September 1988

Formally Integrating Real-Time Specification: A Research Proposal

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

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

Richard Gerber
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-84.

Formally Integrating Real-Time Specification: A Research Proposal

Abstract

To date, research in reasoning about timing properties of real-time programs has considered specification and implementation as separate issues. Specification uses formal methods; it abstracts out program execution, defining a specification that is independent of any machine-specific details (see [I, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] for examples). In this manner, it describes only the high-level timing requirements of processes in the system, and dependencies between them. One then typically attempts to prove the mutual consistency of these timing constraints, or to determine whether the constraints maintain a safety property critical to system correctness. However, since the model has abstracted out machine-specific details, these correctness proofs either assume very optimistic operating environment (such as a one to one assignment of processes to processors), or make very pessimistic assumptions (such as that all interleavings of process executions are possible). Since neither of these assumptions will hold in practice, these "predictions" about the behavior of the system may not be accurate.

The implementation level captures this operating environment: a real- time system is characterized by such things as process schedulers, devices and local clocks. However, advances here have been primarily in scheduling theory (examples of which are [15, 16]) and language design (examples of which are [15, 16, 17, 18,19,20]). Unfortunately, since formal models have not been used at this level, proofs of time-related properties cannot be made. To construct these proofs, we must show that an implementation is correct with respect to a specification; timing properties that can be shown to hold about the specification will therefore be known to hold for the implementation. We therefore need to represent the implementation formally so as to prove that the implementation satisfies the specification. The proof of satisfaction requires a well-defined formal mapping between the implementation and specification models.

We therefore propose to develop an integrated bi-level approach to the problem of reasoning about timing properties of real-time programs. At the specification level, we will use the *Timed Acceptances* model, a logically sound and complete axiom system which we have recently developed [21]. Using this model, the effect of interaction among time dependent processes can be precisely specified and then analyzed. We will then develop a formal implementation model (similar to the specification model) which captures operational behaviors: for example, the assignment of processes to processors, assumptions about scheduling and clock synchronization, and the different treatment of execution and wait times. A mapping will then be formulated between these two layers.

The bulk of our proposed work will be to formulate the implementation layer and define a mapping between it and the specification layer. We also need to continue work on the *Timed Acceptances* model to facilitate its use as a specification model, and to provide "hooks" for mappings between the two layers.

The rest of this proposal is organized as follows. The next section overviews related work in formal specification models. Section 3 describes our current specification model and proposed enhancements. We also detail the proposed implementation model, and required properties of the mappings between the two models. Section 4 provides a summary of the proposed research, and a yearly plan.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-84.

FORMALLY INTEGRATING REAL-TIME SPECIFICATION AND IMPLEMENTATION: A RESEARCH PROPOSAL

Insup Lee, Susan B. Davidson and Richard Gerber

MS-CIS-88-84
GRASP LAB 160

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

October 1988

Acknowledgements: This research was supported in part by NSF grants IRI86-10617, 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.

1 Introduction

To date, research in reasoning about timing properties of real-time programs has considered specification and implementation as separate issues. Specification uses formal methods; it abstracts out program execution, defining a specification that is independent of any machine-specific details (see [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] for examples). In this manner, it describes only the high-level timing requirements of processes in the system, and dependencies between them. One then typically attempts to prove the mutual consistency of these timing constraints, or to determine whether the constraints maintain a safety property critical to system correctness. However, since the model has abstracted out machine-specific details, these correctness proofs either assume very optimistic operating environment (such as a one to one assignment of processes to processors), or make very pessimistic assumptions (such as that all interleavings of process executions are possible). Since neither of these assumptions will hold in practice, these "predictions" about the behavior of the system may not be accurate.

The implementation level captures this operating environment: a real-time system is characterized by such things as process schedulers, devices and local clocks. However, advances here have been primarily in scheduling theory (examples of which are [15, 16]) and language design (examples of which are [15, 16, 17, 18, 19, 20]). Unfortunately, since formal models have not been used at this level, proofs of time-related properties cannot be made. To construct these proofs, we must show that an implementation is correct with respect to a specification; timing properties that can be shown to hold about the specification will therefore be known to hold for the implementation. We therefore need to represent the implementation formally so as to prove that the implementation satisfies the specification. The proof of satisfaction requires a well-defined formal mapping between the implementation and specification models.

We therefore propose to develop an integrated bi-level approach to the problem of reasoning about timing properties of real-time programs. At the specification level, we will use the *Timed Acceptances* model, a logically sound and complete axiom system which we have recently developed [21]. Using this model, the effect of interaction among time dependent processes can be precisely specified and then analyzed. We will then develop a formal implementation model (similar to the specification model) which captures operational behaviors: for example, the assignment of processes to processors, assumptions about scheduling and clock synchronization, and the different treatment of execution and wait times. A mapping will then be formulated between these two layers.

The bulk of our proposed work will be to formulate the implementation layer and define a mapping between it and the specification layer. We also need to continue work on the *Timed Acceptances* model to facilitate its use as a specification model, and to provide "hooks" for mappings between the two layers.

The rest of this proposal is organized as follows. The next section overviews related work in formal specification models. Section 3 describes our current specification model and proposed enhancements. We also detail the proposed implementation model, and

required properties of the mappings between the two models. Section 4 provides a summary of the proposed research, and a yearly plan.

2 Related Work

In this section we briefly survey related work in real-time modeling. Because there are no formal models for the implementation level, we limit our attention to specification models and point out perceived weakness.[1]

Specification models range from the very abstract, where there is no notion of a process or even a program, to more structure-oriented, which permit high-level specification of real-time processes and provide semantics for their parallel composition. Many of the methods discussed here are quite complex, and we attempt only to cover their most salient features. There are some concepts, however, which we strive to emphasize. To be effective, a specification model should incorporate both a sound and complete set of axioms. Without this, verification is not possible. Also, the model must possess a high enough level of abstraction so that most implementation details remain hidden. Furthermore, there should be some notion of "containment." With this feature, properties proved consistent with a set of behaviors are also consistent with its subsets. This is the essence of abstraction, and is fundamental for adequate specification models [23]. Thus, models based on process structure require a process containment relation, while those based on logics employ the implication connective. Finally, a specification method should possess a realistic execution model.

RTL: Jahanian and Mok [1, 2] have developed a highly abstract model called Real-Time Logic (RTL), which represents timing constraints in order to prove safety properties. RTL is essentially a typed, first-order logic containing constants which represent actions, events and transitions. Also included are functions which represent the occurrence times of events, predicates on the system states during time intervals, as well as axioms to describe the properties of events. The objective of RTL is to specify a system's timing constraints, and then show that they satisfy a specific safety property, also expressed in RTL. To achieve this, a graph-theoretic procedure is used.

