TIOA User Guide and Reference Manual

Stephen J. Garland Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

September 15, 2005

Abstract

TIOA is a simple formal language for modeling distributed systems with timing as collections of interacting state machines, called timed input/output automata. The TIOA Toolkit supports a range of validation methods, including simulation and machine-checked proofs. This user guide and reference manual includes a tutorial on the use of timed input/output automata and the TIOA language to model timed systems. It also includes a complete definition of the TIOA language.

Table of Contents

TIOA User Guide

1. Preface

Systems with timing constraints are employed in a wide range of domains including communications, embedded systems, real-time operating systems, and automated control. Many applications involving timed systems have strong safety, reliability, and predictability requirements, which make it important to have methods for the systematic design of these applications and for a rigorous analysis of their timing-dependent behavior.

The correctness and performance of timed systems often depends on the timing of events, not just on the order in which they occur. A typical timed system consists of computer components, which operate in discrete steps, and timing-related components such as clocks or physical processes, whose behaviors involve continuous transformation over time.

Timed (input/output) automata [3],[5] provide a mathematical framework that supports the description and analysis of timed systems. The TIOA language provides notations for describing timed automata precisely. The TIOA language is a variant of the IOA language [2], which was designed for use with basic (untimed) input/output (I/O) automata. Like IOA, TIOA is supported by a variety of analytic tools, ranging from lightweight tools, which check the syntax and static semantics of automaton descriptions, to medium-weight tools, which simulate the action of an automaton, and to heavyweight tools, which provide support for proving properties of automata.

This document is based on material in various descriptions of IOA [2] and TIOA [5]. It is organized into two parts, a *TIOA User Guide* and a *TIOA Reference Manual.* The *User Guide* begins with an informal tutorial on timed automata and the TIOA language. This tutorial consists largely of illustrative examples. Reading it should be sufficient for the reader to begin writing complete TIOA descriptions. The remainder of the *User Guide* provides more details about the constructs used in the examples. The subsequent *Reference Manual* presents the formal syntax and semantics of the TIOA language.

2. Introduction to timed automata

Timed (input/output) automata are nondeterministic state machines that form the basis of a mathematical model suited for describing time-dependent behavior in concurrent systems. By providing precise representations for both systems and their components as timed automata, the model enables us to view systems and to reason about them at varying levels of abstraction. Automata interact with each other and with the environment through discrete actions. Their internal state is invisible to other automata and to the environment; parts of that state can evolve continuously over time.

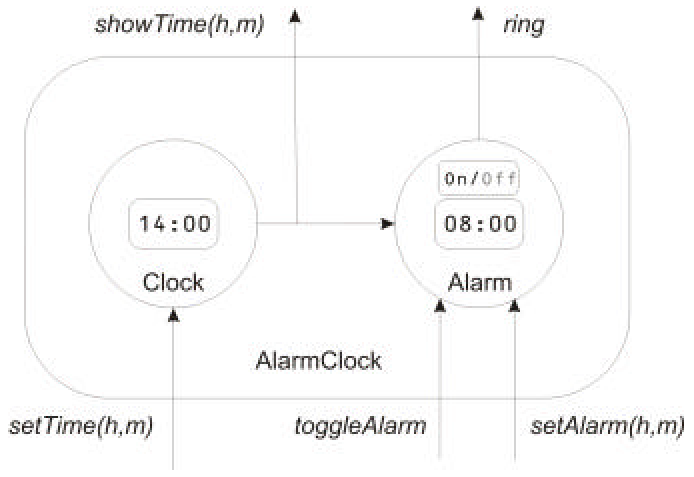

Figure 2.1. An alarm clock modeled as a timed automaton

Figure 2.1 illustrates the description of an alarm clock as a timed automaton: AlarmClock is a timed automaton with two timed automata, Clock and Alarm, as components. Arrows in the figure represent the *actions* through which the component automata communicate with each other and with the environment, which itself can be considered as yet another timed automaton.

