A Logic for Real-Time Discrete Event Processes

Jonathan S. Ostroff

ABSTRACT: Temporal logic can be used to design controllers for real-time discrete event systems. The underlying plant dynamics is most economically described by a state transition structure with time bounds on the transitions. Temporal logic may then be used to 1) specify the problem to be solved and 2) help in the derivation and verification of a suitable controller.

Introduction

This article gives an overview of the use of temporal logic for the design of systems composed of real-time discrete event processes. Typical applications include process control, flexible manufacturing systems, robotics, communication networks, traffic systems, avionics and embedded real-time computer systems.

Timed transition models are used to model or represent real-time discrete event processes. Real-time temporal logic is the specification, verification and analysis language for reasoning about transition systems. The models economically describe the underlying state transition structure of plants and controllers, whereas the logic specifies at a high-level the control problem to be solved. The logic allows for the treatment of some infinite state systems because it is not necessary to check finite state reachability graphs for the existence of required properties. Instead, the logic provides the capability for performing an inductive correctness proof: the initial states must be acceptable, and every transition in the system preserves the desired behavior.

In the rest of this article we will first discuss the nature of real-time discrete event processes, next examine the software verification literature, give a brief overview of the model and logic framework, and finally provide a small example of controller design. The material of the first three sections is taken mainly from the introductory chapter of [30]. The reader is referred to [32], [30], [29] for complete technical detail, as this article only provides an overview of the key concepts.

Jonathan Ostroff is with the Department of Computer Science, York University, North York, Ontario, Canada, M3J 1P3. This work was supported by the Natural Sciences and Engineering Research Council of Canada.

Real-Time Discrete Event Systems

The main components of a discrete event process are its states and events. A state has duration in time, e.g., "the conveyer belt is moving" is a state that the conveyer might be in for some ticks of the clock. In the rest of this article a "state" in the above sense will be called an "activity," whereas the word state will be reserved for the "global state" of all the different devices and processes taken together. An event occurs instantaneously, e.g., "the conveyor fails." A transition from one state to another has the general form: "if the event α occurs in state A, then the condition C must be true and the system instantaneously transfers from A to state B." The following items include some of the characteristic features of real-time discrete event processes:

- Events occur at discrete times and states have discrete values.
- Processes are event-driven rather than clock-driven.
- Processes are typically nondeterministic (capable of "choices" by some mechanism unmodeled by the system analyst). No explicit stochastic feature is postulated—the focus is on the *possibility* rather than the *probability* of event occurrence.
- Processes generally have internal dynamic behavior, and also interact and react with their environments (the interaction is often nonterminating).
- Processes operate concurrently and communicate with each other (e.g., via message-passing over channels).
- "Hard" real-time deadlines must often be met for safe operation. System correctness depends not only on the logical result of the system behavior but also on the time at which the results are produced.

There are many examples where hard real-time deadlines must be met. If the temperature of a nuclear reactor core is too high an alarm must be generated within some deadline. Spray painting a car on a moving conveyor must be initiated at some suitable time and terminated some later time. When an aircraft enters an air traffic control region, the flight controller must be informed in a timely fashion. Once the approach of a train is detected, car and pedestrian traffic at the train intersection must be halted before the train reaches the intersection. If the computer controlling a robot does not command it to stop or turn in time, the robot might collide with another object on the factory floor.

There is general consensus in the software and control systems literature [44], [12], [5], [18], [41] that real-time discrete event systems are difficult to model, specify and design. In addition, experience has shown that software components of systems are problematic (perhaps even more so than mechanical or other hardware components).

Software is complex (consider the documentation needed for even simple modules), nonrobust (small errors have major consequences) and software is notoriously difficult to test (the number of test cases that must be checked becomes unmanageably large even in small systems) [33]. It does not come as a surprise that the first flight of the space shuttle was delayed by a subtle timing error, which was traced to an improbable race condition in the flight control software [10]. Although no loss of life occurred in this instance, other real-time computers are at the heart of systems in which software errors can have catastrophic consequences.

Since real-time discrete event processes occur in so many safety critical systems, there is obviously a need to model, specify, design and verify such processes to ensure their safe behavior.

It has long been conjectured that a formal mathematical approach would be useful in overcoming some of the difficulties in the specification, design and implementation of complex systems. Turning this conjecture into sound practice has proved to be extremely difficult—many practically-oriented engineers will probably consider the conjecture to face insurmountable hurdles.

But what benefits do "theorists" hope to obtain by the use of a formal framework? A list of the benefits includes:

- in the process of formalizing informal requirements, ambiguities, omissions, and contradictions will often be discovered,
- the formal model may lead to hierarchical semi-automated (or even automated) system development methods,
- the formal model can be verified for correctness by mathematical methods (rather than by intractable case by case testing),
- a formally verified subsystem can be incorporated into a large system with greater confidence that it behaves as specified, and
- different designs can be compared.

Control theorists have extensively studied the problem of designing continuous variable dynamic systems. Modern control theory has been successful in solving a variety of problems such as ensuring stability, regulation, filtering, and optimal control of large scale systems. The solution of these problems has been applied to many real world problems (e.g., the application of control theory to the Apollo moon landing). However, since the focus of control theory has been on continuous variable systems modeled by differential or difference equations, it is ill-equipped to cope with systems in which discreteness, modularity and communication are fundamental [18], [45]. In addition, modern control theory is often mathematically too specific to model qualitative system properties. An analogy (cited in [15]) compares the abstract and discrete description of digital logic circuits via boolean functions, to the lower-level view of the digital circuit as a continuously changing numerical vector. The boolean description is useful because it eliminates unnecessary numerical detail. In response to the need to develop theories for discrete event systems, control theorists have recently investigated formal language theory and mathematical logic [42], [37], [15], [4], [3].

In the software verification literature, mathematical logic has played an important role in the specification and analysis of program correctness. Some of these ideas can be used in the design of control systems.

Software Verification

In the software verification literature, much attention has been focused on the description and verification of parallel programs (see [2] for a survey and comparison of methods). Although the focus is usually on programs, many of the techniques are applicable either explicitly or implicitly to complex discrete event systems having mechanical or other hardware components as well as software components (e.g., [24], [34], [8], [17], [43], [23], [13], [36], [11]).

There is an important distinction to be made between a "dual-language" approach and a "single-language" approach [39]. The timed transition/real-time temporal logic framework of this paper is an extension of the dual-language fair transition systems/temporal logic framework introduced by Manna and Pnueli [35], [21], [19], [36].