The structure of the logic results in some very complex specifications of even simple processes. Therefore, a specification tool is provided, called the *Event–Action* model. It permits a high level specification of the system's timing requirements and a method of translating it into the corresponding RTL formulas. Using the event–action model, a system designer denotes the relevant actions and events that occur during execution. Also defined are the predicates that describe the state of the system and the timing constraints relating the actions, events and states. These constraints include assertions about the absolute timing of events and the ordering of actions.

[1] A survey of real-time languages (syntactic implementation models) can be found in [22].

We believe that RTL is a successful, though limited, model. It succeeds in providing a means by which safety assertions can be proved, a feature that few of the other models can claim to possess. However, it is not apparent how one can prove that the system's timing constraints are *mutually* consistent. Moreover, RTL suffers from being *too* high-level. The event-action model does not provide constructs necessary for expressing the *structure* of a system at even a rudimentary level. For example, it does not possess the means for representing a choice between alternative actions; furthermore, it does not have a way to express nondeterminism. Because the model lacks the notion of system structure, it would appear difficult to map specifications to a lower, and more operational level.

DRTL: DRTL [3] is an extension of RTL, which was defined to reflect the constraints placed on message-passing systems. Akin to the event-action model, there is a high-level specification language called RReq [24]. One of the more intriguing goals of this work is to integrate it with the group's implementation language, RNet. Rnet, however, lacks formal semantics.

Quantified Temporal Logic: Temporal logic [4] is a modal logic that expresses the development of situations in time. Using temporal logic, the *execution* of a program can be formalized, and not just the function or relation it computes. In a quantified temporal logic timing properties are expressed explicitly. The quantified temporal logic of Harter and Bernstein [25] allows one to express the relative progress of two programs with respect to each other, as well as the execution time of a program fragment. Quantification of time is introduced by providing a real-time component to the *eventuality operator*. The authors demonstrate their method by proving safety properties of real-time programs, written in a Modula-style language.

The quantification of temporal logic provides a compact means of expressing temporal relationships between program states, and allows reasoning about the execution of real-time programs. The *model* of execution, however, is unrealistic for true real-time programming, in that it employs a pure interleaving approach. Thus, true concurrent program execution cannot be modeled. Because of this, the execution of *all* possible operations in the "ready" state is not forced; instead, the execution of only one is ensured. We believe this defect mandates an assumption of fairness, i.e., scheduling. Without such an assumption, it is possible to specify a process whose timing constraints are consistent when its operations are executed in only one order. Deadlock may intervene when execution proceeds in an "unfair" order.

Net Models Extensive work has been performed using Petri Nets to model concurrent and real-time systems. There are two approaches to augmenting nets with time. Either a time is assigned to each transition in the net [5, 6] or a time is assigned to each place [7]. Petri nets have several advantages as a model. They are simple, elegant and posses an extensive theory that has already been developed. They can directly represent the

causality and dependence between events. Their graph structure implies that they can be used to represent the system at various levels of abstraction. Furthermore, the token labeling makes them particularly useful for performance analysis. However, they are not readily decomposable, and there is no simple notion net "containment" other than other than graph equivalence. As we have stated above, without a containment relation, a specification model does not provide a realistic foundation for verification. It is obvious that the equivalence relation is much too strong.

SCCS: SCCS [8] is the most general of all the real-time models. With only four operators and a recursive construct, all operators found in the other models of real-time concurrency can be expressed. SCCS implicitly models time by associating the occurrence of an event with the passage of a single time unit. Thus, an executing process must engage in an event at each instant in time. This requirement makes it cumbersome, but necessary, to define an idling event in order to represent process delay.

The parallel operator in SCCS is the most general found in any of the models of time dependence. Concurrency is represented by processes executing in a mutually lock-step fashion. This method allows the simultaneous occurrence of events, but does not provide any inherent notion of shared events or synchronization. Interaction among processes is captured by using parallel composition and event restriction. However, this representation is quite awkward, as the restriction operator must allow all *correct* actions. Thus, the user must know *a priori* all allowable actions and every combination of actions.

An SCCS process is analyzed by proving that it has some desired property. This is done by transforming the property into an SCCS process, and showing the two processes to be equivalent. One must derive a bisimulation between them, or manipulate their terms using a set of process identities. Like many other models, SCCS does not have an ordering relation on processes. Thus, to show that an implementation satisfies a specification, the two must be equivalent.

CIRCAL: CIRCAL [9] was developed to provide a useful and powerful calculus to describe and analyze communicating systems. Concurrency is modeled by event-driven synchronization among multiple processes. That is, if an event occurs in several processes simultaneously, the processes interact. Using this method, CIRCAL can be used to model time dependent processes by synchronizing event occurrences with ticks of a clock process. This yields an event-based representation of time, which differs from SCCS in the following respect: Only those events synchronized with a clock have a time associated with them. Therefore, the occurrence time of a nonsynchronized event cannot be derived. Like SCCS, CIRCAL provides a set of rules to show process equivalence. Also like SCCS, there is no containment ordering on processes – the essential ingredient in a verification system.

Timed Stability and Timed Failures: The Timed Stability [10] and Timed Failures models [11] are both extensions of CSP. *Timed Stability* is a temporal extension of the trace model for CSP [26] and is used to represent deterministic, real-time communicating processes. A timed CSP process is defined as a set of ordered pairs (s, α) where s is a *timed trace*[2] and α is the *stability time* associated with s. The stability time defines the earliest time after which a process is able to engage in any possible next event after executing s. The Timed Stability model is consistent with the algebraic laws presented in [12], and these laws can be used to show process equivalence and process containment. There is also a satisfaction relation defined in Timed Stability which can be used to relate a process to its logical specification.

The timed failures model of Gerth and Boucher is an extension of the failure set model of CSP [12]. A timed failure consists of an *event relation* and a *failure relation.* The event relation is similar to a trace, which records all events that occur during a program's execution. A failure relation contains all of the events that can be refused throughout a computation, and a pair $(\surd, i)$, representing termination of the machine at time i. This is the minimum amount of failure information that is needed to model both timeouts and the simultaneous occurrence of actions.

CSP-R and DNP-R: De Roever's group at Eindhoven has proposed a model of real-time computing that extends the linear history semantics for CSP of Francez, Lehman and Pnueli [27]; it is used to define the semantics of Ada's essential features. Their execution model is a variant of the maximal parallelism model of Salwicki and Müldner [28], where events are forced to occur at the earliest time possible. As we have noted above, this notion is a quite unrealistic, as it assumes a one-to-one relationship between processes and processors.

