7-1991

Constructing Real-Time Systems from Temporal I/O Automata

J. F. Peters III

S. Ramanna

SU-CIS-91-22

Constructing Real-Time Systems from Temporal I/O Automata

J. F. Peters III and S. Ramanna

July 1991

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, New York 13244-4100

Constructing Real-Time Systems from Temporal I/O Automata*

J. F. Peters III and S. Ramanna

Syracuse University
School of Computer & Information Science
4-116 CST, Syracuse, NY 13244-4100 USA

Abstract

A new class of communicating automata called Temporal Input/Output Automata ($TA_{i/o}$s) is introduced. A $TA_{i/o}$ is a predicate automaton used to specify real-time systems. The specification provided by a $TA_{i/o}$ includes state predicates with proof expressions and abstract program syntax as attributes. An abstract program is extracted during a constructive proof of the specification using the proof expressions. A $TA_{i/o}$ specification also includes hard, real-time constraints on program behavior. The predictability of deterministic, temporally complete $TA_{i/o}$ is investigated. The formulation of real-time system transductions and transduction rules for $TA_{i/o}$s in explicit clock temporal logic is given. An illustration of the use of $TA_{i/o}$s in specifying light-controlled vehicles is presented. To illustrate the methodology in constructive reasoning about a $TA_{i/o}$, a proof which derives a partial abstract program is given.

Index Terms--Communicating automata, program correctness, program specification, real-time systems, temporal logic.

1. Introduction

Finite state automata are considered the fundamental descriptive tools of computing [Con 80]. The behavior of agents in a system has been modelled with finite-state automata [Alu 90, Hal 89, Hen 91, Lav 90, Lyn 87, Man 89, Ost 89, Ost 90, Pet 90a, Pet 90b, Pet 91a, Pet 91b, Kla 91]. An *agent* is that part of a system which has its own identity, and its own externally observable behavior [Mil 89, Pet 91a]. The *behavior* of an agent is defined to be an infinite sequence of events. An *event* is an externally observable, discrete occurrence. By *discrete event*, we mean an event separable observationally from other events. Examples of events are actions of agents, communications between agents, the observable parts of agent states (length and

* Research supported in part by the School of Computer and Information Science, Syracuse University, Syracuse, NY 13244-4100 USA and by the Research & Development Laboratories, Culver City, CA 90230-6608 USA. Submitted for journal publication.

contents of queues, variables, constants, and so on). Automata can be represented as finite, directed, labelled graphs. The nodes of such graphs represent agent states; the arcs, transitions between states. The specification of the various behaviors of an agent can be given by "annotating" the nodes and arcs of an automaton with predicates. Each automaton node is annotated with a predicate that specifies an activity associated with the state; each arc is inscribed with a predicate identifying an enabling condition for a transition to the next state. Such automata are termed predicate automata [Man 89, Alp 86]. The aim of this paper is to introduce a special class of predicate automata called temporal input/output automata ($TA_{i/o}$s), which can be used to model the time-constrained behavior of real-time systems. In such automata, state predicates can reference an external clock in specifying timing constraints on the behavior of an agent. The language accepted by a $TA_{i/o}$ corresponds to the set of behaviors of an agent which satisfy the specification provided by the predicates on the nodes and arcs of the $TA_{i/o}$.

A $TA_{i/o}$ is used to describe a real-time, computational task independent of the program which carries out the task. Remarkably, there is a connection between $TA_{i/o}$s and the very first conception of finite automata used by McCullock and Pitts to model the behavior of neural nets [McC 43]. That is, McCullock-Pitts neural nets and TAi/os rely on predicates with time parameters to describe process behavior. $TA_{i/o}$s also have affinity with the extended program flowcharts used in PICA [Tor 90] (i.e., both rely on the use of assertion nodes). Predicate I/O automata were introduced in [Pet90a]. A $TA_{i/o}$ is a predicate input/output automaton with a provision for specifying hard, real-time constraints. The relationship between a specified action and a program is expressed with an attributed form of node predicates. The reasoning about a specification embodied in a $TA_{i/o}$ provides a constructive proof that the specification satisfies some property. In this context, the term *property* is an assertion about a specified sequence of events in the behavior of a program. Proofs are regarded as expressions which denote evidence [Con 89]. In other words, these proof expressions provide a basis (evidence) for reasoning about the correctness of a *specified* computation. A proof is termed *constructive* when the evidence denoted by it can be computed from it. In the case of a $TA_{i/o}$, the description of a computation is made possible by annotating the states of the automaton with proof expressions similar to those found in [Con 89]. As in Nuprl [Cons 84, Cons 86, Mur90, Mur 91], the proof of an assertion produces some object either implicitly or explicitly. The object produced by a constructive proof of a specification provided by a $TA_{i/o}$ is a program.

The context for this research is given in Section 2. In Section 3, a formal definition of $TA_{i/o}$s is presented. Section 4 provides an introduction to a subset of real-time temporal logic called TL_{rt} as well as the properties of various members of the class of temporal i/o automata. A specification of a light-controlled vehicle in the TL_{rt}/ $TA_{i/o}$ framework is given in Section 5. The correctness issues relative to $TA_{i/o}$s and a sample constructive proof of a specification are given in Section 6.

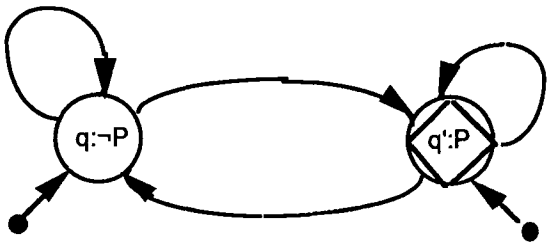
2. Modeling Real-Time Program Behavior with Automata

In the context of real-time systems, the term *modelling* refers to a precise behavioral description of the critical features of a system [Ost 89]. For example, some of the critical features of a controller of a real-time system are synchronization (rendezvous), concurrency (concurrent behaviors of communicating processes), responsiveness (behavior which adheres to timing constraints), determinism (behavioral transitions which satisfy enabling conditions), and non-determinism (interleaving of observed behaviors of concurrent processes). The behavior of a real-time system is constrained by what are known as hard, real-time constraints. A *hard, real-time constraint* specifies that an action by a system agent must be performed within a fixed number of time units. For example, a system agent must respond to input from another agent within 10 milliseconds. To model behaviors with infinite length in the context of real-time systems, it is common to consider finite state automata which accept infinite words. These automata are variations of what are known as Büchi automata.

2.1 Büchi Automata

Büchi Automata (BAs) are finite-state automata which accept infinite words [Büc 62]. A Büchi automaton (Σ, Q, Q_0, R, E) is a finite state machine with an input alphabet Σ, finite set of states Q, start states $Q_0 \subseteq Q$, recurrent states $R \subseteq Q$, and edges $E \subseteq Q \times \Sigma \times Q$. A *recurrent state* is an accepting state, which is visited infinitely many times during a run of a BA. Various variations of Büchi automata have been used to model the behavior of systems [Alp86, Man 89, Ost 89, Pet 90a, Pet 91, Kla 91]. A common feature found in all of these variations of Büchi automata is the presence of recurrent states. For example, Manna and Pnueli [Man 89] introduce $\forall$-automata. A $\forall$-automata is a predicate automaton which accepts inputs from a program computation

of infinite length. Formally, a ∀-automaton is a tuple (Q, C, E) with states Q = {recurrent states} U {stable states} U {start states) U {other states}, entry conditions E (each state q has an entry condition which must be satisfied before an automaton can start its activity in q), and transitions conditions C. The elements of C are predicates of the form c(q, q'). A transition from an automaton state q to a new state q' can occur when a transition condition c(q, q') is satisfied in state q. In other words, the sets E and C consist of first order predicates used to prescribe conditions which must be satisfied during an accepting run of a ∀-automaton. These automata are useful in specifying temporal properties of programs such as "infinitely often property P holds" (symbolized by □ ◇ P and represented graphically as shown in Figure 1). The automaton in Figure 1 is non-deterministic and has two start states (q and q').



Legend:

symbolizes a recurrent state

is an entry edge with default entry condition True.

Figure 1. ∀-automaton for □ ◇ P

The advantage to ∀-automata is that they combine visualization of process behavior with reasoning (via entry and transition predicates) about process behavior. Their disadvantage is that there is no provision for quantitative reasoning about hard, real-time constraints on process behavior.

2.2 Timed Automata

Recently there has been an effort to associate the ticks of a real-time clock with the events in a process behavior modelled by an automaton [Mer 91, Hen 91, Alu 90, Lav 90]. Except for a provision for input/output channels between composed automata found in [Mer 91], the timed Büchi automata (TBAs) introduced by Alur and Dill are closest to the temporal i/o automata introduced in this article. A TBA is defined as a 5-tuple (Σ, Q, Q_0, Clocks, E) with input alphabet Σ, states Q (as in Büchi automata, these include recurrent states $R \subseteq Q$), start states $Q_0 \subseteq Q$, a finite set of real-valued clocks, and a set of transitions E, where E is given by $E \subseteq Q \times \Sigma \times Q \times 2^{Clocks} \times \Phi(Clocks)$. A TBA accepts both finite and infinite timed sequence of events (called timed traces), which are observable during the run of a process modelled by a TBA. As in [Mer91], each event in a timed trace is associated with a non-negative real number, which is a reading of an external clock at the time of the occurrence of the event in the trace. This allows for an unbounded number of environment events (reception of a value by another automaton, for example) between any two events of a system modelled by a TBA.

