Constructively Typed Timed Automata

J. F. Peters III
Constructively Typed Timed Automata

J. F. Peters, III

November 1991

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

J. F. Peters, III

Computer Science Department
University of Arkansas
Fayetteville, AR 72701 USA
peters@uafhp.uark.edu

Abstract

A new class of communicating automata called typed Timed Input/Output Automata (tTA_i/o) is introduced. A tTA_i/o is a predicate automaton used for specifying and reasoning about real-time systems. The typing discipline suggested for predicate automata is in the tradition of Martin-Löf's constructive type theory. A type A is a proposition, which is defined when a prescription for constructing a proof of A is given. A fragment of Girard's linear logic is used in classifying state types. An illustration of the use of tTA_i/o's in specifying a light-controller is presented. An abstract program is extracted during a proof of an automaton specification. To illustrate the methodology in constructive reasoning about a tTA_i/o, a proof which derives a partial abstract program is given.

Keywords and Phrases: Automata, Design Methodology, Constructive Type Theory, Program Specification, Real-Time Systems, Temporal Logic, Visualization.


1. INTRODUCTION

Finite state automata have been widely used to describe the behavior of agents in a real-time system [Alu 90, Chi 91, Con 80, Hal 89, Hen 90, Heu 90, Kla 91, Kle 56, Lav 90, Lyn 87, Man 89, McC 43, Mil 73, Mil 89, Ost 89, Ost 90, Pet 90a, Pet 90b, Pet 91a, Pet 91b, Pet 91c, Sme87, Wol 89]. An agent is that part of a system which has its own identity, its own externally observable behavior, and which persists over time [Mil 89]. Automata can be represented as finite, directed, labelled graphs with nodes representing agent states and 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 to form predicate automata as in [Aba 90, Alu 90, Alp 86,
A predicate automaton node can be annotated with a predicate to specify an activity associated with the automaton state. It is usually the case that the arcs of predicate automata are inscribed with predicates identifying enabling conditions for state transitions. The earliest appearance of predicate automata appears to be in the same paper that introduced automata themselves as a means of modelling the behavior of neural nets [McC 43]. Later, McCullock-Pitts nerve nets were envisaged as an illustration for a general theory of automata [Kle 56] and as a basis for computational semantics [Con 80].

The aim of this paper is to introduce the use of a class of predicate automata called typed timed i/o automata (tTAi/oS) to specify 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. A tTAi/o has provision for communicating with other such automata via input/output channels. In addition, the specification provided by a tTAi/o includes state types defined in terms of state predicates and arc predicates (enabling conditions for state transitions). In keeping with Martin-Löf's constructive type theory [PML 84], a state type is interpreted as a proposition. In constructive type theory, a type is defined by prescribing how to construct an object of that type. State types for tTAi/os are specified using a fragment of linear logic introduced by Girard [Gir 87].

The advantage of the typing discipline imposed on predicate automata is that typing provides a sound as well as convenient basis for proofs of specifications embodied in these automata. Typing the state, input, and output alphabets of an input/output automaton (Mealy machine) has been suggested as a means of simplifying the verification task in proving properties of automata [Chi 91]. The reasoning in the constructive proof of a specification embodied in a tTAi/o provides evidence that the specification satisfies some property. A proof is termed constructive when the evidence denoted by it can be computed from it. As in Nuprl [Con 84; Mur 90] 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 tTAi/o is a program.

An overview of finite automata which accept infinite words and which provide the context for this research, is given in Section 2. In Section 3, typed timed automata are introduced. A brief discussion about clock readings is given in Section 4. Section 5 provides an introduction to a subset of real-time temporal logic called TL\textsubscript{rt}. In Section 6, the notion of a temporally complete timed automaton is given. A temporal
specification of a light-controller for an intersection traversed by robots is given in the tTA\textsubscript{i/o}/TL\textsubscript{rt} framework in Section 7. A sample constructive proof of a specification is presented in Section 8.

2. MODELING REAL-TIME PROGRAM BEHAVIOR WITH TIMED AUTOMATA
To model timed behaviors with infinite length in the context of real-time systems, it is common to consider finite state automata labelled with hard, real-time constraints and which accept infinite words. These automata are variations of what are known as Büchi automata.

2.2 Timed Automata
Büchi Automata (BAs) are finite-state automata which accept infinite words \cite{Buc62}. A Büchi automaton \((\Sigma, Q, Q_0, \delta, F)\) is a finite state machine with an input alphabet \(\Sigma\), finite set of states \(Q\), start states \(Q_0 \subseteq Q\), final states \(F \subseteq Q\), and mapping \(\delta: Q \times \Sigma \rightarrow 2^Q\) representing state transitions labelled by symbols. Let \(\inf(w) \subseteq Q\) be the set of states, which are visited infinitely many times during a run over an infinite word \(w\). A run over an infinite word is an accepting run, if \(\inf(w) \cap F \neq \{\}\). A key advantage of Büchi automata is that temporal logic formulas can be directly translated into equivalent Büchi automata \cite{Var83}. In such translations, automaton state transitions are defined in terms of atoms of a temporal formula \cite{Alp86, deJ91}. As a result, Büchi automata provide a visual means of specifying properties of programs. By extending Büchi automata to include timing features, these timed Büchi automata can be used to model real-time systems.