The single-language approach (e.g., [24]) works on the premise that the same language should be used both for specification and implementation of computational tasks. Thus the language should have a well-identified fragment that can be effectively and efficiently executed. The initial specification emphasizes the desired behavior and pays little attention to implementation details. A sequence of transformations is then applied to the initial specification to produce the final implementation. An algebraic framework usually supports the transformations so that meaning is preserved.

In the dual-language approach, one language is the computer executable programming (or representation) language R, which is prescriptive and algorithmic in nature. R is the language in which we program and which is computer executable. The second language is the specification (or assertion) language S, which is descriptive and expressive enough to specify program requirements (without specifying how the programs are to be implemented). Language S is usually based on mathematical logic.

In the dual-language approach, the main problems that are studied are program verification, development and synthesis. Given a specification $s \in S$ and a program $r \in R$, the verification problem involves the demonstration that the program r satisfies the specification $s \in S$, and a method is sought whereby disciplined humans can be helped to construct a program r that satisfies s. If program development is fully automated, then it is called program synthesis.

Program synthesis is closely related to controller synthesis. In controller synthesis, a plant $p \in R$ and a specification $s \in S$ of the control problem to be solved are provided. The problem is then to automatically derive a controller $c \in R$ that in conjunction with p will ensure that the closed loop system satisfies s.

In what way is controller design different from program synthesis? The representation language R must be flexible enough to model not only programs but also the plant dynamics (e.g., there must be spontaneous as well as controlled events). A mechanism for observation and control must be provided. Furthermore, in program synthesis the designer is free to develop any program from scratch that will satisfy the specification. By contrast, in controller design, there is a fixed component (viz., the plant) that cannot be altered by the designer, and yet it is the appropriate behavior of the plant that is of vital concern. Finally, in program synthesis the designer is mostly interested in ensuring that the program delivers certain services to the environment. By contrast, the purpose of the controller is to ensure that the internal behavior of the plant satisfies appropriate requirements.

The TTM/RTTL Framework

The connection between timed transition models (TTMs) and real-time temporal logic (RTTL) is made via an extension to Manna and Pnueli's notion of a fair transition system. A fair transition system consists of a (possibly infinite) state space, a set of transitions defining state transformations, a set of initial states, and a fairness family. Fairness means that if a process is continuously (or infinitely often) enabled then it must eventually make progress. Although fairness notions are easily incorporated into the TTM/RTTL framework [30], for simplicity we will not further develop these notions in this paper.

A computation is a sequence of states whose initial state is in the initial state set, whose successor states are obtained by applying enabled transitions, and where the fairness families are used to ensure that certain sets of transitions that are continuously enabled or infinitely often enabled in the computation eventually occur. A computation can be used to interpret formulas of temporal logic. To determine if a concurrent program satisfies a temporal logic specification S, the program is first translated into a fair transition system. Translation is usually a straightforward, easily automated process. The program satisfies S exactly when all the program computations satisfy S.

The following extensions are made to fair transition systems to obtain the TTM/RTTL framework:

- Transitions have associated lower and upper time bounds (measured with respect to a global clock) so that real-time properties (including delays and timeouts) can be represented.
- Two distinguished variables are added: a time variable t (the current clock time) and the next-transition variable **n** (for referring to the events of complex systems).
- Instead of computations, the transition

system now generates trajectories or sequences of states that take into account the lower and upper time bounds of transitions.

- The trajectories can be interpreted in real-time temporal logic (which uses standard operators together with the time variable to express quantitative timing properties). No expressive power is lost in making the transition from a fair transition system to the TTM/RTTL framework. The upper and lower time bounds impose additional constraints on transitions, thus necessitating additional proof rules for verifying real-time properties, but preserving the standard temporal logic proof rules for verifying qualitative behavior.
- TTMs are useful for representing the processes of discrete event systems. TTMs model spontaneous events, controlled events, nondeterministic choice of events, events shared by two or more TTMs, inter-TTM communication and parallel composition of TTMs. TTMs can also represent the constructs of real-time programming languages.

The plant of a complex discrete event system can be represented by a TTM PLANT (which itself is composed of many TTMs with each TTM representing one of the plant processes). A specification *S* of required plant behavior can be formulated in RTTL. A controller (possibly implemented in real-time concurrent programming language) can be represented by a TTM CONTROLLER. The controller verification problem can be stated as follows: given PLANT and CONTROLLER, check if PLANT$\|$CONTROLLER (the parallel composition of plant and controller) satisfies *S*. The controller synthesis problem can be stated as: given a specification *S* for PLANT, is there an automatic way to produce CONTROLLER so that the closed loop system given by PLANT$\|$CONTROLLER satisfies *S*.

RTTL specifications may be applied to any model of computation addressed by fair transition systems including: shared variables, asynchronous communication, and Petri net models [36], [30].

The state machine formalism is one of the oldest models of digital computing. It has been used extensively in requirements specifications for real-time computing [46], [1], [11] because it is a natural, visual medium for describing the dynamic behavior of a complex system. This is the reason for choosing TTMs for representing plants and controllers. TTMs are (possibly infinite) state machines, enhanced with the standard programming notions of data variables, guarded events, concurrency and communication.

Temporal Logic

Temporal logic has its origins in philosophy, where it was used to analyze the nature of time [38]. In recent years, it has found application in computer science, especially in the areas of software verification and knowledge-based systems [7].

In physics and mathematics, time has traditionally been represented as just another variable. First order predicate calculus is used to reason about expressions containing the time variable, and there is thus apparently no need for a special temporal logic.

Philosophers found it useful to introduce special temporal operators, such as $\Box$ (henceforth) and $\Diamond$ (eventually), for the analysis of temporal connectives in language. The new formalism was soon seen as a potentially valuable tool for analyzing the topology of time. For example, various types of semantics can be given to the temporal operators depending on whether time is linear, parallel of branching. Another question that may be asked is whether time is discrete or continuous.

The temporal operators have been found useful for specifying program behavior. A structure of states (e.g., a sequence or tree of states) is the key concept that makes temporal logic suitable for program specification. A formula, containing temporal logic operators, is interpreted over a structure of states. In programming languages, the structures represent the computations executed by a program. Such a computation may be used to interpret a temporal formula. In this way, a programming language is said to be endowed with a temporal semantics.