An edge $(q, \sigma, \lambda, \delta, q')$ in a TBA represents a transition from state q to q' with input symbol σ (λ gives the clocks to be reset with this transition), and δ gives the enabling condition. In other words, edges are inscribed with predicates (timing constraints and possibly reset(x)). The reset(x) predicate asserts that clock x is reset to zero. Figure 2 gives an example of a TBA.

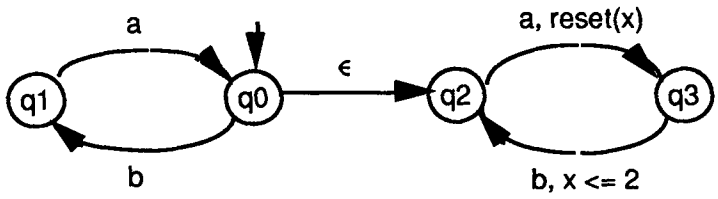


Figure 2. Timed Büchi Automaton referencing external clock x

The predicate reset(x) asserts that clock x is reset in the transition from q_2 to q_3. The timing constraint x <= 2 asserts that the transition from q_3 to q_2 can only occur if the elapsed time is within 2 ticks of clock x. In effect, TBAs are predicate automata resembling property recognizers [Alp 86], where edges are inscribed with transition conditions (predicates without references to external clocks). The drawback of TBAs is

the lack of data variables as found in the Extended State Machines (ESMs) in Ostroff [Ost 89] and Real-time Transition Systems (RTSs) in Henzinger et al. [Hen 91]. Included in the data variables of an ESM, for example, is a rigid clock variable T (this variable saves a reading of an external clock and retains its value despite state changes). This eliminates the need for the reset(x) predicate, which must be part of a transition whenever an external clock is reset. The use of a clock variable rather than the reset(x) predicate, provides a more abstract specification of process behavior, because the role of T is hidden in a specification. The end result is a simpler specification of timing constraints, which are easier to implement in a programming language.

3 Temporal I/O Automata

To model the timed-behavior of communicating processes in real-time systems, we introduce a class of predicate automata called Temporal I/O Automata ($TA_{i/o}$). The timed actions associated with a state are specified with state predicates; arcs of TAi/os are inscribed with enabling conditions for transitions. These are communicating automata. When $TA_{i/o}$s are composed, message-passing between the automata is made possible by the presence of hidden input/output channels. Each $TA_{i/o}$ has input/output channel variables used in sending and receiving messages over i/o channels. Input/output automata ($A_{i/o}$s) were introduced by Lynch and Tuttle [Lyn 88], and extended to include timing constraints by Merritt et al. [Mer 91]. The language accepted by a $TA_{i/o}$ is the set of the timed behaviors of an agent. Acceptance of the behaviors of an agent by a $TA_{i/o}$ ensures that each sequence of events in an agent behavior satisfies a property specified by the automaton. A $TA_{i/o}$ is defined as follows:

$TA_{i/o}$ = (Q, q_o, D, P, Clock, N, E)

where

Q = { start state q_o } U { recurrent states } U { other states }

D = { I (input channel variable), O (output channel variable) }
U { state variables: time, ...} U { rigid variables: T, ...}

P = set of first order predicates

Clock = external clock

N = set of state predicates, where $N \subseteq Q \times P \times \Phi(\text{Clock}) \times I \times O$

E = set of enabling conditions, where $E \subseteq Q \times P \times Q$

A state predicate prescribes a (possibly timed) action associated with an automaton state. As in [Lyn 88, Mer 91, Pet 90a], there are four types of actions which can be predicated of a state of an automaton A; these actions are described informally as follows:

int(A) = local action.
out(A) = action A writes a value to an output channel.
in(A) = action A reads a value from an input channel.
io(A) = action A reads a value from an input channel, and writes a value to an output channel.

In keeping with Ostroff's analysis [Ost 89], a distinction is made between actions and events. Actions lead to events and each event leads to the transformation of a state to a new state. Let int, in, out, io be the names of actions; E, the name of an event; Q, a set of $TA_{i/o}$ states; and let I and O be input and output channels, respectively. The distinction between actions and events is defined formally as follows:

Actions	Events
int: $Q \longrightarrow E$	$E: Q \times I \times O \longrightarrow Q$
in : $Q \times I \longrightarrow E$	
out:$Q \times O \longrightarrow E$	
io :$Q \times I \times O \longrightarrow E$	

Examples of events are timeout (maps a state to a new state when an action times out), reception of a message msg from a source s (written as s?msg in CSP), sending a msg to a destination d (written as d!msg in CSP), the tick of an external clock, and so on. For implementation reasons, it is assumed that communication between $TA_{i/o}$s is synchronous. Further, unlike synchronous communication in CSP [Hoa85], $TA_{i/o}$s are unable to block inputs from other automata. An untimed io action terminates when a synchronization concludes. A system of communicating automata is formed by what is known as a composition. The result of a composition of $TA_{i/o}$s is a collection of communicating automata, which specifies the behavior of a system of communicating agents. Let A_i , A_j be $TA_{i/o}$s and let $A_i \,||\, A_j$ represent the composition of A_i and A_j, where

$A_i = ((Q, q_o, D, P, Clock, N, E)$ and $A_j = (Q', q_{o'}, D', P', Clock', N', E')$

Then composition of A_i and A_j is defined as follows:

$$A_i \,||\, A_j = (Q \cup Q' \cup Q_g, q_o, q_o', q_g, D \cup D' \cup G, P \cup P' \cup P_g, \{Clock, Clock', Clock_g\}, N \wedge N' \wedge N_g, E \wedge E' \wedge E_g)$$

where

G	= {sys. state variables: $time_g$,...} U {sys. rigid variables: T_g,...}
Q_g	$\subseteq G \times Q_i \times Q_j$ (system states)
q_g	= system start state (present with tightly coupled $TA_{i/o}$s)
P_g	= set of system predicates
$Clock_g$	= guardian clock process (gives the system time & acts as a synchronizer of local clocks)
N_g	$\subseteq G \times P_g \times Q_i \times Q_j$ (set of system state predicates)
E_g	$\subseteq G \times P_g \times E \times E'$ (set of enabling conditions for transitions between system states)

The set of system predicates is similar to proof expressions in [Con 89]. In some very real sense, the predicates on the nodes and arcs of either an individual $TA_{i/o}$ or on the nodes and arcs of a composition of $TA_{i/o}$s are part of a *deduction* about a behavior of a program. Their presence makes the proof of correctness of program behavior feasible and makes possible the extraction of the program which they prescribe. A visualization of a composition of automata is given in Figure 3. The notation in Figure 3 is explained as follows:

q^{α}	$= (q^{\alpha}{}_1, q^{\alpha}{}_2, q^{\alpha}{}_3, ..., q^{\alpha}{}_j, ...)$	--seq of $TA_{i/o}$ A^{α} states (α is a $TA_{i/o}$ index).
Q^{Φ}	$= (G, q^1, q^2, q^3, ..., q^{\beta}, ..., q^m)$	--system path for m $TA_{i/o}$s (Φ "phi" is a system path index).
$Q^{\Phi}{}_i$	$= (G, q^1{}_i, q^2{}_i, q^3{}_i, ..., q^{\beta}{}_i, ..., q^m{}_i)$	--i th system state
$Q^{\Phi}{}_i : p(i)$	= system state predicate	--annotates ith system state

e(i) represents an enabling condition for a system state transition.

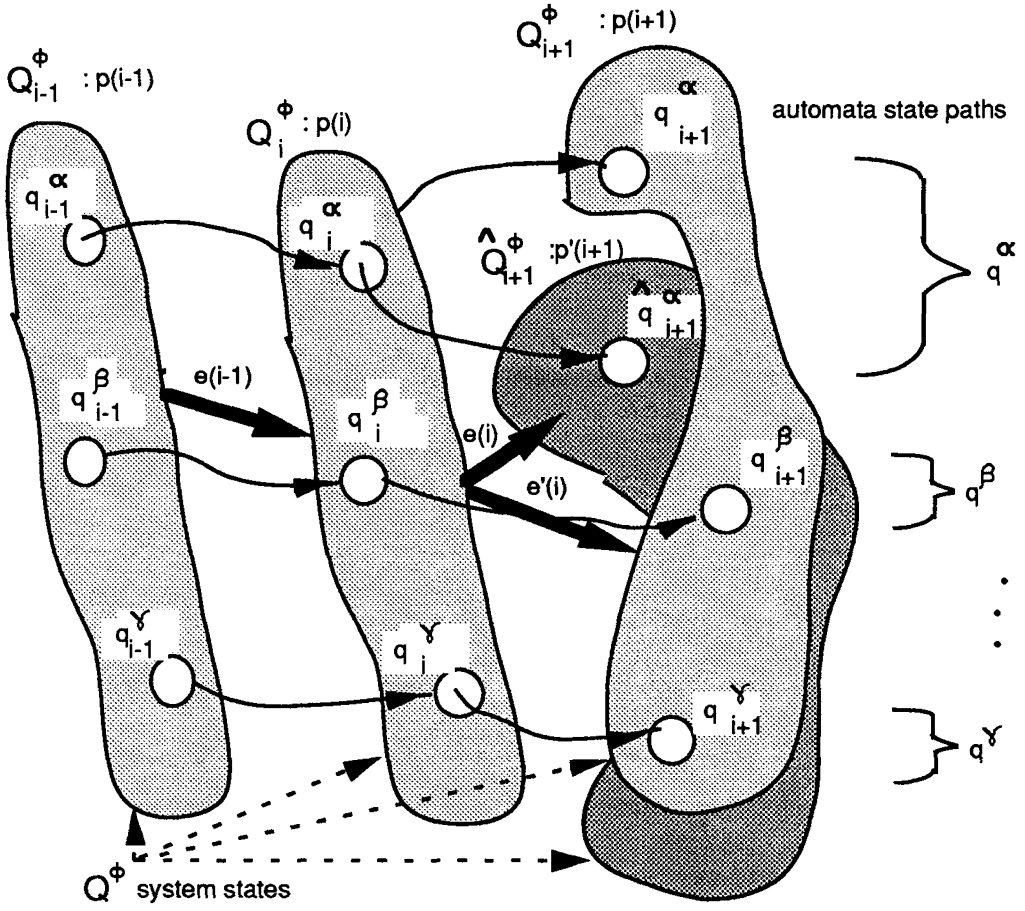


Figure 3. Abstract View of Composition of Automata

In a composition of automata, a guardian $Clock_g$ is present; it gives the system time, and guarantees that local clocks are synchronized with $Clock_g$. The actual synchronization of the local clocks in the composition is hidden, and is not part of the specification provided by $A_i \parallel A_j$. Synchronization of local clocks with respect to the global clock becomes a chief concern whenever a system state has a timing constraint. The set G is a set of global data variables containing rigid variables such as T_g (to store a reading of $Clock_g$), and state variables such as $time_g$ (captures the value of $Clock_g$ in the current state).