A process P is represented by its possible "executions," where each execution consists of a state and a history. The state records the values of the variables if P terminates, and is undefined if P does not. The history is a (possibly infinite) sequence of time records which notes all communications leading to the state. The i^{th} record corresponds to time i, and the length of the history is equal to the time of termination.

The first version of this model was developed by Koymans *et al.* [13] and was used to capture the semantics of Ada constructs in a language called CSP-R. However, it was shown [14] that in this model two fragments with identical semantics may admit different observable behaviors. Huizing, Gerth and de Roever extended the model to correct this defect, and then used it to supply denotational semantics for Occam in a language called DNP-R.

[2] A timed trace is a sequence of event-time pairs representing the events a process executes and their occcurrence times.

3 Our Approach

In this section we define our specification and implementation models. We argue that both are necessary when one reasons about the temporal properties of a real-time system. Moreover, if the reasoning is to be mathematically sound, both models must be based on formal semantics, and there must be precise relationships between the semantics and their meanings. In the traditional specification/verification paradigm [29], the specification "abstracts out" most properties of program execution, and thus is represented at a much higher level than its underlying system. This makes it possible to define a specification that is independent of any machine-specific details. In this manner, the specification of a real-time system describes only the high-level timing requirements of processes, and the timing dependencies between them. Conversely, the implementation model captures *all* aspects that affect a system's timing behavior during execution. At neither level, however, do we model non time-related properties such as variable state.

Our goal is to develop both specification and implementation models, and to define a well-founded mapping between them. Each model consists of a language and a semantic domain, such that representations written in the language denote objects in the domain. Within the specification language, a given specification represents a set of acceptable temporal behaviors. This model is based on the *Timed Acceptance* paradigm that we have developed [21].

The implementation model is used to accurately capture the execution behaviors of real-time programs. First, we distinguish between two different types of time: execution time and wait time. We also preserve the difference between true device concurrency, and synchronized process interleaving. With these tools, we enable the specification of scheduling disciplines. Furthermore, our compositional semantics for concurrency include assumptions on clock discrepancy.

Instead of using an existing programming language as our foundation, we have decided to develop an independent implementation model. There are several reasons for pursuing this strategy. First, there is no real-time programming language that possesses both well-defined semantics and also explicitly captures timing properties. Second, our implementation model is general enough to formally define the semantics of various recent real-time programming languages [30, 31, 22, 32]. Finally, by focusing strictly on timing behavior, we are free to concentrate expressly on the temporal issues of real-time programs.

The remainder of this section is organized as follows: We first describe the *Timed Acceptances* model, and note the additional work required to enhance its specification abilities. Following this, we discuss our implementation model, which enables the representation of real-time programs within their execution environments. The section concludes with our proposed efforts directed toward relating the two models.

3.1 The Timed Acceptances Model

The *Timed Acceptances* model [21] is used to specify the abstract temporal behavior of concurrent processes. In this model, a time dependent process is represented by its possible *executions*. Each *execution* includes both the externally visible events and the internal choices the process makes while executing. The external behavior is maintained by a *timed trace*, while the internal choices are represented by a state set. This notation of nondeterminism is similar to Hennessy's Acceptance Tree model of nondeterministic processes [33]. The difference lies in the fact that we represent the *time* at which events can occur within a process. Furthermore, by including nondeterministic choice we have provided a valid structure for real-time specification. In this manner, we can abstractly represent *if-then-else* constructs based on a process' internal state. Perhaps the most important feature of the model is that it *explicitly* represents temporal behavior by associating an occurrence time with each event in a process' execution.

An *event* is an instantaneous visible action in which a process engages. We have decided to make events instantaneous since an action with a duration can be modeled using two events: one for the beginning of the action and another for the end. Time is modeled by the set, $\boldsymbol{N}^\infty$, containing the natural numbers $\boldsymbol{N}$ and ∞. An execution sequence is called an *acceptance*, and consists of a timed trace and a state set, $(s, \bar{\sigma})$. A *timed trace* is a finite sequence $\langle (a_1, n_1), (a_2, n_2), \ldots, (a_m, n_m) \rangle \in (\Sigma \times \boldsymbol{N}^\infty)^*$, which records the events that a process has executed up to some moment in time. Each (a_i, n_i) pair represents the occurrence of the i^{th} event in the execution sequence. The time n_1 is the time between 0 and the occurrence of a_1. For $i > 1$, n_i represents the relative time between events a_{i-1} and a_i.

A *state set* is a saturated set denoting the possible states a process may enter after executing the trace s. A *state* defines a set of deterministic choices a process may make when deciding its next action. The next action is either the execution of an event at a particular time or a *stop* condition. A state is denoted by a set of event-time pairs $\{(A_1, n_1), \ldots, (A_m, n_m)\} \subseteq \mathcal{P}(\mathcal{P}(\Sigma) \times \boldsymbol{N}^\infty)$, where $\mathcal{P}(A)$ is the powerset of A, and A_i is either $\{a_i\}$ or $\emptyset$. Each $(\{a_i\}, n_i)$ represents the possibility of event a_i occurring at n_i time units after a process enters the state. The pair $(\emptyset, n_i)$ represents the possibility of the process stopping at time n_i. Thus, unlike the failures model, in which a timed failure is considerably more complex than its untimed counterpart, the state in the timed acceptances model is of about the same complexity as the untimed state.

3.1.1 Domain of Processes

Mathematically, a real-time process P is a pair $(\bar{\alpha}P, \mathcal{A}(P))$ where $\bar{\alpha}P$ is the *alphabet* of P and $\mathcal{A}(P)$ is its *acceptance set*. The alphabet is the set of events P can execute. The acceptance set represents all possible executions of P. For example,

$$P = (\{a, b\}, \{\ (\langle\rangle, \{\{(a,3)\}, \{(b,5)\}, \{(a,3),(b,5)\}\}),\\ (\langle (a,3)\rangle, \{\{(\emptyset,1)\}\}), (\langle (b,5)\rangle, \{\{(\emptyset,2)\}\})\})$$

is a process that makes a nondeterministic choice before engaging in any event. This is reflected by the acceptance $(\langle\rangle, \{\{(a,3)\}, \{(b,5)\}, \{(a,3),(b,5)\}\})$. If it chooses to be in the state $\{(a,3)\}$, then it can only engage in a at time 3.

To guarantee that the alphabet-acceptance set pair corresponds to an intuitive notion of process execution, we require that the process only engage in events within its alphabet. Also, its trace set is prefix closed, and the state sets are all saturated. Furthermore, a trace is extensible by all events in all states associated with it.