2.2.1 Timed Büchi 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 \cite{Mer91, Hen90, Alu90, Lav90}. A timed Büchi automaton (TBA) is defined as a 6-tuple \((\Sigma, Q, Q_0, \text{Clocks}, \delta, A)\), which is a Büchi automaton extended with a finite set \(\text{Clocks}\) of real-valued clocks, and a finite set of state transitions given by \(\delta: Q \times \Sigma \times 2^{\text{Clocks}} \times \Phi(\text{Clocks}) \rightarrow 2^Q\). In a TBA, arcs are inscribed with predicates (timing constraints and possibly \(\text{reset}(x)\)). The \(\text{reset}(x)\) predicate asserts that clock \(x\) is reset to zero. Figure 1 gives an example of a TBA which accepts the timed language \(((a+b)c)^{03}\). The predicate \(\text{reset}(x)\) asserts that clock \(x\) is reset to zero in the transition from \(q_2\)
to \(q_3\). The timing constraint \(x \leq 5\) asserts that the transition from \(q_3\) to \(q_2\) can only occur if the elapsed time is within 5 ticks of clock \(x\).

![Timed Büchi Automaton referencing external clock \(x\)](image)

Figure 1. Timed Büchi Automaton referencing external clock \(x\)

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 90]. 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.

2.2.2 Undecideability of Timed Büchi Automata

In verifying whether an implementation \(I\) satisfies its specification \(S\), we can represent \(I\) and \(S\) with TBAs. However, the problem of determining whether an implementation language \(L(I)\) is a subset of a specification language \(L(S)\) is undecidable [Alu 90]. In addition, the class of languages accepted by TBAs is not closed under complementation. The undecideability and complementation features of TBAs have motivated the introduction of deterministic timed Müller automata presented in the next section.

2.2.4 Deterministic Timed Müller Automata.

A Müller automaton (MA) was first introduced in [Mül 63], and further investigated in [Alu 90, Arn 84, de J91, Gue 88]. An MA is a 5-tuple \((\Sigma, Q, Q_0, \delta, A)\) with \(\Sigma, Q, Q_0,\) and \(\delta\) as in a Büchi automaton, and the added feature that the accepting condition is defined by the states \(A \subset 2^Q\). Let \(\text{inf}(w)\) be a set of states of an MA, which are visited.
infinitely many times during a run over an infinite word. A word \( w \) is accepted by an MA, if \( \inf(w) \subseteq A \). In other words, an infinite computation is accepted by a Müller automaton, if the computation eventually cycles through a set of infinitely recurring states. A deterministic timed Müller automaton (dTMA) is a 6-tuple \((\Sigma, Q, Q_0, \text{Clocks}, \delta, A)\) with \( \Sigma, Q, \text{Clocks}, A \) as in a TBA, and with the following additional features:

- \( \text{Card}(Q_0) = 1 \).
- \( 2^{\text{Clock}} = \) sets of clocks.
- \( \Phi(\text{Clock}) = \) set of timing constraints.
- (Enabling conditions are mutually exclusive)

\[
\forall q \in Q, x \in \Sigma, c \in 2^{\text{Clocks}}, \exists ! p \in \Phi(\text{Clocks}), \exists ! q' \in Q: \delta(q, x, c, p) = q'
\]

In the case where there is only one run over any timed trace in a dTMA, the class of timed languages accepted by dTMAs is closed under union, intersection and complementation. As a result, it is now possible to decide whether an implementation satisfies its specification. Let \( M \) be a dTMA. Let \( \text{complement}(M) = (\Sigma, Q, Q_0, \text{Clocks}, \delta, 2^Q - A) \) be the complement of \( M \), and which is another dTMA as shown in [Alu 90]. The \( \text{complement}(M) \) has the same underlying structure as \( M \) with an accepting condition given by \( 2^Q - A \). That is, word \( w \) is accepted by \( \text{complement}(M) \) iff \( \inf(w) \subseteq 2^Q - A \). This line of reasoning allows us to decide whether \( L(M') \subseteq L(M) \), i.e., whether or not \( M' = \text{complement}(M) \). As a result, if we are given deterministic timed Müller automata \( I \) and \( S \), determining whether \( L(I) \subseteq L(S) \) is decidable [Pet 91c]. However, since dTMA nodes do not have predicates identifying actions associated with process states, they lack expressiveness as specifications of system behavior. In addition, a dTMA is untyped, which makes the proof of the correctness of its specifications more cumbersome. These drawbacks of dTMAs have motivated the introduction of typed timed automata presented in the next section.

3 TYPED TIMED I/O AUTOMATA

To model the timed-behavior of communicating processes in real-time systems, we introduce a class of predicate automata called typed Timed I/O Automata (tTAl/o). A tTAl/o is an extension of a deterministic timed Müller automaton. A tTAl/o enforces a constructive typing discipline. The timed actions associated with a state of a tTAl/o are specified with typed state predicates; arcs of tTAl/oS are inscribed with typed enabling
conditions for transitions. The typing discipline enforced by a tTAi/o adheres to the intuitionistic type theory of Martin-Löf [PML 73, PML 79, PML 84, Con 86, Nor 86, Tur 89]. The constructive interpretation of any predicate P is that P is provable. The notation p : P denotes p is of type P. In an attempt to classify the rich set of node structures in a typed timed automaton, the nodes of a tTAi/o are typed. A node q has state type \( Q \), where \( Q \) is the type of its proof. Similarly, an automaton M has type \( T_M \), which is the type of its proof.

Typed TAi/0s are communicating automata. When tTAi/0s are composed, message-passing between the automata is made possible by the presence of hidden input/output channels. Each tTAi/o has input/output channel variables used in sending and receiving messages over i/o channels. Input/output automata (Ai/oS) were introduced by Lynch and Tuttle [Lyn 88], and extended to include timing constraints by [Mer 91]. Temporal Input/Output Automata (TAi/0s) were introduced in [Pet 91a], and elaborated in [Pet 91b, Pet 91c]. However, a TAi/o is less suitable for proofs of specification, since a TAi/o is untyped. The language accepted by a tTAi/o is the set of the timed behaviors of an agent. Acceptance of the behaviors of an agent by a tTAi/o ensures that each sequence of events in an agent behavior satisfies a property specified by the automaton.