3.1 Clock Variable and Timed Behaviors

Timing constraints of a $TA_{i/o}$ reference ticks of an external clock (denoted by variable Clock). The rigid variable T records the Clock value, and retains its value across state changes of a $TA_{i/o}$. We assume that the value of T can be changed when needed (this is analogous to resetting the clock in a TBA [Alu91]). The flexible variable *time* gives the value of Clock in the current state. Clock readings are non-negative, real numbers. Each time an event occurs, a reading of Clock is associated with that event. That is, each event e is a conceptualized as a pair (e, time). As a result, a timed sequence of events β in the behavior of an agent modelled by a $TA_{i/o}$ has a trace of the form:

$$\beta = (e_o, time_o), (e_1, time_1), ...(e_i, time_i), ...$$

Let R^+ denote the non-negative reals; Nats, the natural numbers 0, 1, ... In addition, let $time_i$, $time_j$ belong to β. Then, as in [Alu 91, Pet 90a], a timed trace β has the following properties:

Zero-time in start state:	$time_o = 0$ in $(e_o, time_o)$
Strict Monotonicity:	$\forall$ i, j $\in$ Nats: $time_i < time_j$ for i < j
Unboundedness:	$\forall$ time $\in R^+$, $\exists$ i $\in$ Nats: time < $time_i$

3.2 Semantics of Delay

Responsiveness of a system is measured in terms of actual values of delays. The duration predicate delay(k) asserts that the external clock is allowed to run for k ticks before a timeout occurs. Delay(k) can be used to specify a lower bound on the number of ticks before an action is performed; delay(k) can also be used to specify an upper bound on the duration of an action. In other words, we can use delay(k) to express the fact that an action is enabled after a particular time (lower bound) or than an action is performed within a specified time limit (upper bound).

3.2.1 Lower Time Bound

We can express a lower bound on the number of ticks before a system action begins. If we let ACT be the action to be performed in state q. We can express the fact that we let the external clock run for k ticks before performing ACT by writing informally "delay(k) before ACT." To see this, let T record the time in state q. Assume action ACT is performed in state q. Written by itself, "ACT" is shorthand for the assertion "the action ACT is performed." Let sat(q | (q'), P) mean that predicate P is satisfied in state q of the state sequence (q, q'), and sat(q', Q) mean that predicate Q is satisfied in state q'. The double turnstile ⊨ reads "forces" or "satisfies." Then satisfaction of "delay(k) before ACT" over a state sequence (q, q') is expressed in Prolog form as follows:

```
sat(q | (q'), delay(k) before ACT) :-
                                    q  ⊨ delay(k) and T <= time < T + k,
                                    q' ⊨ ACT and time = T + k.
```

This says that the duration predicate is satisfied in state q and k ticks later the predicate ACT is satisfied in state q' . The idea of using delay(k) to specify a lower bound on when an action can be performed, is expressed graphically in Figure 4.

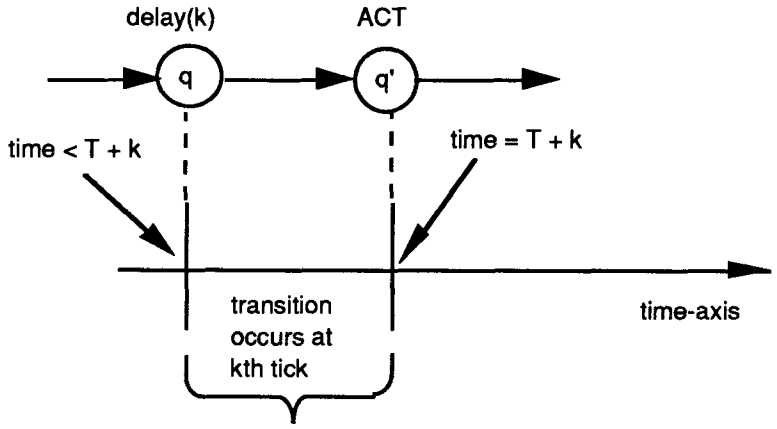


Figure 4. Lower bound on when a system action begins.

3.2.2 Upper Bound on a System Action

We can also express an upper bound on the number of ticks during a system action using delay(k). This is expressed rather simply by writing "ACT; delay(k)," which asserts that ACT cannot be continuously enabled for more than k ticks of the external clock. The predicate *timeout* (see Figure 5) is an enabling condition, which evaluates to true at the kth tick of the clock (i.e., an action which must be performed within k ticks times out, and a transition to the next state occurs). The meaning of this upper bound constraint can be explained concisely by using the satisfaction clause sat(q, P). Then the upper bound timing constraint can be defined as follows:

```
sat(q, ACT; delay(k))   :-   q ⊨ ACT,
                             q ⊨ time < T + k;          /* ; reads "or" */
                             q ⊨ time = T + k and timeout.
```

A graphical interpretation of the upper bound constraint on the duration of a system action is given in Figure 5.

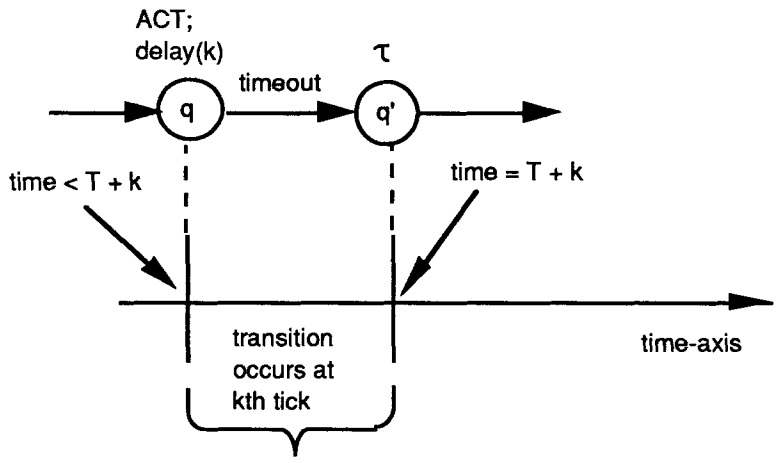


Figure 5. Upper bound on the duration of a system action.

4. Timed-Behavior Expressed with Temporal Logic

The behavior of a real-time system can be specified with Real-Time Temporal Logic (RTTL) given in [Ost 89, Har 90, Hen 91]. When temporal logic is applied to the study of processes, the formulas of temporal logic are interpreted as predicates over sequences of process states [Alp 86]. Each state occurs at some instant in time in which the values of process variables can be inspected. During a succession of states, changing values of state variables may entail changing truth values of predicates about state variables. Hence, it is appropriate to use some form of temporal logic to describe process behavior. Temporal logic allows the specification of a temporal ordering of actions of a system agent. Temporal formulas can be used to enumerate state transitions (transformations of one state into a new state) in a behavior as well as the order in which transitions are made.

RTTL provides a concise means of prescribing a property of a behavior represented by a temporal I/O automaton; such prescriptions are assertional. This form of temporal logic is essentially the same as the original temporal logic introduced by Manna and Pnueli [Man 81, Man 83] with the addition of data variables such as T (for timing constraints) suggested by [Hen 90, Har 90]. Except for some additional derived temporal operators taken from [Pet 90a], the temporal logic used in this article is the same as RTTL. For simplicity, we limit the presentation of RTTL to a discussion of the U (until) and temporal operators derived from U. We also introduce the derived temporal operators before, $\diamond^{\omega}$ (infinitely often), and seq(p_1, p_2, p_3,..., p_n) (a temporally quantified sequence of state predicates where p_1 holds before p_2, which holds before p_3, ..., before p_n).

For the subset of RTTL (named TL_{rt}) we have chosen, the temporal language TL_{rt} is defined as follows:

Alphabet

- A denumerable set of variables: x, y, ...
- A denumerable set of n-ary functions: f, g, ...
- A denumerable set of n-ary predicate symbols: p, q, ...
- symbols $\neg$, or, $\forall$, (,), U

Well-formed formulas of TL_{rt} have the following syntax:

- Every atomic formula is a formula.

- If x is a variable and A is formula, then $\forall$ x A is a formula.
- If A and B are formulas, then $\neg$ A, (A or B), (A U B)

  are formulas.