The set of time dependent processes is partially ordered by *process containment*. *Process containment*, denoted $\sqsubseteq$, is a measure of the *amount* of nondeterminism displayed by one process relative to another. We say that P is more nondeterministic than Q if P can make at least as many nondeterministic decisions as Q after both execute the trace s. We say that P is contained in Q, $P \sqsubseteq Q$, if $\bar{\alpha}P = \bar{\alpha}Q$ and $\mathcal{A}(Q) \subseteq \mathcal{A}(P)$. Process containment is a natural relation to use when comparing a process to its specification [34], because a specification can be expressed as a process P_S. We then say that P satisfies its specification P_S if every behavior of P is also a behavior of P_S. In other words, $P_S \sqsubseteq P$. Process containment also induces an equivalence relation on the domain. This equivalence relation corresponds to the intuitive definition of process equivalence in that two processes are equivalent if they exhibit the same visible behavior under all circumstances, and can make the same nondeterministic decisions at all stages of their computation. That is, they have identical alphabets and acceptances.

3.1.2 Primitive Operators

Here we define a set of operators on the domain. They enjoy a variety of algebraic properties such as monotonicity with respect to process containment and continuity. Furthermore, we believe they fully capture the temporal properties of all known real-time languages.

Timed action represents the sequential execution of events with respect to time. The process $a \overset{i}{\rightsquigarrow} P$ engages in event a and, after delaying for exactly i time units, behaves like the process P. This operator represents the occurrence time of events, the simultaneous occurrence of events and the initial delay in a process' execution.

$P \| Q$ denotes the simultaneous execution of P and Q. Simultaneous execution is represented by interleaving the executions of P and Q in a way that preserves the occurrence times of the events in the two processes. P and Q interact only if they are able to engage in the same event simultaneously. Thus, $P \| Q$ contains all traces that belong to P and Q when they are restricted to the alphabet of P and Q, respectively. The parallel operator is associative and permits the specification of n-way synchronization. The choice construct, $P \,\Box\, Q$, represents the deterministic choice between two processes. Specifically, the choice made on the occurrence time of events. $P \sqcap Q$ represents the nondeterministic choice between two processes. In the above example, the process P can be represented as $(\epsilon \overset{3}{\rightsquigarrow} a \overset{1}{\rightsquigarrow} \text{STOP}) \sqcap (\epsilon \overset{5}{\rightsquigarrow} \text{b} \overset{2}{\rightsquigarrow} \text{STOP})$. On the other hand, $(\epsilon \overset{3}{\rightsquigarrow} a \overset{1}{\rightsquigarrow} \text{STOP}) \,\Box\, (\epsilon \overset{5}{\rightsquigarrow} \text{b} \overset{2}{\rightsquigarrow} \text{STOP})$ has the initial acceptance $(\langle\rangle, \{\{(a,3),(b,5)\}\})$.

Concealment and renaming are abstraction mechanisms. Concealment is used to isolate the relevant events from the surrounding details of a process' execution. $P\backslash A$ engages in all the behaviors of P, but only those events in $\bar{\alpha}P - A$ are in its acceptances. Concealment preserves the occurrence times in P of the visible events. It can also introduce nondeterminism into the execution. Renaming is an abstraction mechanism that is used to define sets of processes with similar behavior. If $f : \Sigma \rightarrow \Sigma$ is a function that renames events, then the process $f[P]$ engages in the event $f(a)$ whenever P would have engaged in a. If f is not one-to-one, then renaming by f may introduce nondeterminism since it may identify two or more events which were distinguishable in P. Repetition is modeled using the recursive construct $\mu P.F(P)$ where F is a function composed of the above operators and P is a process identifier. The semantics of $\mu P.F(P)$ are defined as the least fixed point of the chain of processes approximating $\mu P.F(P)$. The least fixed point exists and is unique since all the other operators are continuous.

3.1.3 Derived Operators

Using the primitive operators, we derive constructs representing interval timing bounds, timeouts and periodic processes. There are two constructs for denoting interval timing bounds. Deterministic timed action on intervals, $a \overset{int}{\rightsquigarrow} P$, is derived from timed action and choice. Thus $a \overset{int}{\rightsquigarrow} P$ is equivalent to the process that executes a at time 0 and then deterministically chooses to start executing P at some time $i \in int$. For example,

$$\epsilon \overset{[3,5]}{\rightsquigarrow} a \overset{1}{\rightsquigarrow} P = (\epsilon \overset{3}{\rightsquigarrow} a \overset{1}{\rightsquigarrow} P') \Box (\epsilon \overset{4}{\rightsquigarrow} a \overset{1}{\rightsquigarrow} P') \Box (\epsilon \overset{5}{\rightsquigarrow} a \overset{1}{\rightsquigarrow} P')$$

Nondeterministic timed action on intervals, $a \overset{int}{\approx\!\!\rightsquigarrow} P$, is derived from timed action and nondeterministic composition. Thus $a \overset{int}{\approx\!\!\rightsquigarrow} P$ is equivalent to the nondeterministic choice among the processes $a \overset{i}{\rightsquigarrow} P'$ for all $i \in int$. For example,

$$a \overset{[3,5]}{\approx\!\!\rightsquigarrow} P' = (a \overset{3}{\rightsquigarrow} P') \sqcap (a \overset{4}{\rightsquigarrow} P') \sqcap (a \overset{5}{\rightsquigarrow} P')$$

A process is subject to a *timeout* if it must execute some event a by a time n. If a is not executed by this time, then either the process fails, or another process is invoked to handle the timeout. A process in which a must occur by time n, and otherwise Q is executed, is represented by $P \overset{a,n}{\mapsto} Q$. Note that this mechanism allows us to specify real-time exception handlers. Finally, a periodic process is a cyclic process that is activated every n time units. We represent a periodic process by $\pi P, n.F(P)$, where P is a process variable and $F(P)$ is the process that must begin execution every n time units.

3.1.4 Examples of a Time Dependent Process

We now describe a timed Producer-Consumer problem using the timed acceptances model. We then use the properties of the algebra to analyze its execution behavior. In

doing so, we find that the timing constraints on the problem lead to deadlock, and are therefore inconsistent.