A tTAi/o is a 9-tuple \((P, Q, Q_0, D, \text{Clocks}, \delta, N, E, A)\) with Clocks and A as in a dTMA, with typed states \( Q \), start states \( Q_0 \subseteq Q \), and where

\[
\begin{align*}
P &= \{ p : P \mid p \text{ is a proposition of type } P \} \\
D &= \{ I \text{ (input channel variables)} \} \cup \{ O \text{ (output channel variables)} \} \\
&\cup \{ \text{state variables } \text{time} : \text{Real}, ... \} \cup \{ \text{rigid variables } T : \text{Real}, ... \} \\
\delta : &~ Q \times P \times P \times D \times \Phi(\text{Clocks}) \rightarrow 2^Q \quad \text{(state transition)} \\
N \subseteq &~ Q \times P \times \Phi(\text{Clocks}) \times I \times O \quad \text{(state predicates)} \\
E \subseteq &~ Q \times P \times I \times O \quad \text{(arc predicates)}
\end{align*}
\]

In the next section, the classification of state types in terms of state and arc predicates is given.

3.1 State Types

The nodes of a finite, directed, labelled graph representing a tTAi/o, are automaton states, which are typed. The set of typed automaton states \( Q \) can be viewed as a union of sets of typed states:
\[ Q = Q_1 \cup Q_2 \cup ... \cup Q_i \cup ... \cup Q_n \]

The set \( Q_i \) is interpreted constructively as a type. Then it is necessary to prescribe formation rules for type \( Q_i \), so it can be determined when an automaton state \( q \) is a member (read "proof object") of \( Q_i \), and when two members of \( Q_i \) are equal. The membership and equality rules for state types of a predicate automaton are defined in terms of a function with fixed points. The *fixed point of a function* \( f : X \rightarrow X \) is an object \( x \) in the domain of \( f \) such that

\[ f(x) = x, \text{ where } x \text{ is a fixed point of } f \text{ for } x \in X \]

Let \( q \) be a state of a tTAi M. Let \( p : P, e : E, \phi : \Phi \) be a state predicate labelling \( q \), enabling condition inscribed on an arc \((q, q')\), and an automaton property for M, respectively, of automaton M. Let

\[ \text{sat}(q, p : P \land e : E \land \phi : \Phi) \]

assert that the conjunction \( p : P \land e : E \land \phi : \Phi \) is satisfied in state \( q \), which has a single outgoing arc. Then the following function \( ii_{\text{seq}} \) has fixed points relative to the satisfiability of \( p : P \land e : E \land \phi : \Phi \):

\[
ii_{\text{seq}}(q) = \begin{cases} 
q \in Q_i \text{ if } \text{sat}(q, p \land e \land \phi) \\
q' \in Q' \mid Q' \cap Q_i = \{ \} 
\end{cases}
\]

That is, state \( q \) is a fixed point of function \( ii_{\text{seq}} \), if the conjunction \( p \land e \land \phi \) is satisfied in state \( q \) and state \( q \) belongs to state type \( Q_i \); otherwise, \( q \) belongs to some other state type \( Q' \), where \( Q' \) and \( Q_i \) are disjoint. Functions of the form \( ii_{\text{seq}} \) are useful in formulating membership and equality rules for state types.

The key to distinguishing one state type from another is identifying the kinds of transitions that are possible from a given state. So, for example, we can collect
together all those states having a single choice of a transition (with outgoing arc labelled e). Let all states with a single outgoing arc be of type $Q_{seq}$ (i.e., as part of a sequence of states beginning with state $q$). Let $(q', q'')^o$ in $E$ denote that $(q', q'')$ is the only outgoing arc from state $q'$. Then the membership and equality rules for type $Q_{seq}$ are given by

\[ \text{membership:} \]
\[
(p' \text{ labels } q', \ e' \text{ inscribes } (q', q'')^o, \ \phi : \phi) \quad q \in Q_{seq} \implies \text{il}_{seq}(q') \in Q_{seq}
\]

\[ \text{equality:} \]
\[
(p = p' \text{ labelling } q', \ e = e' \text{ inscribes } (q', q'')^o, \ \phi : \phi) \quad q = q' \implies q' \in Q_{seq}
\]

### 3.1.1 Linear Logic Classification of State Types

In a typed timed input/output automaton, there is a rich variety of state types. To classify state types, we utilize the disjoint sum $\oplus$ and constructive or (written par) operators from linear logic [Gir 87] given in Table 1.

#### Table 1. Linear Logic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\oplus. a,b$</td>
<td>Disjoint sum (additive disjunction), where $\oplus. a,b$ reads &quot;choice of a single alternative, independent of other choices.&quot;</td>
</tr>
<tr>
<td>$! a$</td>
<td>Iterability of $a$.</td>
</tr>
<tr>
<td>$a \multimap b$</td>
<td>Causal implication (linear implication). Let $\Gamma$ be a set of formulas, then $\Gamma, a \vdash b$</td>
</tr>
<tr>
<td>$a \Rightarrow b$</td>
<td>Intuitionistic implication, where $a \Rightarrow b \equiv (\land a) \Rightarrow b$.</td>
</tr>
<tr>
<td>$a \perp$</td>
<td>Negation of $a$.</td>
</tr>
<tr>
<td>$\mathsf{par.} a,b$</td>
<td>Constructive &quot;or&quot; (dual of $\oplus$), where $\mathsf{par.} a,b$ expresses dependency between two types of actions (negation of $a$ implies $b$ or negation of $b$ implies $a$), i.e., $\mathsf{par.} a,b \equiv a \perp \multimap b \text{ or } b \perp \multimap a$.</td>
</tr>
</tbody>
</table>
In Table 1, the notation \( op. \ a, b \) is the prefix form of \( op \ b \). In classifying state types, we identify various choices of transitions that are possible from a given state. These choices of transitions from a state \( q \) are based on the evaluation of the state predicate \( p \) on \( q \), enabling condition(s) (one or more arc inscriptions symbolized by \( e, e', e'' \), ...) on arc(s) leaving \( q \), and an automaton property \( \phi \) (it must be satisfied in every state of the automaton!). The selection of states which belong to a state type is carried out in terms of the fixed points of an inecond: \( Q \rightarrow Q \) function named in terms of some enabling condition econd. We give a selection of these state types in Table 2. In Table 2, we have hidden the issue of whether a state represents an internal action (without i/o) or a state represents an action with i/o. Normally, the parameters of a state predicate will tell the story.