Semantics of Temporal Operators. The $\neg$ (not), or, and $\forall$ (all) symbols have the usual semantics. In addition, the implication symbol $\Longrightarrow$ (i.e., p $\Longrightarrow$ q $\equiv \neg$ p or q) is used. In defining the following semantics, the notation

$(q_0,...,q_x) \models p$ for x >= 0

asserts that each of the states in the sequence $(q_0,...,q_x)$ satisfy predicate p. In what follows, let q_0 represent the current state in a behavior. Let p, q be first-order predicates. The semantics of U as well as the operators derived from U are as follows:

p U q	$\equiv \exists$ k, x: 0 <= x <= k: $(q_0,...,q_x) \models$ p and $q_k \models$ q
p before q	$\equiv \exists$ k: 1 <= k: $q_0 \models$ p and $(q_1,...,q_k) \models$ p U q
$\Diamond$p	$\equiv$ true U p
$q_k \models$ seq(p)	$\equiv q_k \models$ p
seq(p_1, (seq(p_2, ...,p_n)))	$\equiv$ p1 before seq(p_2, (seq(p_3,...,p_n))
$\Diamond^{\omega}$ p	$\equiv$ seq(p, $\Diamond^{\omega}$ p)

The predicate 'p U q' asserts that the predicate q eventually holds (either in the current or in some future state) and that the predicate p holds in the current state and in each of the states *until* the state when q holds. By contrast, 'p before q' asserts that p is guaranteed to hold initially and sometime later q will hold. For this reason, *before* is called a precedence operator [Krö 85]. These powerful temporal operators provide the basis for the semantics of the remaining operators in the above list.

Notation. Let ACT be the name of an action associated with a state q in a $TA_{i/o}$. Let $x^{>}$ represent a parameter x (of Xtype) whose value is to be written to an output channel. Let $y_{<}$ be a parameter y (of Ytype) whose value is to be read from an input channel. Then the predicate

ACT($x^{>}$: Xtype, $y_{<}$: Ytype) asserts action ACT writes x to an output channel,

and reads y from an input channel.

In the case where ACT is parameterless, we write ACT_{io}.

The temporal assertion ◇ p says there will be some state either now or in the future in which the predicate p evaluates to true. For example, let process be the name of an internal action for an agent which receives values for $x_<$, $\theta_<$ as input, and computes values for $x'^>$, and $\theta'^>$, as output. Then

◇ process($x_<$, $\theta_<$, $x'^>$, $\theta'^>$,)

asserts that eventually the observed values of θ and x will be processed to obtain the predicted values of θ' and x'. Notice that for a named action *ACT*, if we write ◇ ACT, this is a shorthand way of writing "eventually perform ACT."

4.1 Temporal Semigroups

It is possible to define a semigroup relative to the *before* temporal operator. This will allow us to express assertions with seq more concisely. In conventional terms, a semigroup is defined as follows.

> Definition 4.1. Let T be a non-empty set, and let α be an operation on T. A *semigroup* is a pair (T, α) such that for all x, y, z in T, the operation α is associative, i.e., x α (y α z) = (x α y) α z.

The temporal operators in TL_{rt} belong to what is known as the future fragment. That is, temporal predicates written with TL_{rt} always refer either to the present state or some future state. Due to the semantics of *before* and *until*, parenthesizing a precedence- or an until-assertion does not change the temporal evaluation of the formula. As a result, parentheses only provide syntactic sugar (making some formulas easier to read). In this restrictive sense, we can define a temporal semigroup as follows.

> Definition 4.2 Let P be a set of predicates and let τ be a temporal operator. A *temporal semigroup* is a pair (P, τ) such that for all x, y, z in T, the operation τ is associative, i.e., x τ (y τ z) = (x τ y) τ z.

In the case where (P, τ) is a temporal semigroup, then we can remove the parentheses and write the expression x τ y τ z. For example, we can write x before y before z as a result of Proposition 4.1.

Proposition 4.1 Let P a set of predicates. Then (P, before) is a temporal semigroup.

Proof: Let p_1, p_2, p_3 be predicates in P, and let term = (p_2 before p_3). Further assume q_o, ..., q_x, ..., q_k are states with 0 <= x <= k over which we evaluate predicates p_1, p_2, and p_3. Then

0	p_1 before (p_2 before p_3)	assumed
1	p_1 before term	by def.
2	$\exists$ k: 1 <= k: $q_o \models p_1$ and $(q_1,...,q_k) \models p_1$ U term	by def. of before
3	$q_o,...,q_x \models p_1$ and $(q_{x+1},...,q_y,...,q_k) \models$ term, x >= 0	fr 2, WLOG
4	$(q_{x+1},...,q_y,...,q_k) \models p_2$ before p_3	fr 3, def. term
5	$q_{x+1} \models p_2$ and $(q_{x+2},...,q_y,...,q_k) \models p_2$ U p_3	fr 4, def. before
6	$q_{x+1},...,q_y \models p_2$ and $(q_{y+1},...,q_k) \models$ p3	fr 5, WLOG
7	(p_1 before p_2) before p_3	fr 3, 6

∎

Since the seq operator is defined in terms of *before*, predicates like seq(p_1, (seq(p_2,...,p_n))) can also be rewritten as seq(seq(p_1,...,p_{n-1}), p_n). That is, this is another way of writing p_1 *before* (seq(p_2, ..., p_n)). By continuing this expansion of the seq formula, the seq operator is eliminated as in

p_1 before (seq(p_2, ..., p_n))
$\equiv$ p_1 before (p_2 before (seq(p_3,...,p_n))) ...
$\equiv$ p_1 before (p_2 before (p_3 before (...(p_{n-2} before (p_{n-1} before p_n)...)

By repeated application of Prop. 4.1, we can rewrite this assertion as

((...(p_1 before p_2) before p_3) before p_4)...)before p_n

This gives us the following result.

Proposition 4.2 (P, seq) is a temporal semigroup.

Propositions 4.1 and 4.2 allow us to simplify the specification of a temporally ordered sequences of predicates. This is reflected in the next proposition.

Proposition 4.3 Let $p_1, p_2, ..., p_n$ be predicates. Then $seq(p_1, (seq(p_2, ..., p_n)))$ can be written as $seq(p_1, p_2, ..., p_n)$.
Proof: Immediate from Propositions 4.1 and 4.2.

Next, we investigate the use of TL_{rt} in specifying the necessary conditions for a transformation of a particular state into a new state.

4.2 Transductions and Transduction Rules

Transduction rules pinpoint the basis for transitions between states in the observed behavior of a system. They are useful in formulating timing as well as other consistency constraints imposed on system behavior. In the design of a real-time system, we are interested in formulating state-transformational control rules to guarantee consistency in a system behavior. Rather than speak in terms of entire state sequences in a timed-behavior (the macro view), transduction rules provide a refined granularity in the prescription of transitions between states *within* a behavior (the micro view). A *transduction rule* is a satisfaction rule that specifies under what conditions a transformation from one state to another should be made. Let e_{cond} be an enabling condition for the transition between states q and q'. Further, let $Tr_{q,q'}$ be a transduction rule with respect to states q and q' with state predicates P;delay(k) and P', respectively. $Tr_{q,q'}$ is defined as follows:

$$Tr_{q,q'} = sat(q \mid (q'), P; delay(k) \text{ and } e_{cond})$$

A *transduction* defines the transformation of state q into state q' in terms of state predicates P and P', duration of state activity (delay(k)), and possible input from and output to I/O channels by the operation specified by the state predicate. A transduction $Td_{q,q'}$ is defined as follows:

$Td_{q,q'}$ = seq(P; delay(k), P')

A transduction $Td_{q,q'}$ = seq(P; delay(k), P') asserts that "predicate P is satisfied in state q *before* predicate P' is satisfied in state q'". On the one hand, a *transduction rule* is a first-order predicate, which specifies under what conditions a transduction (i.e., transformation of a state into a new state) is made. On the other hand, a transduction $Td_{q,q'}$ is a temporal ordering of state predicates with a tacit ordering of events. In the case where a $TA_{i/o}$ is deterministic, there is a strict relationship between $Tr_{q,q'}$s and $Td_{q,q'}$ s.

4.3 Temporally Complete I/O Automata

It is important for control engineers designing a real-time system to know under what conditions the behavior of a system is predictable. For this reason, the completeness of a temporal I/O automaton with respect to timing constraints is of interest. In terms of timed behavior, there is a need to know that the responsiveness of a system to input from the environment is within some maximum time (referred to as MAXT in [Pus 90]).

Definition 4.3 A temporal I/O automaton is complete if

i) every state has a timing constraint (a lower bound as explained earlier and a finite upper bound specified by delay(k)).

ii) for every state q, there is a transduction rule Trq,q' which is valid.

Let $cTA_{i/o}$ be a temporally complete I/O automaton with arbitrary state q annotated with predicate P. By definition, q has a timing constraint. WLOG, assume that P is of the form ACT; delay(k). If the action specified by ACT times out in k ticks of the clock, then by definition (4.3 (ii)) there must be a transition from q to some state q' which is enabled as a result of the timeout. That is, there must be a transduction rule in $cTA_{i/o}$ of the form sat(q | (q'), ACT; delay and timeout). As a result, we have the following propositions.

Proposition 4.4. Every state in a temporally complete I/O automaton has an exit edge which is inscribed with a timeout enabling condition.

Proposition 4.5 Associated with every state q in a temporally complete I/O automaton, there is a transduction rule of the form sat(q | (q'), ACT; delay(k) and timeout).

The completion of a timed action ACT in a state q means that either ACT is performed within a specified time or there is a timeout. A timed i/o action completes either when it terminates or times out. The completion of a timed action does not preclude a timeout. That is,

Definition 4.4 The *completion of a timed action* ACT in a state q means that sat(q, ACT; delay(k)) holds (i.e., a transition from state q to q' occurs).