The system is composed of three processes; a consumer and two producers, and is denoted by the term $C \| P_1 \| P_2$. The system operates correctly if the Consumer processes the data from both producers and none of the component processes stop. The Consumer deterministically chooses between processing data from Producer_1 or Producer_2. If it chooses to process data from Producer_1 (modeled by the event a) it operates on this data for two seconds and then waits for another data item from either Producer_1 or Producer_2. If it accepts the data from Producer_2 (executes b), it operates on the data for four seconds and then waits for another data item from either process. After processing the data from a producer, it must immediately receive another piece of data. The Consumer is represented by the following expression:

$$C = (\epsilon \overset{[0,2]}{\rightsquigarrow} a \overset{2}{\rightsquigarrow} C) \,\square\, (\epsilon \overset{[0,4]}{\rightsquigarrow} b \overset{4}{\rightsquigarrow} C)$$

Producer_1 generates a data item in one second and then can wait up to two seconds for this data to be accepted. Producer_2 also needs one second to produce a data item, but will only wait one additional second for this item to be accepted. These two processes are represented by the expressions:

$$P_1 = \epsilon \overset{[1,3]}{\rightsquigarrow} a \overset{0}{\rightsquigarrow} P_1$$
$$P_2 = \epsilon \overset{[1,2]}{\rightsquigarrow} b \overset{0}{\rightsquigarrow} P_2$$

Using the properties of parallel composition and choice, we eliminate the parallel operator, obtaining a representation of $C \| P_1 \| P_2$ that explicitly demonstrates the interactions between the three processes, and the effect of these interactions on their subsequent behavior. Examining the simplified expression, we see that two of the executions lead to deadlock. The simplified process is as follows:

$$C \| P_1 \| P_2 = (\epsilon \overset{i \in [1,2]}{\rightsquigarrow} a \overset{2-i}{\rightsquigarrow} \text{STOP}_{\{a,b\}}) \,\square\, (\epsilon \overset{i \in [1,2]}{\rightsquigarrow} \text{b} \overset{3-i}{\rightsquigarrow} \text{STOP}_{\{a,b\}})$$

We see from the first subterm in the above representation of $C \| P_1 \| P_2$ that if C initially chooses to accept a from P_1, the system deadlocks at time 2. Here, the Consumer takes too long to process the data from Producer_1 and as a result, Producer_2 deadlocks. Similarly, if C chooses b from P_2, then P_1 stops at time 3.

Having determined that the system does not execute correctly, and isolated the cause of the problem, we modify the Consumer so that the timing constraints are consistent. The new Consumer C' strictly alternates accepting data from the producers and processes each piece of data in 1 time unit. It is represented by the expression

$$C' = \epsilon \overset{1}{\rightsquigarrow} a \overset{1}{\rightsquigarrow} b \overset{0}{\rightsquigarrow} C'$$

All acceptances of $C' \| P_1 \| P_2$ consist of a trace that begins with the event a and then strictly alternates between the b and a with one second between consecutive events.

The state set consists of a single state. If the last event in the trace is an a, then the state is $\{(b,1),(\eta,1)\}$. If the last event is a b, then the state is $\{(a,1),(\eta,1)\}$. The pair $(\eta,1)$ reflects the potential deadlock time and is necessary to ensure the associativity of parallel composition. It represents the time at which $C'\|P_1\|P_2$ stops if, when composed with another process that also executes a and b, but a and b are not offered.

3.1.5 Enhancements

While the *Timed Acceptances* model is a logically sound and complete axiom system, we feel that several enhancements are needed to make it a useful specification model. These enhancements stem mainly from its extremely formal nature, and are designed to make it more "user-friendly". Note that similar extensions have been proposed for RTL [1, 24]. In this section we mention several of these proposed additions.

Mnemonic Language: As it stands, the operators provide a solid foundation for any real-time specification language. However, we propose mapping them to mnemonic constructs that more closely resemble those of a programming language. For example, the process P_1 shown above could have the following mnemonic specification:

while *true* **do**
after 0 *and* **before** 4
event a

With constructs such as these, a system designer need not be aware of the algebra underlying a specification.

Modularity: A natural extension to the specification language is enhanced modularity. First, this permits a top-down approach to specification, which is a desirable asset. Moreover, it provides the ability to specify *nested* timing constraints. For example, let P_1 and P_2 be two processes with individual timing constraints:

$$P_1 = \epsilon \overset{[1,4]}{\rightsquigarrow} a$$
$$P_2 = \epsilon \overset{[1,2]}{\rightsquigarrow} b$$

Assume we consider P_2 to be a *subroutine* of P_1, which is called during the *transition* $\epsilon \overset{[1,4]}{\rightsquigarrow} a$. The interpretation is as follows: Before executing a, P_1 calls P_2. P_2 must satisfy its own *local* timing constraints, i.e. it must engage in b at either 1 or 2 time units from its entry time.

Currently we can specify such behaviors in a bottom-up fashion, and then use the concealment operator to capture the essential timing constraints of P_1. To implement a top-down approach, we must associate a process with each transition. This new derived operator, yet to be defined, implicitly incorporates P_2's alphabet and constraints into those of P_1.

Automated Analysis: There are three approaches to verification. One is to use the properties of the algebra to demonstrate the correctness of a program. The second is to use it as the model of a logic, and to perform the proofs in the logical system. The third approach is a probabilistic expansion of the execution sequences.

The first approach is a syntactic one. Processes and specifications are represented as terms formed by composing the operators. The process satisfies the specification if all of its behaviors, when restricted to the specified events, are behaviors of the specification. Thus verification is a matter of showing that for a process P and a specification P_S, $P_S \sqsubseteq P$. That is, the specification is less deterministic than the process. For finite processes this can be done by manipulating the two terms, using the properties of the operators and the partial ordering. For infinite processes, the proofs are performed inductively, by considering the finite approximations [35] of the processes. A process satisfies its specification if the process is contained in every finite approximation of the specification.

Our experience shows that these proofs are quite long, but the work involved is mechanical. Therefore, we wish to automate as much of the verification as possible. We plan to investigate the properties of the algebra to determine whether they form a rewrite system [36]. If they do, mechanizing this axiom system will be straightforward. If not, further work must be performed to find approaches for mechanizing the proof system. We also need to investigate methods of automating the process of reasoning about infinite processes. One possible approach is to incorporate some of the techniques used in tactical theorem provers [37, 38] that allow user interface during the proof process.

The second approach to verification is to use the algebra as a model for a logical system (either a first order logic or a quantified temporal logic) and define predicates on processes. This is the approach used by Hoare for proving properties of CSP processes [26]. We also plan to study how we can use the algebra as a model for RTL and prove safety assertions expressed in it.

The third approach is a probabilistic one. The basic idea is to assign probabilities with choices and to explore the execution sequences with probabilities higher than some predetermined threshold [39]. In this way, the timed trace analyzer need not explore all possible execution sequences. Although this method has been developed for processes with finite behaviors, we need to investigate further to see whether it can be applied effectively to our specification model.