Some of the different types of temporal semantics include:

- *Interval semantics* [40], [25], [26]. The semantics is based on intervals of time, thought of as representing finite chunks of system behavior. An interval may be divided into two contiguous subintervals, thus leading to the "chop" operator ";". "*A*; *B*" is true on an interval just so long as that interval can be decomposed into two contiguous subintervals in such a way that *A* is true over the first interval and *B* over the second.
- *Point semantics*, in which temporal formulas are interpreted as requiring some system behavior with respect to a certain reference point in time. Past operators refer to the time prior to the reference point, and future operators to the time after the reference point. Obviously, a point cannot be divided, and the there is thus no simple definition of the chop operator.

  Point semantics may be further divided into three classes.
  - *Linear semantics* [22], [36], [20]. In linear semantics, each moment has only one possible future corresponding to the actual history of the development of the system.
  - *Branching semantics* [9], [6]. In branching time semantics, time has a tree-like nature in which, at each instant, time may split into alternative courses representing different choices made by a system.
  - *Partial order semantics*. Partial order semantics has been explored only recently. The reader is referred to [20] and other articles in the same volume as [20] for further information.

RTTL is based on Manna-Pnueli temporal logic, and therefore has a linear, point semantics. Having decided on the type of structure to be used for interpreting temporal formulas, there is still a further decision to be made, *viz.*, how are the structures to represent program executions or computations. There are at least two possibilities.

- *Maximal parallelism* [14], [16]: The number of instructions in concurrent processes that can be executed simultaneously is maximized. Thus, two processes are never both waiting to achieve a shared communication.
- *Interleaved executions:* Concurrent activity of two parallel processes is represented by interleaving their atomic actions (interleaving is conceptually similar to automata products). Fairness and time bound constraints are then used to exclude inappropriate interleavings (e.g., a sequence in which a continuously enabled action is never executed).

The trajectories (or executions) of concurrent TTMs are constructed from interleavings of their atomic actions. TTMs may therefore be thought of as being endowed with a linear order temporal semantics, in which the trajectories of TTMs are constrained to real-time, fair interleavings.

RTTL uses as its base the linear time quantified temporal logic proof system in [22]. The proof system uses the future fragment of temporal logic. Thus, although the future behavior of a system is easily described, there are no temporal operators which directly express past behavior, e.g., "once in the past the device failed." Past properties can be expressed indirectly, but not in the same natural fashion as future properties.

An Example

Fig. 1 shows two TTMs called TRAIN and GATE representing two concurrent processes. Before the train (the first process) reaches a level-crossing, a gate (the second process) is to be lowered so as to prevent pedestrian or automobile traffic from crossing over while the train is passing through.

A TTM M is defined as a 3-tuple ($\mathcal{V}$, Θ, $\mathfrak{T}$) consisting of a set of variables $\mathcal{V}$, an initial condition Θ and a finite set of transitions $\mathfrak{T}$. The initial condition is any boolean valued expression in the variables. The transitions are explained in more detail below. An example of a TTM is TRAIN which is defined by

$$\text{TRAIN} \overset{\text{def}}{=} (\mathcal{V}_{\text{TRAIN}}, \Theta_{\text{TRAIN}}, \mathfrak{T}_{\text{TRAIN}})$$

where $\mathcal{V}_{\text{TRAIN}} = \{x_1, t, \mathbf{n}\}$, Θ_{TRAIN} is defined by (x_1 = traveling $\wedge$ $\mathbf{n}$ = initial), and $\mathfrak{T}_{\text{TRAIN}}$ = $\{\alpha_1, \alpha_2, \delta, \text{initial}, \text{tick}\}$ as listed in Table I.

The two distinguished variables t (the current time) and $\mathbf{n}$ (the next transition variable) are present in all TTMs. Each TTM has an activity variable, e.g., for TRAIN the activity variable is x_1 with $type(x_1)$ = {traveling, approaching, ingate}. In addition, there may be one or more data variables for representing numeric information (such as pressures, temperatures or fluid levels) or other information (such as sets and queues). The TTM GATE has a data variable y which counts the number of times the gate is lowered after each maintenance cycle.

Each $v \in \mathcal{V}$ has an associated type(v), and $\mathcal{R}$ is the disjoint union of all these types. For the time variable, type(t) is usually the nonnegative integers union with infinity, and type($\mathbf{n}$) = $\mathfrak{T}$.

A state s of a TTM M = ($\mathcal{V}$, Θ, $\mathfrak{T}$) is a mapping s: $\mathcal{V} \to \mathcal{R}$ such that $s(v) \in \text{type}(v)$ for each $v \in \mathcal{V}$. Each state s has an associated state-assignment q which is the restriction of s to ($\mathcal{V}$ − {$\mathbf{n}$}). An example of a state of TRAIN is s_2 = {(x_1, traveling), (t, 1), ($\mathbf{n}$, α_1)}, and the corresponding state-assignment is q_2 = {(x_1, traveling), (t, 1)}. The set of all state-assignments is $\mathcal{Q}$.

A transition (which updates e.g., 2 data variables y_1, y_2) may be visualized by the illustration shown in Fig. 2. The transition may be interpreted as follows: "if the transition τ becomes enabled at time $t = T$ (e.g., the TTM reaches the activity a_s with the guard *guard* evaluating to true), then the edge must be traversed between l and u ticks from T, unless τ is preempted by the occurrence of some other transition which disables τ. When τ occurs in a state-assignment q, the successor state-assignment q' is similar to q except x has the value a_d, and y_1, y_2 have the values of d_1, d_2 respectively as evaluated in the state q."

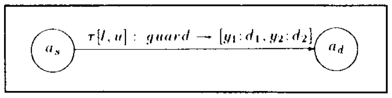


Fig. 2. A transition of a time transition model.

A transition τ is defined by a 4-tuple (e_τ, h_τ, l_τ, u_τ), where l_τ, u_τ are the lower and upper time bounds, respectively. The bounds may be any integer expression d so long as $s(d) \in \text{type}(t)$. The enabling condition is e_τ defined by ($x = a_s \wedge guard$). Where no guard is indicated on the transition graph, the guard is assumed to be true. The transformation function is a partial function h_τ: $\mathcal{Q} \to \mathcal{Q}$, where $\mathcal{Q}$ is the set of all state-assignments. The function is defined for all state-assignments $q \in \mathcal{Q}$ such that $q(e_\tau)$ = true. The transformation function is often denoted by [x: a_d, y_1: d_1, y_2: d_2] to indicate which variables in the state-assignment are updated on making the transition.