### Table 2. State Types

<table>
<thead>
<tr>
<th>State Type</th>
<th>Fixed Point Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q_{\text{seq}} )</td>
<td>( { q, \text{sat}(q, p \land e \land \phi) } )</td>
<td>q has a single outgoing arc and ( q ) satisfies ( p \land e \land \phi ).</td>
</tr>
<tr>
<td>( Q\oplus )</td>
<td>( { q, \text{sat}(q, p \land \oplus . e, e' \land \phi) } )</td>
<td>( q ) has arcs ((q, q'), (q, q'')) labelled ( e, e' ), respectively, and ( q ) satisfies ( p \land \oplus . e, e' \land \phi ).</td>
</tr>
<tr>
<td>( Q_{\text{par}} )</td>
<td>( { q, \text{sat}(q, p \land \text{par}. e, e' \land \phi) } )</td>
<td>( q ) has arcs ((q, q'), (q, q'')) labelled ( e, e' ), respectively, and ( q ) satisfies ( p \land \text{par}. e, e' \land \phi ).</td>
</tr>
<tr>
<td>( Q_{\text{abort}} )</td>
<td>( { q, \text{sat}(q, \neg p) } )</td>
<td>( q ) fails to satisfy its state predicate.</td>
</tr>
<tr>
<td>( Q^{\infty} )</td>
<td>( { q } )</td>
<td>Recurrent states (see Section 3.1.2).</td>
</tr>
</tbody>
</table>

In cases where we want explicit indication of a state type for automaton states with i/o, we use the notation \( Q_{i/o} \). Then, for example, \( Q\oplus \) with i/o would be written \( Q_{i/o \oplus} \), and so
Also notice that the membership and equality rules for formation of state types \( Q_{=} \), \( Q_{\text{par}} \), and \( Q_{\text{abort}} \) have the same form as the formation rules for \( Q_{\text{seq}} \).

### 3.1.2 Recurrent states.

The state type \( Q_{\infty} \) in Table 2 corresponds to the set of recurrent state in a Müller automaton. Then notation \( q : Q_{\infty} \) denotes a state that occurs infinitely often during an accepting run of a tTAi/o. There are many different types of recurrent states relative to (each of the types in Table 2 can be recurrent). In intuitionistic terms, the judgement \( q \in Q_{\infty} \) asserts that \( q \) is a proof object of \( Q_{\infty} \). That is, a recurrent state type \( Q_{\infty} \) is inhabited by a state \( q \), if \( q \) is a proof object in the type \( Q_{\infty} \). This still leaves open the question of the meaning of \( \infty \) in this context. This can be explained using an intuitionistic interpretation of the mathematics of infinity suggested by Martin-Löf [PML 88].

Let the fixed point operator \( \text{fix}(f) \) with respect to some function \( f \) be defined as expressed in the domain theory of Scott [Sco 82]:

\[
\text{fix}(f) = f(f(f(\ldots)))
\]

The following rules for the fixed point operator are used to define infinity:

\[
\begin{align*}
(x \in A) & \quad & (x \in A) \\
a \in A & \quad f(x) \in A & a \in A & \quad f(x) \in A \\
\hline
\text{fix}(a, f) \in A & \quad \text{fix}(a, f) = f(\text{fix}(a, f)) \in A
\end{align*}
\]

Then infinity is defined as

\[
\infty = \text{fix}(0, \text{succ}) = \text{succ}(\text{fix}(0, \text{succ})) \in \text{Nats}
\]

In a similar manner, we can use the notion of the fixed point of a recursive function \( \text{ii}_{\infty} \) in terms of the succ function and seq operator (used to define automaton transductions) to formulate the rules for determining membership in a recurrent state type. The seq operator is explained in terms of transductions. Let \( Td_{q,q'} \) symbolize a transduction from state \( q \) to \( q' \). Informally, a transduction is a sequence of states represented by \( \text{seq}(q, q') \), where \( q \) occurs before \( q' \). The notation \( p \mid (p') \) indicates that \( p \) is at the head of a list of predicates with tail \( (p') \). Each of the predicates \( p \mid (p') \) is associated with the sequence of states \( \text{seq}(q, q') \) in a transition from \( q \) to \( q' \) with enabling condition \( e \).
In a transduction, $\text{sat}(q \mid (q'), p \land e \land \phi)$ occurs before $\text{sat}(q', p' \land e' \land \phi)$. Later, the idea of a transduction will be expressed formally as a temporal logic formula. In the context of transductions from state $q$ to some other state $q'$ in a deterministic timed M"uller machine, the function $ii^{\omega}$ is defined recursively as follows:

$$ii^{\omega}(x, q, \text{seq}) = \begin{cases} 
\text{seq}(q, ii^{\omega}({\text{succ}(x)}, q', \text{seq})) & \text{if } q = q' \text{ in } Td_{q,q'} \\
\text{seq}(q, ii^{\omega}(x, q', \text{seq})) & \text{else}
\end{cases}, x \in \text{Nats}$$