3.2 An Implementation Model

Unlike the specification model, an implementation model should describe all essential aspects of real-time computation. First, it should be possible to distinguish execution time and wait time since their effect on the timing behavior of other processes are logically different. Second, it should be possible to represent processes executing concurrently on a single processor. Most semantics for parallel composition do not support this notion and assume an ideal one-to-one correspondence between processes and pro-

cessors. Third, it should be possible to capture scheduling and clock synchronization assumptions. Since specification models ignore scheduling paradigms, the derived set of potential operational behaviors is artificially inflated. Thus, such specifications may include deadlocked behaviors, when a simple earliest-deadline scheduler would preclude them.

The properties necessary for the implementation model subsumes those for the specification model. This section only describes additional properities needed to represent real-time program execution. Since we have to show that the behavior of an implementation is contained in the behaviors of a specification, it should be natural to transform an implementation behavior to a specification behavior. The relation between the two models is discussed in the next section.

3.2.1 Execution Model

Unlike the specification model in which time is treated uniformly regardless of its purpose, we distinguish the time a process spends executing from the time it spends waiting. Consider the following two processes executing on the same processor:

$$A = a \overset{3}{\rightsquigarrow\!\!>} b \overset{0}{\rightsquigarrow\!\!>} A_1$$

$$B = a \overset{3}{\rightsquigarrow\!\!>} c \overset{0}{\rightsquigarrow\!\!>} B_1$$

If the execution time of A between events a and b is 2, and the execution time of B between events a and c is 2, it is not possible for both A and B to satisfy their local timing constraints. However, if their execution times are 1 and 2, respectively, then both local timing constraints can be satisfied, assuming that there are no other processes running on the same processor. Thus, in order to see whether the local timing constraints can be met, the implementation model must support the notion of execution time. That is, the timed trace must include execution time information, and the timed action operator now needs to denote whether the process is executing an operation, or simply waiting.

There are several ways to extend a timed trace to include execution. For example, additional events can be used o denote the beginning and end of execution, the execution time between two events can be included. As part of the proposed research, we will develop the most appropriate way to extend timed traces to include execution time information. Based on our preliminary study, we plan to operationally represent the passage of time using a *(execution-time; next-event-time)* pair, which means that a process executes for *execution-time* during [0, *next event time*]. That is, the process $a \xrightarrow{(i;j)} P$ engages in event a, executes i time units by time j, and behaves like the process P at time j. The above two processes A and B can then be represented as follows:

$$A = a \xrightarrow{(2;3)} b \xrightarrow{(0;0)} A_1$$

$$B = a \xrightarrow{(2;3)} c \xrightarrow{(0;0)} B_2$$

We also use execution time is eliminate impossible choices between interacting processes. For example, consider

$$C = \epsilon \xrightarrow{(1;2)} (b \xrightarrow{(0;0)} C_1 \,\square\, \epsilon \xrightarrow{(0;3)} c \xrightarrow{(0;0)} C_2 \,\square\, \epsilon \xrightarrow{(0;4)} d \xrightarrow{(0;0)} C_3)$$

$$D = \epsilon \xrightarrow{(2;2)} (b \xrightarrow{(0;0)} D_1 \,\square\, \epsilon \xrightarrow{(0;3)} c \xrightarrow{(0;0)} D_2 \,\square\, \epsilon \xrightarrow{(0;4)} d \xrightarrow{(0;0)} D_3)$$

If C and D are to run on the same processor, they cannot perform event b at time 2, since it is impossible for both C and D to execute 1 and 2 time units within 2 time units. Whether they execute event c at time 5, or event d at time 4, depends on the number of processes running together on the same processor and the semantics of the parallel composition of processes running on the same processor. These issues are discussed in the next section.

Since the execution time of a process cannot be estimated exactly [40, 41], it is important to be able to express an execution time using an interval. The lower and upper bounds on an execution time interval represent the minimum and maximum possible execution times. Since it will be difficult to predict the actual execution time, it is natural to model the execution time nondeterministically. That is,

$$\epsilon \xrightarrow{([i,j];k)} P = \sqcap_{l \in [i,j]} \epsilon \xrightarrow{(l;k)} P$$

Another basic temporal behavior of a time dependent process is waiting for an event until a timeout or deadline. Such a timing behavior can be naturally expressed using a wait time interval. Since the duration of process delay depends on other processes, its choice should be made deterministically. That is,

$$\epsilon \xrightarrow{(i;[j,k])} P = \square_{l \in [j,k]} \epsilon \xrightarrow{(i;l)} P$$

3.2.2 Parallel Operators

At the implementation level, process allocation and process scheduling are important in determining the temporal behavior of a real-time system. Process allocation refers to the assignment of processes to processors. Since two processes on the same processor cannot execute at the same time, it is necessary to distinguish whether two processes are assigned on the same processor. For example, let us reconsider the above two processes A and B. If A and B are on different processors, then they can execute a and c at time 0 and b at time 3. If they are assigned to the same processor, then they cannot execute a and c at time 0, nor can they both be ready to execute b at time 3. This is modeled as deadlock at time 3.

To distinguish whether processes are running on the same processor, we provide two parallel operators: the *local parallel* operator (&) for composing processes running on the same processor, and the *global parallel* operator (||) for composing processes running on different processors. The difference is that the global parallel operator allows the execution times of component processes to overlap, whereas the local parallel operator

only allows interleaved execution of processes. The local parallel operator differs from a traditional interleaving operator, since shared events must happen at the same time when the former is used to compose processes.

To illustrate the importance of local and global parallel operators, consider the producer and consumer problem described in the previous section. Suppose that they are implemented as follows:

$$C' = \epsilon \xrightarrow{(1;1)} a \xrightarrow{(1;1)} b \xrightarrow{(0;0)} C'$$
$$P_1 = \epsilon \xrightarrow{(1;[1,3])} a \xrightarrow{(0;0)} P_1$$
$$P_2 = \epsilon \xrightarrow{(0;[1,2])} b \xrightarrow{(0;0)} P_2$$

C' executes one time unit to consume each data. P_1 executes one time unit to produce the next data, whereas P_2 is an external device that produces data and thus it does not use any execution time. If the current implementation is $(C'\&P_1)\|P_2$ or $C'\&(P_1\&P_2)$, then they become deadlocked at time 1. However, for $(C'\&P_2)\|P_1$, $C'\|(P_1\&P_2)$ or $(C'\|P_1\|P_2)$, it can be shown that these implementations satisfy the specification.