The train transition α_2 has a lower time bound of 10, which represents the fact that it takes a minimum of 10 clock ticks from the time the train is detected until it reaches the gate. An upper time bound of infinity means that the transition is never forced to happen. A legal-trajectory σ of TRAIN is any actual possible sequence of states that TRAIN may execute. For example, the following is a legal trajectory of TRAIN which starts with the initial transition initial followed by one tick of the clock:

$$\sigma \overset{\text{def}}{=} q_0 \xrightarrow{\text{initial}} q_1 \xrightarrow{\text{tick}} q_2 \xrightarrow{\alpha_1} q_3 \xrightarrow{\text{tick}^{10}} q_4 \xrightarrow{\alpha_2} q_5 \cdots \to \cdots = s_0 s_1 s_2 s_3 \cdots . \qquad (1)$$

The train then approaches the level-crossing, followed by 10 ticks of the clock. Finally the train reaches the level-crossing. The rest of the legal-trajectory is not shown.

In addition to the transitions that arise from the transition graph of a TTM, the transition set of any TTM always contains two distinguished transitions *tick* and *initial* as indicated in (1). The *tick* transition must occur infinitely often in any legal-trajectory, and if s_0 is the initial state of a legal trajectory then $s_0(\Theta)$ = true. Each state-assignment in the legal trajectory is related to its predecessor via the transformation function of the corresponding transition.

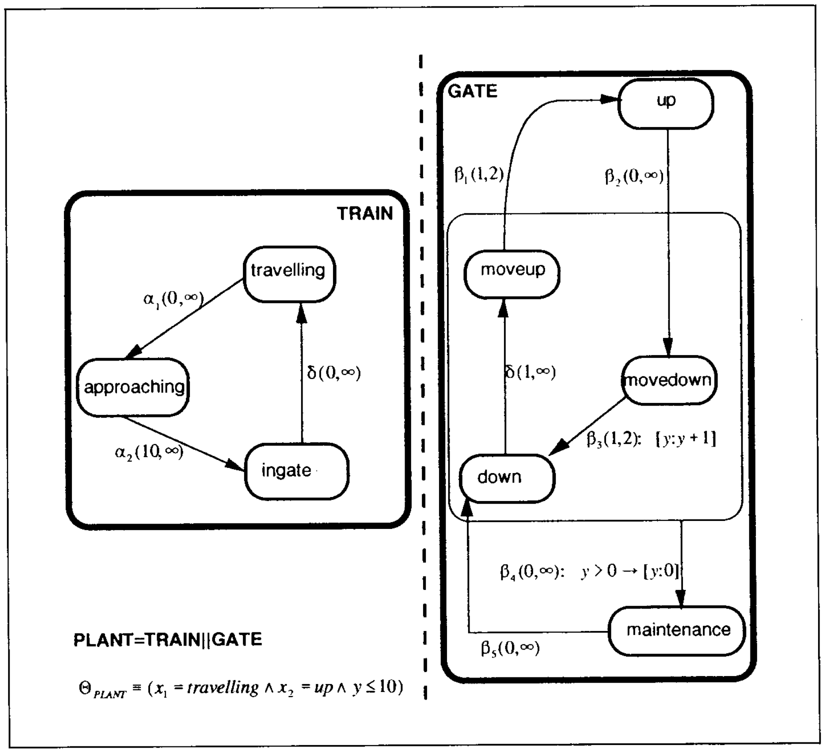


Fig. 1. The plant consisting of a train and gate.

Σ_M denotes the set of all legal trajectories of a TTM M. Σ_M completely characterizes the behavior of M, and thus provides an abstract operational semantics for the TTM.

In the GATE TTM of Fig. 1, typical statechart [11] notation is used. The activities *moveup*, *down*, *movedown* are clustered into a superstate. Thus, to indicate that there is a transition from each of the activities in the superstate to the maintenance activity, only a single edge β_4 need be shown (thus replacing three edges by one). To be in the superstate one must be in exactly one of moveup, down, movedown. The superstate together with β_4 is really an abstraction of a common property of the clustered activities, *viz.*, that β_4 leads from them to maintenance. Abstract edges such as β_4 are easily represented in TTMs. See Table I in which β_4 has the enabling condition $x_2 \in \{\text{moveup, down, movedown}\}$, the transformation function $[x_2\text{: maint}]$ and lower and upper time bounds of 0 and ∞, respectively. The notation $x_2 \in \{\text{moveup, down, movedown}\}$ is an abbreviation for $(x_1 = \text{moveup} \vee x_2 = \text{down} \vee x_3 = \text{movedown})$—since activities are distinct, the "exclusive or" could be used.

The transition β_3 in GATE increments the data variable y by one every time the gate is lowered. The data variable y thus counts the number of times the gate has been lowered. The free behavior of GATE (before it is composed with other TTMs) allows for an infinite number of states to be reached as GATE has legal trajectories which infinitely often increment y. A controller might use the counter to force a maintenance activity after the lowering mechanism has been used a certain number of times. The counter is reset to zero (by β_4) every time the gate is maintained.

When the gate is down it is possible for two transitions (i.e., δ and β_4) to be simultaneously enabled; this represents the nondeterministic case in which any of the enabled transitions may be taken.

Table I
The Transition Sets of TRAIN, GATE, and PLANT

Transitions of TRAIN

Name	Enabling condition	Transformation	Lower	Upper
α_1	$(x_1 = \text{tr})$	$[x_1 : \text{ap}]$	0	∞
α_2	$(x_1 = \text{ap})$	$[x_1 : \text{in}]$	10	∞
δ	$(x_1 = \text{in})$	$[x_1 : \text{tr}]$	0	∞

Transitions of GATE

Name	Enabling condition	Transformation	Lower	Upper
β_1	$(x_2 = \text{mvup})$	$[x_2 : \text{up}]$	1	2
β_2	$(x_2 = \text{up})$	$[x_2 : \text{mvdn}]$	0	∞
β_3	$(x_2 = \text{mvdn})$	$[x_2 : \text{down}, y : y + 1]$	1	2
β_4	$(x_2 \in \{\text{down, mvup, mvdn}\} \wedge y > 0)$	$[x_2 : \text{maint}, y\text{: } 0]$	0	∞
β_5	$(x_2 = \text{maint})$	$[x_2 : \text{down}]$	0	∞
δ	$(x_2 = \text{down})$	$[x_2 : \text{mvup}]$	1	∞