We can determine if a state $q$ is a recurrent state ($q \in Q^{\omega}$) in a $t$TAi/o as follows:

$$q \in Q^{\omega}, \text{ if } \text{sat}(q, p \land e \land \phi) \forall x \in \text{Nats as } x \rightarrow \infty \text{ in } ii^{\omega}(x, q, \text{seq})$$

The $x$ parameter in $ii^{\omega}(x, q, \text{seq})$ serves as a counter, which approaches infinity as long as each occurrence of $q$ in this recursion satisfies the conjunction $p \land e \land \phi$. The $ii^{\omega}(x, q, \text{seq})$ recursion results in a repetition of state $q$ which repeats infinitely often as part of a sequence of transductions. There is also the possibility of a finite number of other automaton states following $q'$ before the recurrent state $q$ recurs (i.e., before $q$ appears again during an accepting run). This is the significance of before in the informal definition of seq. That is, $\text{seq}(q, ii^{\omega}(\text{succ}(x)), q', \text{seq})$ says $q$ occurs before $ii^{\omega}(\text{succ}(x)), q', \text{seq})$ as in

$$\text{seq}(q, ii^{\omega}(\text{succ}(x)), q', \text{seq}) = \text{seq}(q, ..., \text{seq}(q, ii^{\omega}(\text{succ}(x)), q', \text{seq})))$$

$$x \in \text{Nats as } x \rightarrow \infty$$

which expresses the fact that state $q$ is visited infinitely often, if $x \rightarrow \infty$.

**Notation.** $\text{seq}(q, ..., \text{seq}(q, ii^{\omega}(\text{succ}(x)), q', \text{seq})))$ asserts that state $q$ occurs before state $q'$ and eventually there is a transition from $q$ to $q'$ ('before' and 'seq' are defined formally in Section 5.1).

The formation rules for recurrent states are
membership:

\[(p' \text{ labels } q', \quad e' \text{ inscribes } (q', q'')^0, \quad \phi : \Phi ) \quad \text{sat}(q', p' \land e' \land \emptyset ) \quad \forall x \in \text{Nats as } x \rightarrow \infty \quad q \in Q^\infty \quad \text{in } ii^\infty(x, q', \text{seq}) \]

equality:

\[(p = p' \text{ labelling } q', \quad e = e' \text{ inscribes } (q', q'')^0, \quad \phi : \Phi ) \quad \text{sat}(q', p' \land e' \land \emptyset ) \quad \forall x \in \text{Nats as } x \rightarrow \infty \]

\[q' \in Q^\infty \quad q = q' \in Q^\infty \]

Let \(\sigma = \text{seq}(q_0, q_1, ..., q_i, ..., q_n)\) be a sequence of states visited during an accepting run of a tTAi/o for an infinite word \(w\). Let \(\text{inf}(w)\) be the set of states in \(\sigma\) each of which is visited infinitely many times during a run over \(w\). As a result,

\[\text{inf}(w) = \{q \in Q \mid \text{sat}(q, p \land \text{econd } \land \phi ) \quad \forall x \in \text{Nats as } x \rightarrow \infty \quad \text{in } ii^\infty(x, q, \text{seq})\} \subseteq A.\]

### 3.1.3 Example.

In illustrating state types, we introduce a notation for what we call transduction rules.

**Notation.** A transduction rule is a satisfaction clause of the form \(\text{sat}(q \mid (q'), p \land \text{econd } \land \phi )\), which is symbolized by \(\text{Tr}_{q,q'}\). In a typed timed automaton, the satisfaction of a \(\text{Tr}_{q,q'}\) is accompanied by a transduction \(\text{Td}_{q,q'}\) from state \(q\) to \(q'\). Because we are interested in using constructive type theory in proving a specification provided by a tTAi/o, we normally indicate the typing of an automaton state by

\[q : \text{sat}(q \mid (q'), p \land \text{econd } \land \phi )\]

In the case where there is no need to indicate succeeding states in a transduction rule (see Figure 2, for example), we simply write \(q : \text{sat}(q, p \land \text{econd } \land \phi )\). To denote that a state \(q\) is a recurrent state type, we write \(q : \text{sat}(q \mid (q'), p \land \text{econd } \land \phi )^\infty\).

An example of tTAi/o is given in Figure 2. In Figure 2, \(q_0\) is both a start state and a recurrent state of type \(Q^{\infty}\). The remaining two states (\(q_1\) and \(q_2\)) in Figure 2 are of type \(Q_{\text{seq}}\) (in each of these states, there is only one possible transduction). In this
example, the presence of input/output channels is hidden. Later, we return to an interpretation of this automaton in terms of a hardware controller, where there is explicit use of i/o channels (see Figure 3).

![Figure 2. Typed Automaton](image)

### 3.2 Automaton Types

The typing of automata is hierarchical. This hierarchy starts with automaton states, and extends to automata, and has been extended to systems of automata [Pet 91c]. Let $T_m$ be an automaton type, and let

$$IN = \text{set of possible inputs to } t \in T_m,$$

$$t(\text{in})(\text{in} \in IN) = \text{set of outputs of automaton } t$$

Let $\phi$ be a property which is satisfied by automaton $t$. The notation $\text{sat}(t, \phi)$ says property $\phi$ is satisfied by automaton $t$. The membership and equality rules for $T_m$ are

**membership:**

$$t \in T_m \quad \text{sat}(t', \phi)$$

**equality:**

$$(\forall \text{ in } IN, \phi \in \Phi, t' \in T_m)$$

$$t \in T_m \quad t(\text{in}) = t'(\text{in}) \quad \text{and} \quad \text{sat}(t', \phi)$$

$$t' \in T_m \quad t = t' \in T_m$$
In constructive type theory, a type is synonymous with a proposition. In the context of typed timed i/o automata, an automaton type $T_m$ is a proposition which specifies a real-time program. The judgement $t \in T_m$ asserts that $t$ is a proof of the specification $T_m$. The membership rule tells us that if $T_m$ is inhabited ($t \in T_m$), automaton property type $\Phi$ is inhabited ($\Phi \in \Phi$), and automaton $t'$ satisfies property $\Phi$ ($\text{sat}(t', \Phi)$), then $t' \in T_m$. The equality rule tells that two automata of type $T_m$ are equal, if they have the same output and satisfy the same property.