Outgoing arrows represent *output actions* that are under the originating automaton's control. Figure 2.1 depicts two output actions: *showTime*, which represents a change in the time being displayed, and *ring*, which represents the alarm going off.

Incoming arrows represent *input actions*, which are not under the receiving automaton's control. The input actions *setTime*, *setAlarm*, and *toggleAlarm* in Figure 2.1 represent actions that originate in the environment and can occur at any time. The action *showTime* is an input action of Alarm that originates as an output action of Clock. In the model, paired output and input actions such as these occur simultaneously and indivisibly.

The *state* of an automaton is determined by the values of its *state variables*, which are visible only from within the automaton. State variables internal to the Clock automaton represent the current time, and state variables local to the Alarm automaton representing the on/off status of the alarm and the time at which it will ring if it is on. Note that because the states of these automata are not visible to each other or to the environment, the Clock automaton can communicate the current time to the Alarm automaton and to the environment only through the occurrence of a *showTime* action.

Actions occur, and the state of a timed automaton changes, instantaneously by *discrete transitions*. The state can also change over an interval of time by following a *trajectory*, which is a function that describes its evolution between discrete transitions. Each state variable has both a *static type*, which defines the set of values it may assume, and a *dynamic type*, which defines the set of trajectories it may follow.

3. Introduction to TIOA

Figure 2.1 depicts the actions of the automata that comprise the alarm clock, but it does not describe the actions completely (i.e., it says nothing about the variables *h* and *m*), only hints at the states of the automata, and provides no information at all about transitions or trajectories. We use the TIOA language to supply this missing information.

Figure 3.1 contains a TIOA description of the Alarm automaton. As shown in Figure 2.1, this automaton has three input actions and one output action. Here, however, we learn that two of the input actions are each parameterized by two elements of the set Nat of natural numbers, which are intended to express the time of day in hours and minutes on a 24-hour clock. The **let** statement that appears before the automaton definition defines a predicate, legalTime, used to constrain the values of these action parameters.

The automaton Alarm has three state variables: alarmTime, which has static type Nat, represents the time of day at which the alarm is set to ring (expressed in minutes past midnight), turnedOn, which has static type Bool (the set {true, false} of boolean values), represents whether the alarm is turned on or off, and ringing, also of static type Bool, represents whether the alarm should be ringing. The automaton has a single initial state in which these variables have the values 0, false, and false. The values of these variables can change only by the occurrence of a discrete transition.

The transitions of the automaton Alarm are given in precondition/effect style. The input actions have no preconditions, which is the same as having true as a precondition. This is the case for all input actions; that is, every input action in every automaton is enabled in every state. The effect of setAlarm is to set alarmTime to the time at which the alarm should ring (if it is on). The effect of toggleAlarm is to switch between the alarm being on and off (¬ is the logical *not operator*, which can be entered by typing the symbol ~). The effect of showTime is to set ringNow to true if the alarm should ring; this happens when the alarm is turned on and the automaton learns, through this input action, that the time shown on the clock is the time at which the alarm should ring. Finally, the output action ring can occur only when it is enabled, that is, only in states in which ringNow is true. Its effect is to set ringNow to false, which prevents the action from occurring again until another appropriate showTime action enables it once again.

```
let legalTime(hour, minute: Nat) = minute < 60 /\ hour < 24

automaton Alarm
  signature
    input showTime(hour, minute: Nat) where legalTime(hour, minute),
          setAlarm(hour, minute: Nat) where legalTime(hour, minute),
          toggleAlarm
    output ring
  states
    alarmTime: Nat := 0,
    turnedOn: Bool := false,
    ringNow: Bool := false
  transitions
    input setAlarm(hour, minute)
      eff alarmTime := (60*hour) + minute
    input showTime(hour, minute)
      eff ringNow := turnedOn /\ alarmTime = (60*hour) + minute
    input toggleAlarm
      eff turnedOn := ØturnedOn
    output ring
      pre ringNow
      eff ringNow := false
```

Figure 3.1. TIOA description of Alarm component

As for the Alarm automaton, we discover information about the parameters for the Clock automaton's actions in the TIOA description in

Figure 3.2. This automaton also has three state variables: whatToShow represents the time that will be displayed by the next showTime action, whenToShow represents when the next showTime action will occur (expressed in minutes beyond the time at which the clock was started), and now (a real number) represents the current time (also expressed in minutes beyond the time at which the clock was started).