Transitions of PLANT = TRAIN‖GATE

Name	Enabling condition	Transformation	Lower	Upper
α_1	$(x_1 = \text{tr})$	$[x_1 : \text{ap}]$	0	∞
α_2	$(x_1 = \text{ap})$	$[x_1 : \text{in}]$	10	∞
δ	$(x_1 = \text{in} \wedge x_2 = \text{down})$	$[x_1 : \text{tr}, x_2 : \text{mvup}]$	1	∞
β_1	$(x_2 = \text{mvup})$	$[x_2 : \text{up}]$	1	2
β_2	$(x_2 = \text{up})$	$[x_2 : \text{mvdn}]$	0	∞
β_3	$(x_2 = \text{mvdn})$	$[x_2 : \text{down}, y : y + 1]$	1	2
β_4	$(x_2 \in \{\text{down, mvup, mvdn}\} \wedge y > 0)$	$[x_2 : \text{maint}, y : 0]$	0	∞
β_5	$(x_2 = \text{maint})$	$[x_2 : \text{down}]$	0	∞

The two distinguished transitions tick and initial are excluded from the tables as they are always present. The tick transition has enabling condition true and transformation function $[t : t + 1]$. Since all time bounds are relative to clock ticks, the clock tick itself has no bounds. The transition tick is the only transition that changes the value of t. The enabling condition of initial is true, the transformation function is [ ] (i.e., nothing changes) and the lower and upper time bounds are both zero.

Parallel Composition

In Table I, the transition set of TRAIN and GATE each have a transition called δ—such transitions which share their label names are called shared-transitions. When composing two concurrent TTMs each containing a component of a shared-transition, a new composite transition (also labeled δ) in the composed TTM is formed. The composed transition represents the combined simultaneous interaction of the component transitions. For example, the δ component in TRAIN (i.e., exit from the level crossing) and GATE (i.e., start raising the gate) occur simultaneously thus modeling a mechanical interlock, which starts raising the gate whenever the train exits. In the transition table (Table I) for PLANT = TRAIN ‖ GATE the composite transition δ is shown.

Given any two component transitions $A = (e_A, h_A, l_A, u_A)$ from TTM M_A and $B = (e_B, h_B, l_B, u_B)$ from TTM M_B, the composite transition C is given by $C = (e_C, h_C, l_C, u_C)$ where e_C is defined by $e_A \wedge e_B$, l_C is defined by max (l_A, l_B), and u_C is defined by min (u_A, u_B). If $h_A = [v_1\text{: } d_1, \cdots, v_k\text{: } d_k]$ and $h_B = [v_{k+1}\text{: } d_{k+1}, \cdots, v_{k+n}\text{: } d_{k+n}]$, then h_C is defined by $[v_1\text{: } d_1, \cdots, v_{k+n}\text{: } d_{k+n}]$. To aid in the easy identification of shared-transitions, A, B, and C are all given the same transition label name (where no confusion will result) as illustrated in the case of the δ transition.

The above method for computing the composite bounds allow shared transitions to be used for control. For example, some TTM M_1 may impose control over a transition τ in another TTM, M_2 by including a transition (with some finite upper time bound) with the same label τ in M_1, thus forcing the composite transition τ to occur by the upper time bound whenever the components are simultaneously enabled.

The definition of parallel composition of two TTMs can now be given. Let M_1 and M_2 be two TTMs, i.e.

$$M_1 = (\mathfrak{V}_{M_1}, \Theta_{M_1}, \mathfrak{J}_{M_1})$$

$$M_2 = (\mathfrak{V}_{M_2}, \Theta_{M_2}, \mathfrak{J}_{M_2}).$$

Then the composition $M_3 = M_1 \| M_2$ is given by

$$M_3 \overset{\text{def}}{=} (\mathfrak{V}_{M_1} \cup \mathfrak{V}_{M_2}, \Theta_{M_1} \wedge \Theta_{M_2}, \mathfrak{J}_{M_1} \| \mathfrak{J}_{M_2}). \qquad (2)$$

If there are no shared-transitions then $\mathfrak{J}_{M_1} \| \mathfrak{J}_{M_2}$ is just the union $\mathfrak{J}_{M_1} \cup \mathfrak{J}_{M_2}$ of the two transition sets. For each shared transition, the components of the shared transition must be removed from the union and replaced by the resulting composite transition. The parallel composition of TTMs is itself a TTM.

In [30], [32], parallel composition involving communicating transitions (sending and receiving a message) is also defined. Parallel composition is important because it allows the designer to build complex systems from smaller component parts.

An example of parallel composition is provided in Table I in the case of PLANT = TRAIN $\|$ GATE.

Imposition of Control

A controller is a TTM which is composed in parallel with PLANT thereby imposing control over some of the actions of the plant in order to achieve suitable plant behavior.

The set of observable variables of PLANT is some subset of $\mathcal{V}_{\mathrm{PLANT}}$ (time t is always observable and $\mathbf{n}$ is never observable). Observable variables are treated by the plant as "read-only," i.e., the observable variables may be used in guards of plant transitions and to compute updates to controller data variables—however, plant variables may never be changed by controller transitions.

The set of controllable transitions of PLANT is some subset of $\mathcal{T}$ (*tick* and *initial* are never controllable). Controllable transitions are transitions that the controller can treat as shared-transitions, i.e., a controllable transition τ matching a plant transition (also called τ) may occur anywhere in the controller. The guard of τ in the controller may use any of the observable plant variables. Furthermore the bounds of τ in the controller may be set to any values so long as the composite transition has its upper time bound at least as large as its lower time bound. The controller can therefore disable the corresponding plant transition under some conditions (via the guards) or force the occurrence of the plant transition (via the time bounds and guards) under other conditions.

In the case of PLANT = TRAIN $\|$ GATE the observable variables could be $\{x_1, x_2, y\}$ and the controllable transitions $\{\beta_2, \beta_4\}$. The transition β_2, which denotes the command to the gate to move down, is a typical controllable transition. The transition α_1 is a typical uncontrolled (or "spontaneous" transition) because the controller cannot determine what the train will do (in this case start approaching the gate). Other examples of spontaneous events are failures, external interrupts and operator actions such as pushing on/off buttons.

The free behavior of the plant is described by the legal trajectories of PLANT prior to control, i.e., the controllable transitions are treated as spontaneous transitions. The behavior of the open loop plant is usually unsatisfactory in some respect—specifying appropriate behavior is the task of the next section.