4 CLOCKS AND TIMED BEHAVIORS

Timing constraints of a typed $\text{TA}_{i/o}$ reference ticks of an external clock in the set Clocks. The flexible variable time (in the set of data variables $D$ in a $\text{TA}_{i/o}$) gives the value of a clock in the current state. Clock readings are non-negative, real numbers. Each time an event occurs, a reading of an external clock is associated with that event. That is, each event $e$ is conceptualized as a pair $(e, \text{time})$. As a result, a timed sequence of events $\beta$ in the behavior of an agent modelled by a $\text{TA}_{i/o}$ has a trace of the form:

$$\beta = (e_0, \text{time}_0), (e_1, \text{time}_1), ... (e_i, \text{time}_i), ...$$

Let $R^+$ denote the non-negative reals; Nats, the natural numbers 0, 1, ... In addition, let $\text{time}_i, \text{time}_j$ belong to $\beta$. Then, as in [Alu 90; Pet 90], a timed trace $\beta$ has the following properties:

- **Zero-time in start state:** $\text{time}_0 = 0$ in $(e_0, \text{time}_0)$
- **Strict Monotonicity:** $\forall i, j \in \text{Nats}: \text{time}_i < \text{time}_j$ for $i < j$
- **Unboundedness:** $\forall \text{time} \in R^+, \exists i \in \text{Nats}: \text{time} < \text{time}_i$

5 TIMED-BEHAVIOR EXPRESSED WITH EXPLICIT CLOCK 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. 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), and the inclusion of linear logic disjunction operators $\oplus$, $\text{par}$ [Gir 87]. Except for some additional derived temporal operators taken from [Pet 90], 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 $U: \Diamond \omega$ (infinitely often), and $\text{seq}(p_1, p_2, ..., p_n)$ (a temporally quantified sequence of state predicates where $p_1$ holds before $p_2$, and so on).

For the subset of RTTL (named $\text{TL}_{rt}$) we have chosen, the temporal language $\text{TL}_{rt}$ 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 $\neg$, $\forall$, $\oplus$, $\text{par}$, $\forall$, $(, )$, $U$

Well-formed formulas of $\text{TL}_{rt}$ 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 $\neg A$, $(A \oplus B)$, $(A \text{par} B)$, $(A U B)$ are formulas.

**5.1 Semantics of Temporal Operators.**

The $\neg$ (not) and $\forall$ (all) symbols have the usual semantics. In defining the semantics of the temporal operators for $\text{TL}_{rt}$, the notation

$$(q_0, ..., q_x) \models p \text{ for } x \geq 0$$
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 \(p, p', p_1, p_2, ..., p_n\) be predicates. The semantics of \(U\) as well as the operators derived from \(U\) are as follows:

\[
p U p' \quad \equiv \quad \exists k, x: 0 \leq x \leq k: (q_0, ..., q_x) \models p \text{ and } q_k \models p'
\]

\[
p \text{ before } p' \quad \equiv \quad \exists k: 1 \leq k: q_0 \models p \text{ and } (q_1, ..., q_k) \models p U p'
\]

\[
\diamond p \quad \equiv \quad \text{true U } p
\]

\[
q_k \models \text{seq}(p) \quad \equiv \quad q_k \models p
\]

\[
\text{seq}(p_1, p_2, ..., p_n) \quad \equiv \quad p_1 \text{ before seq}(p_2, p_3, ..., p_n)
\]

\[
\diamond \omega p \quad \equiv \quad \text{seq}(p, \diamond \omega p)
\]

The predicate \(p U p'\) asserts that the predicate \(p'\) 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 \(p'\) holds. By contrast, \(p\) before \(p'\) asserts that \(p\) is guaranteed to hold initially and sometime later \(p'\) will hold. For this reason, \(\text{before}\) 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.

5.2 Transductions and Transduction Rules

A transduction rule defines the basis for a transition between states in an automaton. A transduction rule is 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 \(e_{\text{cond}}\) be an enabling condition for the transition between states \(q\) and \(q'\), and let \(\phi \in \Phi\) be a property which is satisfied in state \(q\). Further, let \(Tr_{q, q'}\) be a transduction rule with respect to states \(q\) and \(q'\) having state predicates \(P_{< k}\) and \(P'\), respectively. \(Tr_{q, q'}\) is defined as follows:

\[
Tr_{q, q'} = \text{sat}(q \mid (q'), P_{< k} \land e_{\text{cond}} \land \phi \in \Phi)
\]
Notation. Let $t$ represent the current time; $k$, the number of ticks of an external clock; timeout, an exception condition which occurs if the evaluation of state predicate $P$ is not completed before the deadline specified by $k$ is reached. $P_{<k}$ is a timed state predicate $P$ with an upper time bound $k$ defined as follows:

$$P_{<k}(t) \equiv (\exists t' \in [t, t+k] \mid \Theta \cdot P(t'), \text{timeout})$$

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

$$Td_{q,q'} = \text{seq}(P_{<k}, P')$$

A transduction $Td_{q,q'} = \text{seq}(P_{<k}, P')$ asserts that "predicate $P$ is satisfied within $k$ ticks in state $q$ before predicate $P'$ is satisfied in state $q'$". On the one hand, a *transduction rule* is a 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 corresponding events. In the case where a $\text{tTA}_{i/o}$ is deterministic, there is a strict relationship between $Tr_{q,q'}$s and $Td_{q,q'}$s.