The first two state variables are *discrete variables* whose values can change only by the occurrence of a discrete transition. Because the third state variable, now, has static type Real, it is an *analog variable* whose value can change continuously over time. TIOA uses the set of real numbers to represent real time, and it treats the values of analog variables as functions of real time.

The definition of the timePassage trajectory of the Clock automaton governs the evolution of the value of the analog variable now and its effect on the operation of the automaton. The evolve clause in this definition, by constraining the first derivative of now to have the constant value 1, constrains now to be a function $now(t) = t + C$ for some constant C; in essence, now represents the real time that has elapsed since the automaton started. The stop when clause in the transition definition states that time stops when the value of now equals whenToShow (the floor function truncates the real value of now to an integer). Time cannot advance again until some action, which is enabled when the stopping condition is true, causes the stopping condition to become false.

The effect of the setTime transition in the automaton Clock is similar to that of the setAlarm transition in the automaton Alarm. This action sets whatToShow to the time that should be displayed on the clock. In addition, it sets whenToShow so that the stopping condition of the timePassage trajectory becomes true, which prevents time from advancing until a showTime action occurs. A showTime action is enabled if time has stopped and the values of its parameters correspond to the value of whatToShow. The effect of such an action is to reset whenToShow to the next time the display needs to be updated (i.e., one minute from now) and to reset whatToShow to the time that will be shown then (with the displayed time wrapping back to 00:00 after it reaches 23:59).

```
automaton Clock
  signature
    output showTime(hour, minute: Nat) where legalTime(hour, minute)
    input  setTime(hour, minute: Nat) where legalTime(hour, minute)
  states
    whatToShow: Nat := 0,
    whenToShow: Nat := 0,
    now: Real := 0
  transitions
    input setTime(hour, minute)
      eff whatToShow := (60*hour) + minute;
          whenToShow := floor(now)
    output showTime(hour, minute)
      pre whenToShow = floor(now);
          hour = div(whatToShow, 60);
          minute = mod(whatToShow, 60)
      eff whenToShow := floor(now) + 1;
          whatToShow := mod(whatToShow + 1, 24*60)
  trajectories
    trajdef timePassage
      stop when whenToShow = floor(now)
      evolve d(now) = 1
```

Figure 3.2. TIOA description of Clock component

Finally, Figure 3.3 uses TIOA to define the alarm clock shown in Figure 2.1 as the composition of the Alarm and Clock automata. This definition matches each output action showTime of the Clock automaton with the input action of the same name and the same parameter values in the Alarm automaton. Actions matched in this fashion are performed simultaneously and indivisibly.

```
automaton AlarmClock
  components Clock; Alarm
```

Figure 3.3. TIOA description of an alarm clock

4. Specifying timed automata, in mathematics and in TIOA

Mathematically, a *timed (input/output) automaton* A is a tuple with six elements

- an *action signature* A_{sig} which is the union of disjoint sets A_{in}, A_{out}, and A_{int} of discrete *input*, *output*, and *internal* actions,
- a set A_V of *state variables*,
- a set A_S of *states*, which is a subset of the set of all possible valuations of A_V, (a *valuation* is a function f that assigns to each variable v in A_V a value f(v) in the static type of V),
- a set A_{S_0} of *initial states*, which is a non-empty subset of A_S,
- a discrete *transition relation* A_{tran}, which is a subset of $A_S \times A_{sig} \times A_S$, and
- a set A_{traj} of *trajectories* for A_V, which is a set of functions from intervals of time starting with 0 to A_S.