Specification of the Control Problem to be Solved

The control problem to be solved will be specified using real-time temporal logic (RTTL). A brief description follows below. For more detail the reader is referred to [30], [32].

A *state-formula* is any boolean-valued expression in the variables of a TTM, e.g., the state-formula $(\mathbf{n} = \alpha_2 \wedge t \leq 10 \wedge (x_1 = \text{approaching} \vee y = 1))$ is satisfied in the state $s = \{(x_1, \text{ingate}), (y, 1), (t, 10), (\mathbf{n}, \alpha_2)\}$.

Unlike a state-formula which can be evaluated in a single state, an RTTL formula can only be evaluated in a trajectory (infinite sequence of states). For simplicity, we use two basic temporal operators $\bigcirc$ (next), and $\mathcal{U}$ (until) from which we can define many other useful operators including: $\Box$ (henceforth), $\Diamond$ (eventually), U (unless), and $\mathcal{P}$ (precedes).

For an arbitrary trajectory $\sigma = s_0 s_1 s_2 \cdots$, denote by σ^k the k-shifted trajectory suffix given by $\sigma^k = s_k s_{k+1} s_{k+2} \cdots$. The satisfaction relation for arbitrary temporal formulas is defined inductively as follows (the notation $\models^{\sigma} w$ means that the trajectory σ satisfies the formula w): For temporal formulas w, w_1, w_2, and trajectory σ:

- if w is a state-formula, then $\models^{\sigma} w$ iff $s_0(w)$ = true.
- $\models^{\sigma} \bigcirc w$ iff $\models^{\sigma^1} w$,
  $\bigcirc w$ may be paraphrased: w will be true in the *next* stage.
- $\models^{\sigma} w_1 \mathcal{U} w_2$
  iff $\exists k \geq 0$ such that $\models^{\sigma^k} w_2$ and $\forall i$, $0 \leq i < k$, $\models^{\sigma^i} w_1$.
  $w_1 \mathcal{U} w_2$ may be paraphrased: eventually w_2 will hold and *until* then w_1 holds continuously.

New temporal operators may be defined from $\bigcirc$ and $\mathcal{U}$ as follows:

- $\Diamond\, w$ is an abbreviation for (true $\mathcal{U}\, w$).
- $\Diamond\, w$ may be paraphrased: *eventually* w will hold true in some state.
- $\Box w$ is an abbreviation for $\neg(\Diamond(\neg w))$. $\Box w$ may be paraphrased: *henceforth*, w holds true in all states.
- $w_1 \mathcal{P} w_2$ is an abbreviation for $(\neg((\neg w_1)\mathcal{U} w_2))$.
  $w_1 \mathcal{P} w_2$ may be paraphrased: if w_2 eventually occurs then w_1 must *precede* w_2.
- $w_1 U w_2$ is an abbreviation for $(\Box w_1) \vee (w_1 \mathcal{U} w_2)$.
  $w_1 U w_2$ may be paraphrased: w_1 holds true unless w_2 becomes true, i.e., either w_2 never occurs and w_1 is henceforth true, or w_1 holds true until the first occurrence of w_2.

The temporal formula $\Box \Diamond (\mathbf{n} = \text{tick})$ can be read as: "the clock ticks infinitely often." The formula

$$(w_1 \wedge t = T) \rightarrow \Diamond (w_2 \wedge (t < T + 4)) \tag{3}$$

may be read as: "if w_1 is true at time T then w_2 must happen before the clock reads $T + 4$ (i.e., within 4 clock ticks)."

In (3), the variable T is a "global" variable that has the same value in every state of a trajectory. In RTTL formulas quantification is allowed over global variables but not over the "local" variables (variables in $\mathcal{V}$); local variables always occur free. Global variables range over fixed data domains (e.g., T ranges over type(t)) and denote elements thereof. Local variables change from state to state.

All formulas with occurrence of global variables are assumed to have their global variables universally quantified. Thus (3) actually means

$$\forall T[(w_1 \wedge t = T) \rightarrow \Diamond (w_2 \wedge (t < T + 4))].$$

Instead of using the time variable t explicitly in RTTL formulas we may use abbreviations such as:

- $\Diamond_{[l,u]} w$ for $[(t = T) \rightarrow \Diamond (w \wedge (T + l \leq t \leq T + u))]$, i.e., eventually between l and u ticks from now w will become true. The formula $\Diamond_{\leq d} w$ abbreviates $\Diamond_{[0,d]} w$, and $\Diamond_d w$ abbreviates $\Diamond_{[d,d]} w$.
- $w_1 \mathcal{U}_{[l,u]} w_2$ abbreviates $[(t = T) \rightarrow (w_1 \mathcal{U} (w_2 \wedge (T + l \leq t \leq T + u)))]$, i.e., w_2 will become true between l and u ticks from now, and until then w_1 remains true. The formula $(w_1 \mathcal{U}_d w_2)$ abbreviates $(w_1 \mathcal{U}_{[d,d]} w_2)$, and $(w_1 \mathcal{U}_{\leq d} w_2)$ abbreviates $(w_1 \mathcal{U}_{[0,d]} w_2)$.

A variety of real-time properties can be expressed using these abbreviations including:

Exact time: $\Box (w_1 \rightarrow \Diamond_d w_2)$—every w_1 is followed by a w_2 in exactly d ticks.

Maximum time: $\Box (w_1 \rightarrow \Diamond_{\leq d} w_2)$—every w_1 must be followed by a w_2 within d ticks.

Minimum time: $\Box(w_1 \rightarrow \neg \Diamond_{\leq d} w_2)$—$w_1$ and w_2 are at least d ticks apart.

Periodicity with period d: $\Diamond w \wedge \Box[w \rightarrow \bigcirc(\neg w \mathcal{U}_d w)]$—$w$ occurs regularly with an exact period of d.

Let S be an RTTL specification characterizing the behavior that a TTM M is required to satisfy. Then S is M-valid iff all legal trajectories of M satisfy S.

Development of a Controller

The designer may want to design a controller so that PLANT will satisfy the following specifications.

S1—Safety

$$\Box[x_1 \neq \text{ingate} \vee x_2 = \text{down})]$$

i.e., henceforth, if the train is on the level crossing, the gate must be down.

The above property cannot be made PLANT-valid if maintenance procedures are to be performed on the gate. A more realistic specification is thus

$$\Box(x_2 \neq \text{maintenance})$$
$$\rightarrow \Box[x_1 \neq \text{ingate} \vee x_2 = \text{down}] \tag{4}$$

i.e., so long as no maintenance is henceforth performed, when the train crosses the gate must be down.