6 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.

**Definition** 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 $\text{delay}(k)$).

ii) for every state $q$, there is a transduction rule $Tr_{q,q'}$ which is valid.

By definition, a timed action specified by a node predicate leads to an event. Every event induces a transition to a new state in a complete $\text{TA}_{i/o}$, 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 1. (Peters and Ramanna, 1991b) Given the assertion $\text{ACT}_k$ on node $q$ in a complete $\text{TA}_{i/o}$. The completion of a timed action implies $Tdq, q'$. That is, a transition from state $q$ to $q'$ occurs.

A complete $\text{TA}_{i/o}$ ($P, Q, Q_0, D, \text{Clock}, \delta, N, E, A$) is deterministic if $\delta$ is a function. In the case where a temporally complete automaton is deterministic, we can state the relationship between transduction rules and transductions formally as follows:

Proposition 2. (Peters, 1991c). Let $\text{sat}(q | (q'), P < k \wedge e_{\text{cond}} \wedge \phi)$ 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 complete $\text{TA}_{i/o}$. Then

$$\text{Tr}_{q,q'} : \text{sat}(q | (q'), P < k \wedge e_{\text{cond}} \wedge \phi) \iff Tdq,q' : \text{seq}(P < k, P')$$

7 EXAMPLE SPECIFICATION
We illustrate the visualization of a controller in a real-time system in terms a very simplified model for a "seeing eye" controller, which guards an intersection used by mobile robots similar to those described in [Mar 90] and elaborated in Peters and Ramanna [Pet 91a]. In this model, a robot wanting to cross a light-controlled intersection and which sees a green light, uses its navigation controller to send a request to the light controller for permission to enter the intersection. It is the responsibility of the light controller to grant a request to cross the intersection, provided the intersection is clear. In the case where a robot approaching the intersection sees a red light, its navigation controller asks the light controller to change the lights. When a robot sees a green light, it still must request permission to cross an intersection. For simplicity, we assume the intersection is always clear in exactly one direction. The temporal specification of the behavior of the light controller is given textually by
--timed behavior of light controller

\[ \diamond \omega ( \text{delay}(10); \]
\[ \ominus. \text{when } \text{IsClear}_\text{red} \text{ do} \]
\[ \quad \text{GrantRequest}_i^<15, \]
\[ \quad \text{od}, \]
\[ \quad \text{when } \text{IsClear}_\text{green} \text{ do} \]
\[ \quad \text{ChangeLights}_i^<15, \]
\[ \quad \text{od} \]  
---time to synchronize the lights.
---wait 15 ticks for request
---for access to intersection.
---wait 15 ticks for request
---to change lights.

**Notation.** The subscript \(<15\) on \(\text{GrantRequest}_i^<15\) indicates that a deadline of 15 clock ticks has been imposed on \(\text{GrantRequest}_i\), which is a parameterless remote procedure (its connection to an i/o channel is symbolized by the io subscript). The light controller waits up to 15 clock ticks for a call by a navigation controller of a mobile robot wishing to enter the intersection.

The temporal logic specification of the light controller says that infinitely often after delaying 10 ticks, the controller waits 15 ticks for either (when the red direction is clear) a request from a robot to enter the intersection or (when the green direction is clear) a request from a robot to change the lights. The controller should preserve mutually exclusive access to the intersection. Let Waiting be the set of all robots currently waiting to cross the intersection; RedDirection, the set of all robots moving (or stopped!) within the intersection and in the direction in which the intersection light is red; GreenDirection, the set of all robots within the intersection and in the direction in which the intersection light is green. The visualization of a special case in the behavior of the light controller in terms of a tTAi/o is given in Figure 3.

The program specified by a tTAi/o is extracted while proving that an automaton satisfies required properties. To extract the program specified by a tTAi/o, the meaning of each predicate is defined with an attribute representing a fragment of program code. Let \(p_0\), \(p_1\), and \(p_2\) be state predicates for \(q_0\), \(q_1\), and \(q_2\), respectively. These predicates are attributed as follows:

\[ p_0, \text{ [ loop, select ] } \]
\[ p_1, \text{ [ when IsClear}_\text{gr} \text{ do, od]} \]
\[ p_2, \text{ [ when IsClear}_\text{red} \text{ do, od } ] \]
compiles to \(\text{loop } p_0 ; \text{ select}\)
compiles to \(\text{when IsClear}_\text{gr} \text{ do } p_1 \text{ od}\)
compiles to \(\text{when IsClear}_\text{red} \text{ do } p_2 \text{ od}\)
An annotated version of the guard in Figure 3 is given in Figure 4. 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).

\[
\begin{align*}
q_0 : \text{sat}(q_0, \text{delay}(7), \\
\land \text{IsClear}_{gr}, \text{isClear}_{red}, \\
\land \text{Card(Intersection)} = 0) \\
\end{align*}
\]

\[
\begin{align*}
q_1 : \text{sat}(q_1, \\
\text{GrantRequest}_{io<15}, \\
\land \text{Card(GreenDirection)} = 0) \\
\end{align*}
\]

\[
\begin{align*}
q_2 : \text{sat}(q_2, \\
\text{Changelights}_{io<15}, \\
\land \text{Card(RedDirection)} = 0) \\
\end{align*}
\]

Figure 4. Attributed Typed \(TA_i/o\)

8 CORRECTNESS ISSUES

In Figure 5, the attributes for a fragment of an abstract program are extracted each time a transduction is made during a proof of the specification. The property we wish to prove is that the light controller guarantees mutual exclusion (only one mobile robot can be in an intersection at any one time). The controller must control access to the
intersection it governs so that it is clear before changing the lights, or granting a robot permission to cross the intersection. In Fig. 5, completes(a), atmostone(a), mutex( ) mean "action a completes," "at most one i/o action completes," and "timed trace guarantees mutually exclusive access to a shared resource," respectively.