By definition, a timed action specified by a node predicate leads to an event. Every event induces a transition to a new state in $cTA_{i/o}$, either as a result of a timeout or because the specified action has completed within a specified number of ticks of the external clock. This proves

Proposition 4.6 Given the assertion ACT; delay(k) on node q in $cTA_{i/o}$. The completion of a timed action implies Tdq, q'. That is, a transition from state q to q' occurs.

4.4 Deterministic, Temporal I/O Automata

A $TA_{i/o}$ is deterministic if mutual exclusion among transduction rules holds. This idea is stated formally in Def. 4.5.

Definition 4.5 Let q, q', q" be states in a $TA_{i/o}$ and let $e_1,...,e_i,...,e_j, ..., e_n$ the enabling condition on transitions leading from q to other states. Let $Tr_{q,q'}$ and $Tr_{q,q"}$ be transduction rules for enabling conditions e_i and e_j for $1 <= i, j <= n$, where $i <> j$, respectively. The transduction rules are *mutually exclusive* if $\neg(Tr_{q,q'}$ and $Tr_{q,q"})$ holds.

In the case where a temporally complete automaton is deterministic, we can state the relationship between transduction rules and transductions formally as follows:

Proposition 4.7. Let sat(q | (q'), P; delay(k) and e_{cond}) be the transduction rule for a transformation of state q to q' and let P' be the state predicate which labels the node q' of a deterministic $cTA_{i/o}$. Then

$$sat(q \mid (q'), P; delay(k) \text{ and } e_{cond}) \iff Td_{q,q'}\ (seq(P;\ delay(k), P'))$$

specifies the transduction from q to q'.

Proof (by construction).
($\Longrightarrow$) Let sat(q | (q'), P; delay(k) and e_{cond}) be a transduction rule which is satisfied in state q. Assume "P; delay(k)" labels state q and P' is the state predicate which labels q'. By definition of a transduction rule, "P; delay(k) and e_{cond} " holds in state q. Hence, the transformation from state q to q' can be made. This is another way of saying the node predicate "P; delay(k)" will be satisfied in state q within the time imposed by the timing constraint specified by delay(k). This also says the enabling condition e_{cond} also holds, which enables the transition from q to q'. In addition, since P' is the state predicate which labels q', by definition P' must be satisfied in state q'. That is, a predicate which labels a state is satisfied in that state. Since $TA_{i/o}$ is deterministic, the mutual exclusion property holds. In addition, since $TA_{i/o}$ is temporally complete, we know by Prop. 4.6 that $Td_{q,q'}$ (seq(P; delay(k), P') holds.
($\Longleftarrow$) Immediate from the definition of seq(P; delay(k), P').

∎

4.5 Predictability of Temporal I/O Automata.

The importance of temporally complete automata becomes apparent in the investigation of quantitative measures of predictability. That is, given a $cTA_{i/o}$, we can compute upper bounds (values of MAXT) on the response times of prescribed actions of the automaton. In the case where a $cTA_{i/o}$ is deterministic, the computation of MAXT for every action in a specified behavior is straightforward. To see this, we introduce the following notation.

Notation. Let $cTA_{i/o}$ be a temporally complete automaton. Let a_i be an action specified by the node predicate for state q_i of a $cTA_{i/o}$ and a timed action trace α of such an automaton be represented by

$$\alpha = (a_0;delay(k_0)), (a_1;delay(k_1)), ..., (a_i;delay(k_i)),...)$$

Also, let the maximum response time for the ith action a_i be $MAXT_{ai}$. In the case of a deterministic $cTA_{i/o}$, the following result is easy to prove:

Proposition 4.8. Every action in a deterministic, temporally complete $TA_{i/o}$ is part of a single timed trace.

To say that an action must be performed with k ticks of an external clock is somewhat ambiguous. In the case where an action ACT has a timing constraint given by delay(k), ACT times out if it takes k ticks to complete. Otherwise, if ACT does not time out, there is an upper bound on the duration of ACT which we call a local-MAXT for ACT to complete without a timeout, namely, k - 1 ticks. Relative to a timed trace α, we introduce the notion of global-MAXT with respect to the final action in α. These terms are defined as follows:

Definition 4.6. (local-MAXT) The *upper bound on the normal response time* for an action ACT with timing constraint delay(k) is k - 1. If ACT takes k ticks of the external clock to complete, then it times out.

Definition 4.7. (global-MAXT) The maximum time $MAXT_{ai}$ is the upper bound on the normal response time of TAi/o measured from a_0 to a_i in a timed action trace α (i.e, MAXT is the overall time for normal response).

A timing constraint delay(k) is considered a local-MAXT. The maximum time MAX_{ai} is considered a global maximum time (for an entire timed trace); this is analogous to the analysis of MAXT with respect to an entire program given in [Pus 90]. Let "ACT, delay(k_0)" be the node predicate on the start state for a $TA_{i/o}$. By the start state axiom, we know that the value of the clock variable has an initial value of zero. The completion

of ACT occurs within k_o - 1 ticks of the external clock (the initial value of the clock is counted as 1 tick). This proves

Proposition 4.9. local-MAXT$_{ao}$ = k_o-1 for a timed action in the start state.

This suggests a way to compute the value of MAXT in a cTA$_{i/o}$. Using mathematical induction, we can prove the following for a deterministic cTA$_{i/o}$:

Proposition 4.10. In a deterministic cTA$_{i/o}$, the maximum response time for the ith timed action a_i over a timed trace a_o to a_i is given by

$$\text{global-MAXT} = k_o + k_1 + ... + k_i - i$$

For a nondeterministic cTA$_{i/o}$, computation of global-MAXT is somewhat more difficult, since each action can be part of more than one timed trace. If we let sample_MAXT be the maximum response time of an action over a timed trace in a nondeterministic cTA$_{i/o}$, then

Proposition 4.11. In a non-deterministic cTA$_{i/o}$, the maximum response time for the ith timed action a_i over n sample, timed traces from a_o to a_i is given by

$$\text{global-MAXT} = \max(\text{sample\_MAXT}_0, ..., \text{sample\_MAXT}_n)$$

where sample_MAXT$_j$ is the maximum response time for a_j over the jth sample, timed trace from a_o to a_j.

4.6 Named Temporal I/O Automata

When automata are composed, it is important to have some means of identifying the automata in a composition. In the case where there are a limited number of automata (no more than 10) to be composed into a system, colors could be used to distinguish automata. This becomes important when we are specifying actions representing communications between automata. So, for example, an automaton with nodes "painted"

yellow would call an automaton with nodes painted green. Then if yellow sends green a message (msg), we can write

yellow: green ! msg --yellow $TA_{i/o}$ sends msg to green $TA_{i/o}$

With an arbitrary number of automata in a composition, we need to choose some naming scheme (machine id number, for example) to write a specification for communications. For this reason, we introduce named $TA_{i/o}$s. A named $TA_{i/o}$ is tuple (name, Q, q_o, D, P, Clock, N, E), where name is a unique form of identification of the automaton. This gives rise to following notation for named automata.

> Notation. Let mac and sun be the names of two $TA_{i/o}$s which have been composed and let ACT be an action belonging to sun which mac calls, then we write mac: sun.ACT. The prefix mac identifies mac as the caller. In the event that mac "sends" sun a value using $x^{>}$ and receives a value $y_{<}$, we write
>
> mac: sun.ACT($x^{>}$, $y_{<}$) --mac calls ACT in sun

When it is clear from the context who the caller is in a communication, we adopt the CSP convention and drop the caller-prefix. We illustrate these ideas with a specification of a real-time system.

5. Specification of a Light-Controlled Vehicle

In this section, we utilize TL_{rt} and named $TA_{i/o}$s to specify a control system for an autonomous vehicle which relies on what is known as reactive navigation to gain access to light-controlled intersections [Ark 90]. Reactive navigation is a form of robot control which consists of a stimulus-response relationship with the external world. The controller for the autonomous vehicle in Figure 6 consists of two parts: a reactive navigation unit for a Light-Controlled Vehicle (LCV) and a guard unit (traffic light-controller) for intersections used by LCVs.