Another safety property is $\Box(y \leq 125)$, i.e., the gate should never be used more than 125 times without going through the maintenance procedures.

S2—No Unsolicited Response

$$\Box[(x_2 = \text{down} \wedge x_1 = \text{ingate})$$
$$\rightarrow (\mathbf{n} = \alpha_1 \mathcal{P} \mathbf{n} = \beta_2)]$$

i.e., henceforth if the gate is down and the train is crossing then α_1 must precede the next occurrence of β_2. Thus, the gate may be lowered only after the approach of the train. As in the case of (4) the complete specification is an implication with $\Box x_2 \neq$ maintenance) as the antecedent.

S3—Real-time Response

$$\Box[(\mathbf{n} = \alpha_1) \rightarrow \Diamond_{\leq 9}(\mathbf{n} = \beta_3)]$$

i.e., henceforth the gate must be lowered no more than 9 ticks after the train approaches the level crossing. As in the case of (4) the complete specification is an implication with antecedent $\Box(x_2 \neq \text{maintenance})$.

None of the above specifications is valid for the free behavior of the plant. A controller is therefore needed to impose the proper controls.

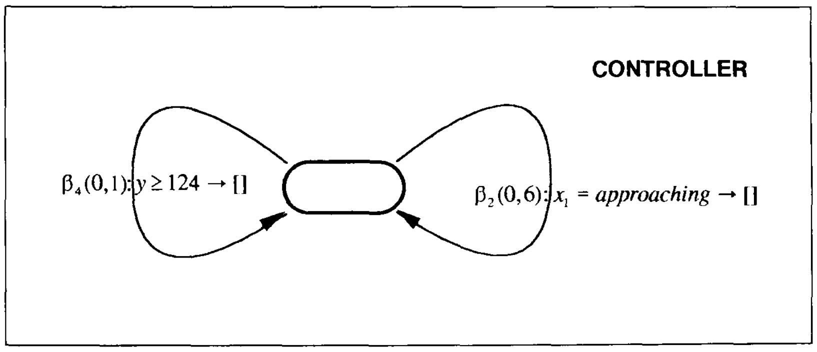


Fig. 3. The timed transition model CONTROLLER.

The control problem for the train-gate plant may now be specified as follows: given that the variables x_1, y of PLANT are observable, and given that the transitions β_2, β_4 are controllable, find a TTM CONTROLLER, so that for SYSTEM = PLANT‖CONTROLLER, the formulas $S1$, $S2$, and $S3$ are SYSTEM-valid.

Systematic development of controllers for safety properties using weakest preconditions is described in [29] and heuristics for real-time properties in [30]. A typical controller for PLANT is provided in Fig. 3. Note that a controller transition $\beta_2(0, 11)$ will not satisfy specifications $S1$ and $S3$, whereas $\beta_2(0, 6)$ does.

Because the synthesis procedures employ the logical manipulation of RTTL formulas, the synthesis procedures are applicable to finite state as well as infinite state TTMs. Complexity of verification in the finite state case is discussed in [28].

Concluding Remarks

The example of the previous section indicates how temporal logic can be used to design controllers. An important area of related interest involves the automation of verification procedures, i.e., given a TTM and a specification of required behavior, automatically check whether the TTM satisfies its specification. The references below explore finite state and infinite state verification.

Automated Finite State Verification

The articles in [27], [31], [28] give procedures for verifying safety, liveness and real-time response properties of TTMs with finite state reachability graphs. The procedures have complexity linear in the size of the reachability graph. The procedures have been implemented in Prolog and are available from the author.

Semi-automated Verification of Infinite State Systems

Constraint logic programming (CLP) has been used to semi-automate the verification of some infinite state systems [31], [30]. Constraint logic facts and rules represent TTMs in a natural fashion that stays close to their mathematical representation. The knowledge about TTMs is separated from the use that the knowledge is put to. Thus it is relatively easy to effect changes in the TTMs. The designer is aided in the construction of RTTL proofs of TTM correctness. The problem of combinatorial explosion of states is somewhat alleviated—CLP exploits the rich structure provided by real numbers to solve real-valued constraints using the simplex method for example, instead of solving constraints by state enumeration.

Current research is focused on developing modular controller synthesis methods and the design of hierarchical controllers.

References

[1] B. Auernheimer and R. A. Kemmerer, "RT-ASLAN: A specification language for real-time systems," *IEEE Trans. Software Eng.*, vol. SE-12, no. 9, pp. 879-889, Sept. 1986.

[2] H. Barringer, *A Survey of Verification Techniques for Parallel Programs*, vol. 191 of *LNCS*. Berlin: Springer-Verlag, 1985.

[3] A. Benveniste and P. Le Guernic, "Hybrid dynamical systems theory and nonlinear dynamic systems over finite field," in *Proc. 27th IEEE Conf. Decision and Control*, Austin, TX, Dec. 1988, pp. 209-213.

[4] K. P. Brand and J. Kopainsky, "Principles and engineering of process control with Petri nets," *IEEE Trans. Automatic Control*, vol. 33, no. 2, pp. 138-149, Feb. 1988.

[5] J. F. Cassidy, T. Z. Chu, M. Kutcher, S. B. Gershwin, and Y. Ho, "Research needs in manufacturing systems," *IEEE Control Systems Mag.*, vol. 5, no. 3, pp. 11-13, Aug. 1985.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic," *ACM Trans. Programming Languages Syst.*, vol. 8, no. 2, pp. 244-263, Apr. 1986.

[7] A. Galton, Ed., *Temporal Logics and Their Applications*. Academic, 1987.

[8] E. A. Emerson and E. C. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," *Sci. Computer Prog.*, vol. 2, pp. 241-266, 1982.

[9] E. A. Emerson and J. Y. Halpern. "'Sometimes' and 'not never' revisited: On branching versus linear time temporal logic," *J. Assoc. Computing Machinery*, vol. 33, no. 1, pp. 151-178, Jan. 1986.

[10] J. R. Garman, "The bug heard round the world," *ACM SIGSOFT Software Engineering Notes*, vol. 6, no. 5, 1981.

[11] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comp. Prog.*, vol. 8, pp. 231-274, 1987.

