL86] Amy Zwarico and Insup Lee. *A Syntax and Semantics For Deterministic Real-Time Computing.* Technical Report MS-CIS-86-70, Department of Computer and Information Science, University of Pennsylvania, 1986.

[ZRS87] W. Zhao, K. Ramamritham, and J.A. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Transactions on Software Engineering*, SE-13(5):564–577, May 1987.