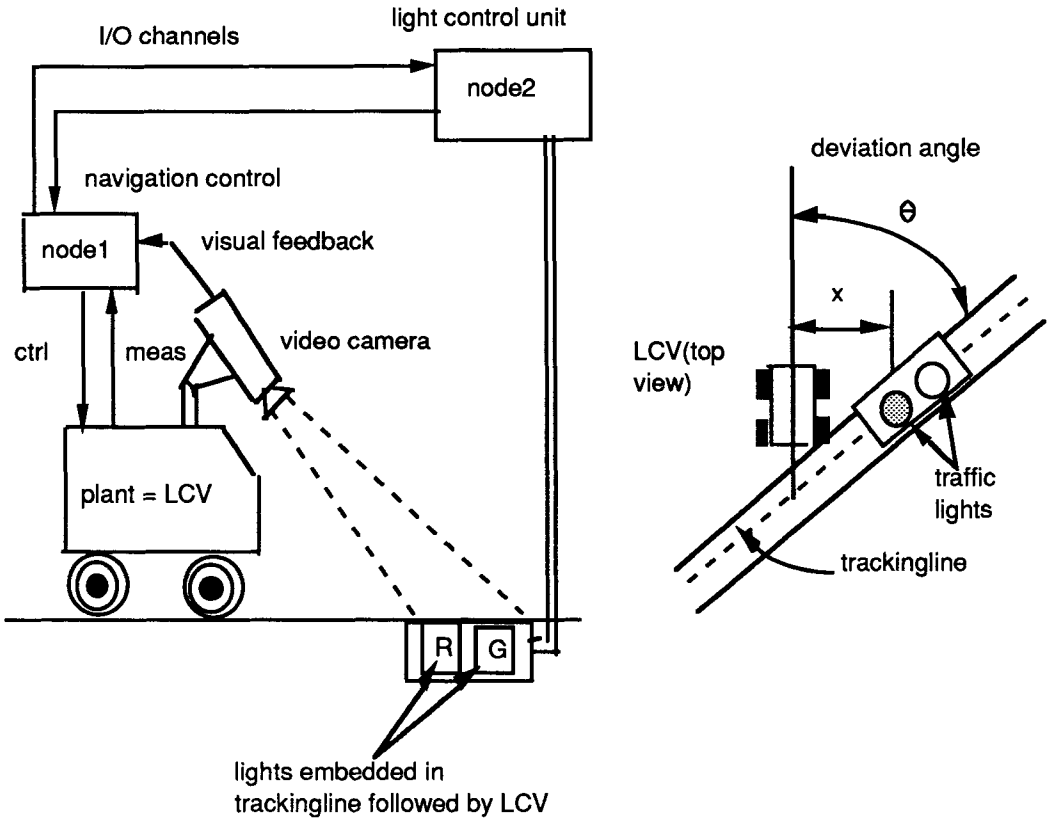


Figure 6. Light-Controlled Vehicle

For simplicity, the LCV in Figure 6 is modelled as an enhanced form of the mobile robot described in [Mar 90]. The LCV will use its camera to detect lights embedded in the path marked by the tracking tape. In addition to responding to observed deviation angles and x-distances, the LCV will also respond to traffic lights when they are detected in the sequence of images from its camera. The nodes in Figure 6 represent loosely coupled computers which communicate via a local area network. In summary, the real-time system in Figure 6 consists of the following components:

```
controller= LCV navigation control || Trafficlight control unit
plant     = LCV mobile unit || Trafficlights
rts       = controller || plant
```

5.1 Description of LCV Controller Behavior.

The LCV processes camera images which include traffic lights. The AV (autonomous vehicle) in Figure 6 is a mobile robot which relies on visual images captured by an on-board video camera to steer the AV along a tracking tape. In the discussion that follows, we have made some simplifying assumptions about the dynamics of the robot in Figure 6 to make the modelling of the behavior of this control system more concise. We assume that the tracking tape is over a perfectly flat terrain, the universe of the robot is limited to following the tracking tape which crosses light-controlled intersections, and we consider only three control variables: deviation angle θ, distance x, and image (used to detect traffic lights). The camera images are processed by a computer to obtain any necessary adjustments in terms of two directional control variables: the deviation angle θ of the wheels and an x-distance of the AV plant relative to the tracking line in Figure 6. The deviation angle θ is used to change the direction of the AV so that it travels in parallel with the tracking line. The AV controller also determines an x-distance (the distance between the AV's longitudinal axis and the center of the tracking line). The x-distance in Figure 6 is used to guide the AV back onto the tracking line.

The navigation unit in Figure 6 also analyzes feedback from the video camera to detect intersection lights. In the temporal specification in Figure 7, the action

process($x_<$: real; $\theta_<$: real; $image_<$: imagetype),

specifies the processing of the images by the LCV navigation unit. The image parameter affects the behavior of the LCV if either a green or red light is detected. A light-controlled intersection is a shared resource (only robots going in the same direction can cross the intersection). If an LCV "sees" green, it stops rolling and its navigation unit (node 1) transmits a request to the light controller to enter the intersection. Once the LCV acquires permission from the light controller (node 2) to continue, it rolls through the intersection. On the other hand, if an LCV sees red, it also stops rolling and requests a green light. Once the light changes to green, the LCV must still request permission to enter the intersection. The behavior of the guard unit software running on node2 of Figure 6, consists of synchronizing the directional lights *infinitely often* and either responding to a request for access to the intersection or responding to a robot which wants a red light changed to green. In the context of the real-time system in

Figure 6, the processes running on nodes 1 and 2 are called agents, which can be concisely specified with real-time temporal logic.

5.2 Temporal Specification of Controller Behavior.

We make some simplifying assumptions to reduce the complexity of the temporal specification of the behavior of the control system. First, by treating light-controlled intersections as "critical sections" (only one LCV at a time traverses an intersection), we have eliminated the need for a yellow (warning) light. Second, only a pair of robots traveling in opposite directions compete for access to light controlled intersections. Third, a robot which receives permission to enter an intersection always clears the intersection before the light controller changes the lights. The specification of the behavior of the navigation and guard units of the LCV is given in Figure 7.

```
--navigation unit of mobile robot:

◇ω seq(delay(5),                                  --time to align camera
      process(x< : integer; θ< : real; image<: imagetype); delay(10),
      update(x> : integer; θ> : real;  signal< : signaltype); delay(k),
      timeout ⟹ correct( ); delay(50),
      SeeLight(image) ⟹
            seq(  τ; delay(10),          --internal action
                  SeeGreen(image) ⟹ seq(guard.request_io; delay(15),
                                        roll; delay(30))
                  or
                  ¬SeeGreen(image) ⟹ guard.changelights_io; delay(30)))

--Light control guard:

◇ω seq(delay(7),                                  --time to synchronize lights
      request_io; delay(10),
      or IsClear ⟹ changelights_io; delay(10))
```

Figure 7. Temporal Specification of Controller for Mobile Robot

5.3 Modelling the LCV Controller with $TA_{i/o}$s

The behaviors of the navigation unit and guard can be modelled as $TA_{i/o}$s as shown in Figure 8. The navigation unit in Figure 8 is deterministic but not temporally complete, since no timeout transitions are specified, except for the update action on state q_2. The guard automaton in Figure 8 is also not temporally complete, since no timeout transitions are specified. The guard is non-deterministic, since a transition to either q_2' or q_1' from q_0' is always possible. Arcs without inscriptions are assumed to have enabling conditions which are true.

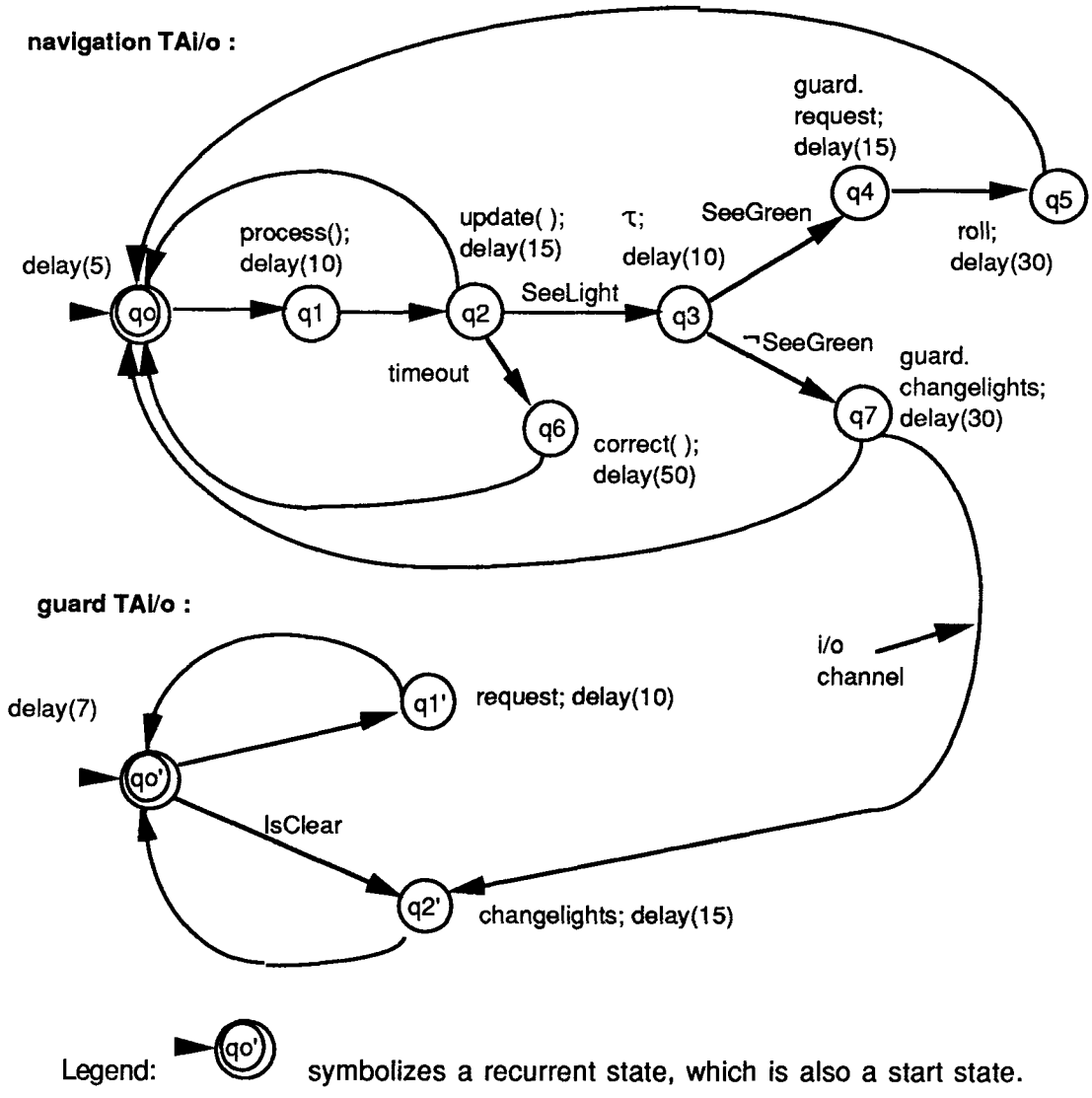


Figure 8. LCV Modelled with $TA_{i/o}$s

5.4 Tabular Representation of Timed Behavior

Automata can be conveniently represented in tabular form. To analyze the timed-behavior of the TAi/os in Figure 8, we construct table 5.1 for the guard automaton. Since there is only one timed trace containing the request action, the determination of the global-MAXT is just the sum of the upper bounds (6 and 9) for the actions on nodes 0 and 1. For simplicity, we ignore the timing constraints of the navigation unit in Table 5.1.