[12] D. Harel and A. Pnueli, "On the development of reactive systems," in K. R. Apt, Ed., *Logics and Models of Concurrent Systems*, vol. 13, *NATO ASI*, Springer-Verlag, 1985, pp. 477-498.

[13] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[14] J. Hooman and J. Widom, "A temporal logic based compositional proof system for real-time message passing," Department of Computer Science, Cornell University, Ithaca, NY, June 1988, Tech. Rep. TR-88-919.

[15] K. Inan and P. P. Varaiya, "Finitely recursive process models for discrete event systems," *IEEE Trans. Automatic Control*, vol. 33, no. 7, pp. 626-639, July 1988.

[16] R. Koymans, R. K. Shyamasundar, W. P. de Roever, R. Gerth, and S. Arun-Kumar, *Compositional Semantics for Real-time Distributed Computing*. Springer-Verlag, June 1985.

[17] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Programming Languages Syst.*, vol. 5, pp. 190-222, Apr. 1983.

[18] A. H. Levis, "Challenges to control: A collective view," *IEEE Trans. Automatic Control*, vol. AC-32, Apr. 1987.

[19] Z. Manna and A. Pnueli, "How to cook a temporal proof system for your pet language," in *Proc. Symp. Principles of Programming Languages*, Austin, TX, Jan. 1983, pp. 141-154.

[20] Z. Manna and A. Pnueli, "The anchored version of the temporal framework," in J. W. de Bakker, W. P. de Roever, and G. Rozenburg, Eds., *Models of Concurrency: Linear, Branching and Partial Orders*. Springer-Verlag, 1989.

[21] Z. Manna and A. Pnueli, "Verification of concurrent programs, part 1: the temporal framework," Department of Computer Science, Stanford University, June 1981, Tech. Rep. STAN-CS-081-836.

[22] Z. Manna and A. Pnueli, "Verification of concurrent programs: a temporal proof system," Dept. of Computer Science, Stanford University, June 1983, Tech. Rep.

[23] Z. Manna and P. Wolper, "Synthesis of communicating processes from temporal logic specifications," *ACM Trans. Programming Languages Syst.*, vol. 6, pp. 68-93, Jan. 1984.

[24] R. Milner, *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[25] B. Moszkowski, "A temporal logic for multilevel reasoning about hardware," *Computer*, vol. 18, pp. 10-19, Feb. 1985.

[26] K. T. Narayana and A. A. Aaby, "Specification of real-time systems in real-time temporal interval logic," in *Proc. Real-Time Systems Symp.*, Dec. 1988, pp. 86-95.

[27] J. S. Ostroff, "Automatic verification of timed transition systems," in E. M. Clarke, A. Pnueli, and J. Sifakis, Eds., *Proc. Workshop Automatic Verification Methods for Finite State Systems*. Springer-Verlag, 1989.

[28] J. S. Ostroff, "Real-time temporal logic decision procedures," in *Proc. 10th IEEE Real-Time Systems Symp.*, Santa Monica, CA, Dec. 1989, pp. 92-101.

[29] J. S. Ostroff, "Synthesis of controllers for real-time discrete event systems," in *Proc. 28th IEEE Conf. Decision and Control*, Tampa FL, Dec. 1989, pp. 138-144.

[30] J. S. Ostroff, *Temporal Logic for Real-Time Systems* (Advanced Software Development Series). Research Studies Press Limited (distributed by John Wiley and Sons), England, 1989.

[31] J. S. Ostroff, "Verification of finite state real-time distributed processes," in *Proc. 9th IEEE Int. Conf. Distributed Computing Syst.*, June 1989, pp. 207-216.

[32] J. S. Ostroff and W. M. Wonham, "A framework for real-time discrete event control," *IEEE Trans. Automatic Control*, Apr. 1990.

[33] D. L. Parnas, A. J. van Schouwen, and S. P. Kwan, "Evaluation standards for safety-critical software," Department of Computer Science, Queen's University, Tech. Rep. TR 88-220, May 1988.

[34] J. L. Peterson, *Petri Net Theory and the Modelling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[35] A. Pnueli, "The temporal logic of programs," in *Proc. 18th IEEE Annual Symp. Foundations of Computer Science.*, Providence, RI, Nov. 1977, pp. 46-57.

[36] A. Pnueli, "Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends," in J. de Bakker, W. P. de Roever, and G. Rozenburg, Eds., *Current Trends in Concurrency*. Springer-Verlag, 1986.

[37] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete-event proceses," *SIAM J. Control and Optimization*, vol. 25, pp. 206-230, Jan. 1987.

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

[39] W. P. De Roever, "Questions to Robin Milner—A responder's commentary," in H. J. Kugler, Ed., *Information Processing 86*. Elsevier Science Publishers B.V. (North Holland), 1986, pp. 515-518.

[40] R. L. Schwartz and P. M. Melliar-Smith, "From state machines to temporal logic: Specification methods for protocol standards," *IEEE Trans. Communications*, vol. COM-30, Dec. 1982.

[41] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next generation systems," *Computer*, vol. 21, pp. 10-19, Oct. 1988.

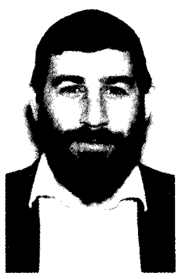
[42] J. G. Thistle and W. M. Wonham, "Control problems in a temporal logic framework," *Int. Control*, vol. 44, 1986.

[43] G. von Bochmann, *Concepts for Distributed System Design*. Springer-Verlag, 1983.

[44] N. Wirth, "Towards a discipline of real-time programming," *Communications ACM*, vol. 20, Aug. 1977.

[45] W. M. Wonham, "Some remarks on control and computer science," *IEEE Control Syst. Mag.*, vol. 7, pp. 9-10, Apr. 1987.

[46] P. Zave, "An operational approach to requirements specification for embedded systems," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 250-269, May 1982.



Jonathan S. Ostroff received the B.Sc. degree from the University of the Witwatersrand in Johannesburg in 1976, and the M.A.Sc. and Ph.D. degrees in electrical engineering from the University of Toronto, Canada, in 1979 and 1987 respectively. From 1979 to 1981 he worked as a systems analyst for Imperial Oil Limited designing process control software. He has been an Assistant Professor in the Department of Computer Science, York University, Canada, since 1986, where he is also a member of the ISTS (Institute for Space and Terrestrial Science) Laboratory. He is author of the monograph *Temporal Logic for Real-Time Systems* (Research Studies Press and John Wiley and Sons), and is a member of IEEE and ACM.