**Constructive Proof**

0 \( q_0 \models \text{Card(waiting)} > 0 \) \hspace{1cm} \text{assump.}
1 \( q_0 \models \text{Card(RedDirection)} = 0 \) \hspace{1cm} \text{assump.}
2 \( q_0 \models \text{Card(GreenDirection)} = 0 \) \hspace{1cm} \text{assump.}
3 \( q_0 \models \text{delay}(7) \) \hspace{1cm} \text{assump.}
3.1 \( q_0 \in Q_{\Theta} \) \hspace{1cm} \text{fr graph in fig. 4}
3.2 \( i_{i\Theta}(q_0) \) \hspace{1cm} \text{fr 3.1}
4 \( \text{sat}(q_0, \text{delay}(7)) \) \hspace{1cm} \text{fr 3.2}
\hspace{1cm} \land \Theta . \text{IsClear}_\text{gr}, \text{IsClear}_\text{red}
\hspace{1cm} \land \text{Card(waiting)} > 0 \) \hspace{1cm} \text{fr 0}
5 \( \text{IsClear}_\text{red} \) \hspace{1cm} \text{fr 1, 4}
6 \( Tr_{q_0,q_2} \) \hspace{1cm} \text{fr 3, 4, 5, def. of Tr}
7 \( Td_{q_0,q_2} \) \hspace{1cm} \text{fr 6, Prop. 2}
\hspace{1cm} \text{loop}
\hspace{1cm} \text{delay}(7); \hspace{1cm} \text{select}

7.1 \( q_2 \in Q_{\text{seq}} \) \hspace{1cm} \text{fr graph in fig. 4}
7.2 \( i_{\text{seq}}(q_2) \) \hspace{1cm} \text{fr 7.1}
8 \( \text{seq}(\text{delay}(7), \text{ChangeLights}_{io<15}) \) \hspace{1cm} \text{fr 7}
9 \( \text{sat}(q_2, \text{ChangeLights}_{io<15} \land \text{Card(RedDirection)} = 0) \) \hspace{1cm} \text{fr 1}
10 \( q_2 \models \text{ChangeLights}_{io<15} \) \hspace{1cm} \text{fr 9, def. of sat}
11 \( \text{completes}(\text{ChangeLights}_{io<15}) \) \hspace{1cm} \text{fr 10, assumed WLOG}
12 \( Tr_{q_2,q_0} \) \hspace{1cm} \text{fr 9, 11, def. of Tr}
13 \( Td_{q_2,q_0} \) \hspace{1cm} \text{fr 12, Prop. 2}
\hspace{1cm} \text{when IsClear}_\text{red} \text{ do}
\hspace{1cm} \text{ChangeLights}_{io<15} \hspace{1cm} \text{od}
14 \( q_0, q_2, q_0 \models \text{atmostone(ChangeLights}_{io}) \) \hspace{1cm} \text{fr 7, 13}
15 \( q_0, q_1, q_0 \models \text{atmostone(GrantRequest}_{io}) \text{ by symmetry} \hspace{1cm} \text{fr 14, 15}
\hspace{1cm} \text{when IsClear}_\text{gr} \text{ do}
\hspace{1cm} \text{GrantRequest}_{io<15} \hspace{1cm} \text{od}

**Figure 5. Partial Abstract Program from Constructive Proof**

In this discussion, we have not treated the automation of proving the correctness of a typed automaton specification. This is done by formulating an automaton property as a goal in Nuprl, and formulating tactics which automate the production of subgoals in
proving state types. We also have not treated the problem of how the code for an abstract program would be extracted during a constructive proof. In the example proof in Figure 5, the partial code for an imperative program is extracted (without addressing the issue of marking states like qo so that its attribute is not extracted more than once, or a transition so that it is not taken more than once during a proof). It should be noted that the imperative program extracted in Figure 5 is superfluous, since a complete proof of an automaton type done in Nuprl will result in computational content. However, if it is the intent of a designer to derive a controller to be run on a transputer, for example, then an imperative program (perhaps in Ada) might be desirable. The main thrust of this article is not on the program which is a byproduct of a constructive proof, but rather on the benefit of using a typed automaton in designing a software system.

9 CONCLUSION
The tTAi/o/TLrt framework provides a basis for modelling the behavior of a real-time system. The typing of automaton states contributes useful information in constructing provably correct prototypes of real-time systems. To the extent that a program is identified with its behavior, a constructive proof of a typed TAi/o is the specified program. In other words, the proof constructs the specified behavior. The attributes of node predicates facilitate the extraction of some form of familiar program code during a constructive proof. In effect, typed TAi/oTs provide a visual programming approach to the development of provably correct real-time systems. TLrt provides a concise means of expressing properties of automata we wish to prove. The combination of visual programming, constructive proofs, and the expressiveness provided by typed automata and TLrt, offers an appealing approach to the design of reliable real-time systems.

REFERENCES


[Nor 86] Nordström, B. Martin-Löf's Type Theory as a Programming Language. Report 27, Programming Methodology Group, University of Göteborg and Chalmers University of Technology (Sept. 1986).


[Sme 87] Smedings, R. Using trace theory to model discrete events. Report CS 8708, Department of Mathematics and Computer Science, University of Groningen, POB 800, Groningen, the Netherlands, August 1987.