An action of an automaton is called *external* if it is an input or output action.

TIOA provides notations for defining timed automata either as *primitive* automata by specifying their names, signatures, state variables, transition relations, and trajectories, or as *composite automata* by specifying their decomposition into simpler timed automata. The following subsections describe these notations and their relation to the mathematical model of timed automata.

4.1. Automaton names and parameters

The first line of an automaton description in TIOA consists of the keyword **automaton** followed by the name of the automaton. The name may be followed by a list of formal parameters enclosed within parentheses. For example, the Channel automaton defined in Figure 4.1 has three parameters, i being the index of a process that uses the channel to convey messages of type M to another process with index j.

```
automaton Channel(i, j: Nat, M: type)
  signature
    input   send(const i, const j, m: M)
    output receive(const i, const j, m: M)
  states buffer: Seq[M] := Ø
  transitions
    input send(i, j, m)
      eff  buffer := buffer ?  m
    output receive(i, j, m)
      pre buffer ¹ Ø /\ m = head(buffer)
      eff  buffer := tail(buffer)
```

Figure 4.1. TIOA description of an untimed FIFO communication channel

There are two kinds of automaton parameters. An individual parameter, such as i: Nat or j: Nat, consists of an identifier and an associated type, and it denotes a fixed element of that type. Individual parameters with the same type can be specified together, as in i, j: Nat. A type parameter, such as M: type, consists of an identifier followed by the keyword **type**, and it denotes a type.

An automaton with individual parameters can contain a clause that constrains the values of those parameters. For example, an automaton whose definition begins with

automaton Swap(A, B: Set[Int]) **where** $A \subset B$

is parameterized by two sets of integers, the first of must be a proper subset of the second.

4.2. Action signatures

The signature for an automaton is declared using the keyword **signature** followed by lists of entries describing the automaton's input, internal, and output actions. Each entry contains a name and an optional list of parameters enclosed in parentheses. There are two kinds of action parameters. Varying parameters (such as hour, minute: Nat in

Figure 3.2) consist of identifiers with associated types, and they denote arbitrary elements of those types. A fixed parameters (such as **const** i and **const** j in Figure 4.1) consists of the keyword **const** followed by term denoting fixed element of its type. Neither kind of parameter can have **type** as its type.

Each entry in the signature denotes a set of actions, one for each assignment of values to its varying parameters. Thus the set of input actions for the Channel automaton contains one action send(i, j, m) for each value of the action parameter m of type M; the values of i and j in these actions are fixed by their values as parameters of the automaton.

It is possible to constrain the values of the varying parameters for an entry in the signature using the keyword **where** followed by a predicate. For example, the **where** clauses in

Figure 3.2 constrain the values of the parameters hour and minute. Thus the set of output actions for the Clock automaton contains one action showTime(hour, minute) for each pair of values of its parameters that satisfy the predicate legalTime(hour, minute).

4.3. State variables

As in the examples, state variables are declared in TIOA using the keyword **states** followed by a comma-separated list of state variables and their static types. The initial values of state variables can be constrained using the assignment operator :=. For example, the initial value of the state variable buffer in the Channel automaton (Figure 4.1) must be the empty set; hence there is a single initial state for this automaton.

Initial values of state variables need not be constrained in this fashion. For example, if the assignment := 0 were omitted from the declaration of the state variable whatToShow in the Clock automaton (

Figure 3.2), then that automaton would have an infinite number of initial states, one for each natural number n. If n is less than 24*60, the clock will display that time when power is turned on. Otherwise, the clock will display nothing until either a setTime action occurs or at least an entire day of real time elapses.

To rule out this latter aberrant behavior, the initial value of whatToShow can be constrained to be some arbitrary, but legal time of day by means of a declaration such as

```
whatToShow: Nat := choose n where n < (24*60)
```

When such a nondeterministic **choose** clause is used to initialize a state variable, there must be some value of the variable that satisfies the predicate following the where clause. If the predicate is true for all values of the variable, then the effect is the same as if no initial value had been specified for the state variable.