Process scheduling determines which process to execute next. Obviously, scheduling affects the overall timing behaviors of a real-time system. It is important to be able to represent various scheduling disciplines, such as earliest deadline first and highest priority process first, in the implementation model. The representation of scheduling can be defined implicitly or explicitly. The implicit scheduling approach is to include the scheduling assumption in the definition of a parallel operator. For example, the semantics of a parallel operator can be defined to obey "maximum parallelism" [28]; that is, a process event is never unnecessarily delayed in a "ready" state. This ensures that an event happens at the earliest possible time. However, maximum parallelism is meaningful only if each process is assigned to a dedicated processor. Thus, it is appropriate for the global parallel operator only. For the local parallel operator, the next event can be selected based on highest priority process first, earliest next event time first or smallest laxity first, where laxity equals *next-event-time − execution-time.* We plan to define several local parallel operators with these meanings and a hierarchy among them.

The explicit scheduling approach is to use a scheduler process that enforces a desired discipline on process interaction. In general, the purpose of scheduling is to limit the possible behaviors of the processes to only those with desired properties. Similarly, the parallel composition of processes restricts the behaviors of a component process to those that are mutually consistent with the behaviors of the other processes. Thus, for a given set of processes and a scheduling discipline used in the implementation, it is possible to define a scheduler process that enforces the discipline. As we have shown in [42], formulating an explicit schedule is at times possible. However, it typically results in a very complex process; at the worst case, a scheduler process might have to explicitly enumerate all valid execution sequences. We plan to investigate whether there are some high-level operators, such as filtering undesirable execution sequences, that can be used define a scheduler process succinctly. Once we better understand the implicit and explicit

scheduling issues, we also plan to investigate their relationship.

3.2.3 Clock Synchronization

All formal models, except DRTL [3], assume that there is a global clock. This, however, is an unrealistic assumption for the implementation model since local clocks are not perfectly accurate. We have studied three different ways to implement deadlines associated with communication primitives [43, 44]. They assume that the maximum drift between any two clocks can be bounded, the difference between any two clock rates can be bounded and the maximum message delay between any two processors can be bounded, respectively. Since time based on one clock cannot be converted exactly to time based on another clock, it seems natural to represent a time value depending on a remote clock as an interval, whose exact value is resolved nondeterministically. However, the explicit representation of clock discrepancy results in convoluted timing information, which makes the analysis difficult. We plan to study the exact effect of clock synchronization on the timing behavior of an implementation, and develop a clean way to represent clock discrepancy in the implementation model. Our current attempt is to capture the clock discrepancy in the semantics of global parallel operator. In particular, we are investigating whether it is possible to define a parameterized global parallel operator with respect to clock difference.

3.2.4 Verification

The verification of an implementation consists in showing that the timing constraints are consistent, and that the temporal behaviors of the implementation satisfies the specification. To prove the correctness of an implementation, the language used to represent the implementation must be well-founded. For this, we plan to develop appropriate semantics for the implementation model. The consistency of the timing constraints can then be shown by proving that the execution times of each process are guaranteed, and that processes do not time deadlock. Techniques for proving them should be similar to those described for the specification model. To show the latter, we need to know the exact relation between the specification model and the implementation model.

3.3 Relations Between the Two Models

There are two ways to relate the specification model and the implementation model. The first approach is to use the same semantic domain (of processes) for both the specification model and the implementation model. Then, to show that an implementation satisfies a specification, we use the process containment relation after implementation-dependent events are hidden. Another approach is to define a different semantic domain for the implementation model.

Let Imp and $Spec$ denote the semantic domains of the implementation model and the specification model, respectively. Since the implementation model allows a finer

distinction on process behaviors than the specification model, it should be possible to find an embedding-projection (e,p) pair of functions with $e : Spec \rightarrow Imp$ and $p : Imp \rightarrow Spec$. This pair should relate two semantic domains naturally in the following sense [45]:

$$p \circ e = ID_{spec} \quad \text{and} \quad e \circ p \sqsubseteq ID_{imp},$$

where ID_{spec} and ID_{imp} denote the identify function on their respective domain. The latter containment relation states that, for every $x \in Imp$, $e \circ p(x)$ should approximate x.

In addition to relating their semantic domains, we also plan to investigate syntactic transformations between the implementation language and the specification language. We believe that it should be possible to translate a representation in the implementation model to a corresponding representation in the specification model, and *vice versa*, through purely syntactic manipulations. For example, a $(execution{-}time; next{-}event{-}time)$ pair can be syntactically translated to an interval in the specification model as follows:

$$a \xrightarrow{(3;5)} \text{STOP} = \text{a} \overset{5}{\rightsquigarrow} \text{STOP}$$

$$a \xrightarrow{([1,2];3)} \text{STOP} = \text{a} \overset{3}{\rightsquigarrow\!\!\!\!\gg} \text{STOP}$$

$$a \xrightarrow{(1;[2,3])} \text{STOP} = \text{a} \overset{[2,3]}{\rightsquigarrow} \text{STOP}$$

$$a \xrightarrow{([1,2];[3,4])} \text{STOP} = \text{a} \overset{[3,4]}{\rightsquigarrow} \text{STOP}$$

The embedding-projection pair should also induce syntactic translations of other operators.

4 Summary and Research Plan

The goal of the proposed research is to provide a formal framework for reasoning about the temporal properties of real-time systems. We propose to develop two models, one for specification and another for implementation, and to relate them mathematically. More specifically, we plan to do the following:

1. Refine our specification model (the timed acceptances model) and develop a high-level specification language that is natural to use and that supports modular specification.

2. Investigate various ways to automate analysis. Here, we plan to look at rewrite systems, tactical theorem provers and probabilistic verification methods.

3. Develop the syntax of operators for the implementation model so that the execution behavior and operational environment of real-time programs can be represented explicitly. Also, we plan to develop the semantics of the implementation model and the relations between the specification model and implementation model.

4. Evaluate our approach by modeling and analyzing the timing properties of a distributed robot system being developed at Penn [46].

These issues will be addressed over the next three years as follows:

Year 1: Refine specification model and language; begin to develop implementation model.

Year 2: Develop semantics of implementation model; relate specification and implementation models. Start looking at tools for automated analysis.

Year 3: Finalize the models, and evaluate by modeling and analyzing distributed robot systems. Complete tools for automate analysis.

Acknowledgements

We wish to thank A. Zwarico for her assistance in evaluating related work and for her valuable suggestions throughout the preparation of this paper.

References

[1] F. Jahanian and A. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 890–904, September 1986.