Table 5.1 Timed Behavior for LCV guard automaton

states	timing constraint	lb	up	local-MAXT	global-MAXT
qo' q1'	qo': delay(7)	0	6	6	6
qo' q2'	qo': delay(7)	0	6	6	6
q1' q0'	q1': request; delay(10)	6	9	9	15
q2' qo'	q2':changelights; delay(15)	6	14	14	20

5.5 Tabular Representation of Transductions

The tabular representation of transductions for an automaton is useful because it facilitates correctness proofs about the specification and construction of the specified program. To prepare a $TA_{i/o}$ for a proof of its correctness, and to establish the relationship between $TA_{i/o}$ predicates, we introduce a partial list of proof expressions (Part A) similar to those found in [Con 89] and attributes (Part B) of state predicates:

Table 5.2 Annotations on TAi/o Nodes

A. Proof Expressions.

Let $Tr_{q,q'}$, $Td_{q,q'}$ be transduction and transduction rule, respectively;

let p, q be predicates.

or(p, q)	--p or q
orin(p, q)	--or introduction
impel(p, q)	--implication elimination
def(p, q)	--q follows from definition of p
andin(p, q)	--and introduction
completes(ACT_{io})	--ACT_{io} completes
atmostone(ACT_{io})	--at most one i/o action completes
mutex(timed trace)	--timed trace guarantees mutually exclusive access to a shared resource.

B. Attributes of state predicates and conditions.

recurrent state predicate: [loop]	begin sequence: [seq]
transition to R state: [end loop]	end seq: [qes]
branching node: [or]	impl cond: [if]
separator: [;]	end if: [fi]
guard on acceptance of a call: [when ec =>]	select call: [select]
i/o node predicate p: [accept p]	end select: [end select]
p; delay(k): [p or delay(k)]	

The proof expressions facilitate proofs of automaton properties (e.g., mutual exclusion for an intersection guarded over by the "seeing eye" traffic light system hardware controlled by the guard program--only one mobile robot can be in the intersection any one time). The program specified by a $TA_{i/o}$ is extracted while proving that an automaton satisfies required properties. To extract the program specified by a $TA_{i/o}$, the meaning of each predicate is defined with an attribute representing a fragment of program code. Every node in Figure 9 has three types of predicates (proof expressions, state predicate, and attributes). The proof expressions used in Figure 9 are listed in Table 5.2A. For example, node q_o in Figure 9 is annotated with impel(delay(7), or($Tr_{qo',q1'}$, $Tr_{qo',q2'}$)), which is an application of implication elimination relative to delay(7) and the transduction rules evaluated in state q_o. The state predicate on node qo is also attributed with [loop] delay(7) [select]. To maintain the generality of the specification, the attributes of each part of a specification belong to an abstract programming language. The attributes of $TA_{i/o}$ predicates should be thought of as annotations (they are normally hidden, and added during the later stages of modelling). An annotated version of the guard in Figure 8 is given in Figure 9.

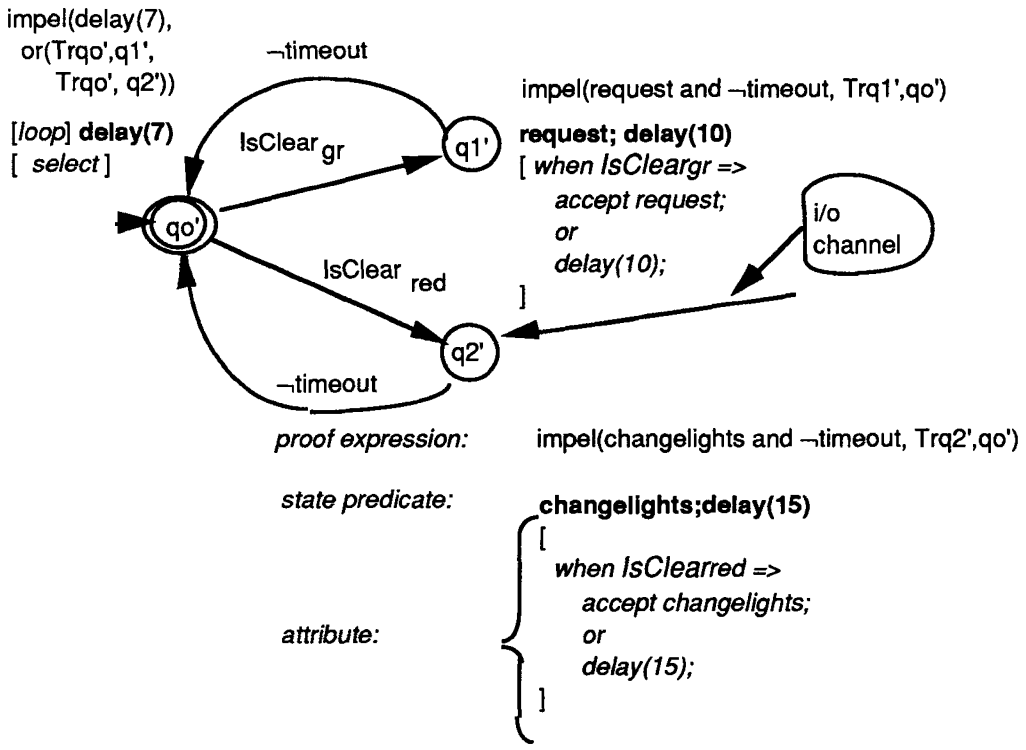


Figure 9. Annotated, Temporally Complete $TA_{i/o}$

Table 5.3 Automaton with Proof Expressions & Attributes

state	node	Transductions	Tr Rules	Proof Express.	Attributes
0	delay(7)	Tdqo',q1'(seq(delay(7), request; ...)	Trqo',q1': sat(0, delay(7))	impel(delay(7), or(Trqo',q1', Trqo,q2))	loop delay(7); select
1	request; delay(10)	Tdq1',qo' (seq(request; delay(10), delay(7))	Trq1',qo': sat(1, request...)	impel(request, Trq1', qo')	when IsCleargr=> accept request; or delay(10);
2	changel-.. lights; delay(15)	Tdq2', qo'(seq(changelights; delay(15), delay(7))	Trq2',qo': sat(2, change...& IsClearred)	impel(change..., Trq2',qo')	when IsClearred=> accept changelights; or delay(15);

We illustrate a correctness proof of an automaton specification in terms of the guard in Figure 9. We have minimized the states in this machine for the sake of illustration (a more elaborate machine would used in the general case). We will assume that this automaton has been made temporally complete (making sure that each state, except for the start state, has a timeout transition). In addition, we replace the inscriptions on the edges (in Fig. 8) with enabling conditions which are mutually exclusive for the case being considered:

replace $q_{0'},q_{1'}$: T *with* $q_{0'},q_{1'}$: IsClear_{gr} --green direction is clear
replace $q_{0'},q_{2'}$: IsClear *with* $q_{0'},q_{2'}$: IsClear_{red} --red direction is clear
replace $q_{1'},q_{0'}$: T *with* $q_{1'},q_{0'}$:¬timeout
replace $q_{2'},q_{0'}$: T *with* $q_{2'},q_{0'}$:¬timeout

As an aid to implementation of a fully attributed $TA_{i/o}$, we store the parts of the automaton in Table 5.3. The information in Table 5.3 also provides the basis for both proofs of automaton properties and program derivation. Arrival at a node provides *evidence* that the attributes of the node predicate belong to a correct specification (up to that point). In other words, proving an automaton property constructs a program.

6 Correctness Issues

There are three types of constraints that can be imposed on the behavior specified by a $TA_{i/o}$. On the state transition level, the conjunction of an enabling condition and node predicate serves as a constraint on a state change. This form of constraint is expressed by a transduction rule. An understanding of the remaining two types of constraints hinges on making a distinction between what we call "atomic automata" and "molecular automata." An *atomic automaton* ($aTA_{i/o}$) consists of nodes without underlying channels connecting them. A *molecular automaton* ($mTA_{i/o}$) has at least one pair of nodes connected by an underlying i/o channel; an $mTA_{i/o}$ is the result of a composition. On the atomic automaton level, a constraint is some property such as predictability, temporal completeness, determinism, mutual exclusion and so on, which the $aTA_{i/o}$ satisfies. On the molecular automaton level (a system of automata), a constraint can be placed on the interaction between atoms in the $mTA_{i/o}$. Examples of system properties are precedence (ordering of communications) and safety (nothing bad happens).

The proof of correctness of constraints on the state transition level has been developed for the specification of a knowledge-based, real-time system [Ram 91]. On this level, the concern is that state changes satisfy the consistency constraints to preserve the integrity of the information within a system. The proof of correctness of a state change is a deduction which is made with the help of a knowledge base. The proof of the correctness of an atomic automaton specification (i.e., demonstrating that the specification satisfies some property) is performed with the help of the information contained in proof-expression table given in Table 5.2A (we illustrate this idea below). A technique for proving that the specification provided by a molecular automaton satisfies system properties has been given in [Pet 90a, Pet 90b]. For simplicity, we only treat the case where the intersection is clear in the red direction, and a robot is waiting for the guard to change the lights. For this case, we show In Figure 10 the extraction of a partial abstract program from a constructive proof (for readability, we have omitted the single quotes on the states in Figure 9).