It is also possible to constrain the initial values of all state variables taken together, whether or not initial values are assigned to any individual state variable. This can be done using the keyword **initially** followed by a predicate (involving state variables and automaton parameters). For example, we can allow the Clock automaton to display an arbitrary time of day when its power is turned on by constraining the three state variables of the Clock automaton to have the same unspecified value:

```
states
   whatToShow: Nat, whenToShow: Nat, now: Real
   initially whatToShow = whenToShow /\ whatToShow = floor(now)
```

The order in which state variables are declared makes no difference: they are initialized simultaneously. Furthermore, the expressions denoting their initial values cannot refer to the values of any state variables.

4.4. Transition relations

Transitions for the actions in an automaton's signature are defined following the keyword **transitions**. A transition definition consists of an *action type* (i.e., **input**, **internal**, or **output**), an action name with optional parameters (see Section 4.5), an optional **where** clause, an optional precondition (see Section 4.6), and an optional effect (see Section 4.7). This definition groups transitions that involve a particular type of action together into a single piece of code.

More than one transition definition can be given for an entry in an automaton's signature. For example, we could define the transitions of the showTime action in the Clock automaton in two parts, one

```
output showTime(h, m) where h = 23 Þ m < 50
  pre whenToShow = floor(now);
      div(whatToShow, 60) = h;
      mod(whatToShow, 60) = m
  eff whenToShow := floor(now) + 1;
      whatToShow := whatToShow + 1
```

describing what happens before midnight and the other

```
output showTime(23, 59)
    pre whenToShow = floor(now) /\ div(whatToShow, 60) = 23 /\ mod(whatToShow, 60) = 50
    eff whenToShow := floor(now) + 1;
        whatToShow := 0
```

how the time of day is reset to 00:00 at midnight.

4.5. Transition parameters

The parameters that follow an action name in a transition definition must match those that follow the action name in the automaton's signature, both in number and in type. The simplest way to formulate parameters for a transition definition is to erase the keyword **const** and the type modifiers from the parameters given for the action in the automaton's signature; thus, in Figure 2.1, the parameters of the send action are given as (**const** i, **const** j, m: M) in the signature, but are shortened to (i, j, m) in the transition definition.

Action parameters and transition parameters differ in several respects. Parameters in the action signature can be terms (identified by the keyword **const**) that denote fixed values or they can be (declarations for) variables. If they are variables, their types matter, but their names do not. On the other hand, all parameters in transition definitions are terms, and the keyword **const** does not appear. Parameters in transition definitions can denote either fixed or varying values. If they contain no variables other than automaton parameter, then they denote fixed values. For example, the parameters in showTime(23, 59) denote fixed values. If they contain other variables (such as h and m), these variables can have arbitrary values.

Transition definition can contain additional *local parameters*, which are specified after the ordinary parameters and identified by the keyword **local**. Local variables serve two purposes. They can be constrained by a transition's precondition and used in the effects, as in

```
automaton PitchTwo(s: Set[Nat])
  signature output pitch(n: Nat)
  states left: Set[Nat] := s
  transitions output pitch(n; local x: Nat)
    pre n Î left /\ x Î left /\ n < x
    eff left := delete(n, delete(x, left))
```

which defines an automaton that discards two numbers at a time from a set, but communicates only the smaller of the two when a transition occurs. When the effects clause in a transition definition does not assign any values to a local variable, as is the case here, the definition can be rewritten using explicit quantification instead of local variables, as in

```
  transitions output pitch(n)
    pre n Î left /\ $ x: Nat (x Î left /\ n < x)
    eff left := choose s where $ x: Nat (x Î left /\ n < x /\ s = delete(n, delete(x), left))
```

In general, to eliminate local variables to which no values are assigned, one quantifies them explicitly in the precondition for the transition, and then repeats the quantified precondition as part of the effects clause.

Local parameters can also be used as temporary variables in the effects clause, as in the following definition of an automaton that sorts an array into ascending order by swapping pairs of incorrectly ordered elements.