[2] F. Jahanian and A. Mok, "A Graph-Theoretic Approach for Timing Analysis and its Implementation," *IEEE Transactions on Computers*, vol. C-36, pp. 961–975, August 1987.

[3] G. MacEwen and T. Montgomery, "The RNet Programming System: Distributed Real-Time Logic," Tech. Rep. Report 87-3, Dept. of Computing and Information Science, Queen's University, Kingston, Ontario, November 1987.

[4] N. Rescher and A. Urquhart, *Temporal Logic.* Springer-Verlag, 1971.

[5] C. Ramchandani, "Analysis of Asynchronous Concurrent Systems by Petri Nets," Tech. Rep. Project MAC, TR-120, M.I.T., Cambridge, MA, 1974.

[6] C. Ramamoorthy and G. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Trans. on Software Eng.*, vol. SE-6, pp. 440–449, September 1980.

[7] J. C. Jr. and N. Roussopoulos, "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 603–616, September 1983.

[8] R. Milner, "Calculi for synchrony and asynchrony," *Theoretical Computer Science*, vol. 25, pp. 267–310, 1983.

[9] G. Milne, "CIRCAL and the Representation of Communication, Concurrency, and Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 270–298, April 1985.

[10] G. Reed and A. Roscoe, "A Timed Model for Communicating Sequential Processes," in *Proceedings of ICALP '86, LNCS 226*, 1986.

[11] R. Gerth and A. Boucher, "A Timed Failure Semantics for Extended Communicating Processes," Tech. Rep. TR. 4-4(1), Department of Mathematics and Computing Science, Eindhoven University of Technology, March 1987.

[12] S. Brookes, "A Model for Communicating Sequential Processes," Tech. Rep. CMU-CS-83-149, Department of Computer Science, Carnegie–Mellon University, 1983.

[13] R. Koymans, R. Shyamasundar, W. de Roever, R. Gerth, and S. Arun-Kumar, "Compositional Semantics for Real-Time Distributed Computing," in *Logic of Programs Workshop '85, LNCS 193*, 1985.

[14] R. G. C. Huizing and W. de Roever, "Full Abstraction of a Denotational Semantics for Real-time Concurrency," in *Proc.* 14^{th} *ACM Symposium on Principles of Programming Languages*, pp. 223–237, 1987.

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

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

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

[18] E. Klingerman and A. Stoyenko, "Real-time euclid: a language for reliable real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-12, Sep. 1986.

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

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

[21] A. Zwarico, *Timed Acceptance: An Algebra of Time Dependent Computing.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1988.

[22] I. Lee, S. Davidson, and V. Wolfe, "Motivating time as a first class entity," Tech. Rep. MS-CIS-87-54, Dept. of Computer and Information Science, University of Pennsylvania, July 1987.

[23] W. de Roever, "Questions to Robin Milner – A Responder's Commentary," *Information Processing 86*, pp. 515–518, 1986.

[24] G. MacEwen and T. Montgomery, "The RNet Programming System: Requirements Language Definition," Tech. Rep. Report 87-1, Dept. of Computing and Information Science, Queen's University, Kingston, Ontario, November 1987.

[25] A. Bernstein and P. H. Jr., "Proving Real-Time Properties of Programs with Temporal Logic," in *Proc. Symposium on Operating Systems Principles*, 1981.

[26] C. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[27] N. Francez, D. Lehmann, and A. Pnueli, "A Linear History Semantics for Distributed Programming," *Theoretical Computer Science*, vol. 32, pp. 25–46, 1984.

[28] A. Salwicki and T. Müldner, "On the Algorithmic Properties of Concurrent Programs," in *Proceedings of Logic of Programs, LNCS 125*, 1979.

[29] H. Berg, W. Boebert, W. Franta, and T. Moher, *Formal Methods of Program Verification and Specification.* Prentice-Hall, Inc., 1982.

[30] C. Belzile, G. MacEwen, and G. Marquis, "RNet: A Hard Real-Time Distributed Programming System," *IEEE Transaction on Computers*, vol. C-36, pp. 917–932, August 1987.

[31] E. Kligerman and A. Stoyenko, "Real-Time Euclid: A Language for Reliable Real-Time Systems," *IEEE Transactions on Software Engineering*, vol. se-12, pp. 941–949, September 1986.

[32] S. Yang, *Timing Specification and Verification for Fault-Tolerant Distributed Computing Systems.* PhD thesis, Department of Computer Science and Engineering, University of South Florida, 1986.

[33] M. Hennessy, "Acceptance Trees," *J. ACM*, vol. 32, pp. 896–928, October 1985.

[34] E. Olderog and C. Hoare, "Specification-Oriented Semantics for Communicating Processes," *Acta Informatica*, vol. 23, pp. 9–66, 1986.

[35] I. Guessarian, *Algebraic Semantics.* Vol. 99 of *Lecture Notes in Computer Science*, Springer-Verlag, 1981.

[36] N. Dershowitz, "Orderings for Term-Rewriting Systems," *Theoretical Computer Science*, vol. 17, pp. 279–301, 1982.

[37] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smity, *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[38] M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation.* Vol. 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[39] N. Maxemchuk and K. Sabnani, "Probabilistic Verification of Communication Protocols," in *Protocol Specification, Testing and Verification*, North-Holland, 1987.

[40] R. Oberg, "Performance Modeling of High End Processors in the Evaluation of Real-Time Operating Systems," in *Proc. 2nd Workshop on Real-Time Operating Systems*, pp. 79–85, November 1984.

[41] A. Shaw, "Reasoning About Time in Higher-Level Language Software," Tech. Rep. 87-08-05, Department of Computer Science, University of Washington, August 1987.

[42] I. Lee, R. Gerber, and A. Zwarico, "Specifying Scheduling Paradigms for Time Dependent Processes," in *Submitted to 5th Workshop on Real-Time Software and Operating Systems*, 1988.

[43] I. Lee and S. Davidson, "Adding Time to Synchronous Process Communications," *IEEE Trans. on Comp.*, vol. c-36, pp. 941–948, August 1987.

[44] I. Lee and S. Davidson, "A Performance Analysis of Timed Synchronous Communication Primitives," Tech. Rep. MS-CIS-87-101, Dept. of CIS, University of Pennsylvania, November 1987.

[45] C. Gunter and D. Scott, "Semantic Domains," 1988. To appear in Handbook of Theoretical Computer Science.

[46] I. Lee, R. King, and X. Yun, "A Real-Time Kernel for Distributed Multi-Robot Systems," in *To appear in Proc. American Control Conf.*, June 1988.