	Constructive Proof		(Partial) Abstract Program
1	$q_o \models$ delay(7)	given	
2	$q_o \models$ $IsClear_{red}$, $q_o \models \neg IsClear_{gr}$	assumed	
3	$q_o \models$ impel(delay(7), or($Tr_{qo,q1}$,$Tr_{qo,q2}$))	fr 1, graph	
4	or($Tr_{qo,q1}$,$Tr_{qo,q2}$)	fr 1, 3	
5	not $Tr_{qo,q1}$	fr 2	
6	or($Tr_{qo,q1}$,$Tr_{qo,q2}$)) and $\neg Tr_{qo,q1}$	fr 4,5, andin	
7	$Tr_{qo,q2}$ = sat(qo, delay(7) and $IsClear_{red}$)	fr 6	
8	$Td_{qo,q2}$ = seq(qo, delay(7), $changelights_{io}$...)	fr 7, Prop. 4.7	loop
9	$q_2 \models$ $changelights_{io}$; delay(15)	fr 8	delay(7);
10	completes($changelights_{io}$)	fr 9, def. 4.4	select
11	$q_2 \models \neg$timeout	assumed WLOG	
12	impel($changelights_{io}$ and ¬timeout ,$Tr_{q2,qo}$)	fr10, 11, graph	
13	$Tr_{q2,qo}$	fr 10,11,12	
14	$Td_{q2,qo}$	fr 13, Prop. 4.7	when $IsClear_{gr}$ =>
15	$q_o \models$ delay(7)	fr 14	accept
16	$q_o,q_2,q_o \models$ atmostone($changelights_{io}$)	fr 8, 14	$changelights_{io}$;
17	mutex($Td_{qo,q2}$, $Td_{q2,qo}$)	fr 16	or delay(15);

Figure 10 Extraction of Partial Abstract Program from Constructive Proof

The property we wish prove is that the guard guarantees mutual exclusion (only one mobile robot can be in an intersection at any one time). The guard must control the hardware so that the intersection is clear before instructing the hardware to change the lights. In Figure 10, we prove the correctness of the guard automaton in the case where the guard instructs the hardware to change the lights. The attributes for a fragment of an abstract program are extracted in column 4 of Figure 10 each time a transduction is made during the constructive proof. The remainder of the abstract program started in Figure 10 is obtained from the constructive proof that mutex($Td_{q0,q1}$, $Td_{q1,q0}$) holds. The proof expressions on the nodes in Figure 9 serve as an aid in automated reasoning about the specification. The formulation of the proof expressions stem from an interpretation of the structure of a TAi/o graph relative to the definitions and propositions we have given. To the extent that a program is identified with its behavior, a constructive proof of a $TA_{i/o}$ is the specified program. In other words, the proof constructs the specified behavior.

7 Conclusion

The $TA_{i/o}/TL_{rt}$ framework provides a basis for modelling the behavior of a real-time system. The annotation of node predicates with proof expressions makes it possible to construct provably correct prototypes of real-time systems. The attributes of node predicates facilitate the extraction of program code during a constructive proof. In effect, $TA_{i/o}$s provide a visual programming approach to the development of provably correct real-time systems. TL_{rt} provides a concise means of expressing transductions and properties of automata we wish to prove. The combination of visual programming, constructive proofs using transductions and transduction rules, and the expressiveness provided by TL_{rt}, provides an appealing approach to the design of reliable real-time systems.

Acknowledgement

We would like to thank Gideon Frieder and the other members of the School of Computer & Information Science at Syracuse University for providing an excellent environment for this research. We would also like to thank William Hankley, CIS Dept., Kansas State University, and Chetan Murthy, Computer Science Dept., Cornell University, for many helpful discussions concerning constructive logic and specification theory.

References

[Alp 86] Alpern, B.L. Proving Temporal Properties of Concurrent Programs: A Non-Temporal Approach, Ph.D. dissertation, Cornell University, TR-86-732, 1986.

[Alu 90] Alur, R. Dill, D. Automata for Modelling Real-Time Systems, LNCS 443: 322-335, Springer-Verlag, NY, 1990.

[Ark 90] Arkin, R.C. Integrating Behavioral, Perceptual, and World Knowledge in Reactive Navigation, Robotics and Autonomous Systems, vol. 6: 105-122 1990.

[Büc 62] Büchi, J.R. On a decision method in restricted second-order arithmetic, Proc. Int. Congr. Logic, Methodology, and Philosophy of Science 1960, Stanford University Press, 1-11, 1962.

[Con 80] Constable, R.L. The Role of Finite Automata in the Development of Modern Computing Theory, Proc. of the Kleene Symposium, North-Holland, 61-83, 1980.

[Con 84] Constable, R. L. Bates, J.L. The Nearly Ultimate Pearl, TR 83-551, Computer Science Dept., Cornell University, Jan., 1984.

[Con 86] Constable, R.L. et al. Implementing Mathematics with the Nuprl Proof Development System, Prentice-Hall, NJ, 1986.

[Con 89] Constable, R.L. Assigning Meaning to Proofs: A Semantic Basis for Problem Solving Environments, Constructive Methods in Computing Science, NATO ASI Series, vol. F 55, 63-90, 1989.

[Har 90] Harel, E. et al. Explicit Clock Temporal Logic, Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science, , 402-413, June, 1990.

[Hal 89] Halbwachs, N. et al. Specifying, Programming and Verifying Real-Time Systems Using a Synchronous Declarative Language, Proc. of Joint Univ. of Newcastle Upon Tyne/International Computers Limited Seminar, II.27-II.49, Sept. 1989.

[Hen 91] Henzinger, T.A., et al. Temporal Proof Methodologies for Real-Time Systems, Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages, 353-366, Jan., 1991.

[Hoa 85] Hoare, C.A.R. Communicating Sequential Processes, Prentice-Hall, NJ, 1989.

[Kla 91] Klarlund, N. Schneider, F.B. Proving Nondeterministically Specified Safety Properties Using Progress Measures, TR 91-1204, Cornell University, May, 1991.

[Krö 85] Kröger, F. Temporal Logic of Programs Lecture Notes, Report TUM-I8521, Institut fur Informatik, Technische Universitat Munchen, 1985.

[Lav 90] Lavignon, J.-F., and Y. Shoham. Temporal Automata, Report No. STAN-CS-90-1325, August, 1990.

[Lyn 88] Lynch, N., and M. Tuttle. An Introduction to Input/Output Automata, Report MIT/LCS/TM-373, November, 1988.

[Man 81] Manna, Z. A. Pnueli. Verification of Concurrent Programs, Part 1: the temporal framework, Report No. STAN-CS-81-836, Dept. of Computer Science, Stanford University, June, 1981.

[Man 83] Manna, Z. A. Pnueli. Verification of concurrent programs: a temporal proof system, Report No. STAN-CS-83-967, Dept. of Computer Science, Stanford University, June, 1983.

[Man 89] Manna, Z. Pnueli, A. Specification and Verification of Concurrent Programs by ∀-Automata, LNCS 398, 125-164, 1989.

[Mar 90] Maravall, D., et al. Guidance of an Autonomous Vehicle by Visual Feedback, Cybernetics and Systems, vol. 21, 257-266, 1990.

[McC 43] McCullock, W.S. Pitts, W. A logical calculus of ideas immanent in nervous activity, Bull. Math. Biophys., vol 5: 115-133, 1943.

[Mer 91] Merritt, M. et al. Time-Constrained Automata. To appear in Proc. 2nd Int. Conference on Concurrency Theory (Concur'91), August, 1991.

[Mil 89] Milner, R. *Communication and Concurrency*, Prentice-Hall, Inc., NJ, 1989.

[Mur 90] Murthy, C. Extracting Constructive Content from Classical Proofs, Ph. D. diss., Report 90-115, Cornell University, Aug., 1990.

[Mur 91] Murthy, C. Classical Proofs as Programs: How, What, When, and Why, To appear in the Proc. of Constructivity in Computer Science, summer, 1991.

[Ost 89] Ostroff, J.S. *Temporal Logic for Real-Time Systems*, John Wiley & Sons, Inc., New York, 1989.

[Ost 90] Ostroff, J.S. Wonham, W.M. A Framework for Real-Time Discrete Event Control, IEEE Transactions on Automatic Control, vol. 35, no. 4, 386-396, Ap., 1990.

[Pet 90a] Peters, J.F. Constructive Specification of Communicating Processes Using Temporal Logic, Ph.D. dissertation, Computing & Information Sciences, Kansas State University, 1990a.

[Pet 90b] Peters, J.F. Hankley, W. Proving Specifications of Tasking Systems Using Ada/TL, Proceedings of ACM Tri-Ada'90, 4-13, Dec., 1990b.

[Pet 91a] Peters, J.F. Ramanna, S. Modelling Timed-Behavior of Real-Time Systems with Temporal Logic. To appear in Cybernetics and Systems: An International Journal, 1991.

[Pet 91b] Peters, J.F. Ramanna, S. Prototyping Provably Correct Real-Time Systems, Report No. SU-CIS-91-23, School of Computer & Information Science, Syracuse University, July, 1991.

[Pus 90] Puschner, P. Zainlinger, R. Developing Software with Predictable Timing Behavior, Research Report 5/90, ITI, Technische Universitat Wien, Austria, Feb., 1990.

[Ram 91] Ramanna, S. Peters, J.F. Explicit Clock Logic in Consistency Constraints in Real-Time Systems. To appear in the IFAC Workshop on Artificial Intelligence in Real-Time Control (AIRTC91), Sept., 1991.

[Tor 90] Torn, A.A. PICA--A graphical program development tool, Acta Cybernetica, vol. 9, no. 3: 303-322, 1990.