.

```
automaton Arrange
  signature output swap(i, j: Nat)
  states A: Array[Nat, Nat]
  transitions output swap(I, j; local temp)
    pre A[I] < A[j]
    eff  temp := A[I]; A[I] := A[j]; A[j] := temp
```

4.6. Preconditions

The *precondition* in a transition definition is a *predicate* (that is, a boolean-valued expression) on the state indicating the conditions under which the transition can occur. In TIOA, preconditions can be defined for transitions of output or internal actions using the keyword **pre** followed by one or more predicates. If no precondition is present, it is assumed to be true. If a precondition contains more than one predicate, it is equivalent to the conjunction of those predictes.

An action π is said to be *enabled* in a state s if there is a state s' such that the triple (s, p, s') is the transition relation of the automaton. In TIOA, an action π is enabled in a state s if there are values for the local variables in one of the transition definitions for π that satisfy the transition's where clause and, together with the values of the state variables in state s, also satisfy the transition's precondition.

Since transitions of input actions cannot have preconditions, input actions are enabled enabled in every state; i.e., automata are not able to ``block" input actio ns from occurring.

4.7. Effects

The *effects clause* in a transition definition describes the changes that occur as a result of the action, either in the form of a simple program or in the form of a predicate relating the *pre-state* and the *post-state* (i.e., the states before and after the action occurs). However a transition is defined, it always happens instantaneously and indivisibly.

In TIOA, the effect of a transition is defined following the keyword **eff**, generally in terms of a (possibly nondeterministic) program that assigns new values to state variables. If a transition definition has no effects clause, then that transition leaves the state unchanged. The amount of nondeterminism in a transition can also be limited by a predicate relating the values of state variables in the post-state to each other and to their values in the pre-state.

4.7.1. Using programs to specify effects

A *program* is a list of *statements*, separated by semicolons. Statements in a program are executed sequentially. There are three kinds of statements:

- assignment statements,
- conditional statements, and
- **for** statements.

4.7.2. Assignment statements

An *assignment statement* changes the value of a state or local variable. The statement consists of a state or local variable followed by the assignment operator := and an expression. When a state variable is an array or a tuple, then terms denoting its elements or its fields can also appear on the left hand side (*lhs*) of the assignment operator, as in the automaton Arrange displayed at the end of Section 4.5.

The expression following the assignment operator must have the same type as the variable on the lhs of the assignment operator. TIOA considers the value of the expression to be defined mathematically, rather than computationally. This value is determined in the state in which the assignment statement is executed, and it becomes the value of the variable on the lhs in the subsequent state. Execution of an assignment statement does not have side effects; i.e., it does not change the value of any state or local variable other than the one on the left side of the assignment operator.

As illustrated in the discussion of the automaton PitchTwo (see page 10), the expression on the right side of an assignment statement can consist of a nondeterministic **choose** clause. The value of such a clause is constrained by a predicate following the keyword **where**. If the **choose** clause does not contain the keyword **where** (as in the statement x := **choose**), then it is treated as if it contained **where true**, and it produces an arbitrary new value.

4.7.3. Conditional statements

A *conditional statement* selects one of several program segments to execute in a larger program. Each conditional statement starts with the keyword **if** followed by a predicate and a **then** clause. The **then** clause contains a program segment that is executed if the condition is true. Each conditional statement ends with the keyword **fi**. As illustrated by

```
if x < y then x := x + y fi;
if x < y then x := x + y else y := x + y; x := x + y fi;
if x < y then x := x + y elseif y < x then y := x + y fi;
if x < y then x := x + y elseif y < x then y := x + y else y := x fi;
if x < y then x := x + y elseif y < x then y := x + y elseif x + y < z then y := x fi;
```

a conditional statement can contain any number of **elseif** clauses (each of which contains a predicate and a **then** clause) and/or a final **else** clause, which also contains a program segment that. The effect of executing a conditional statement is that of executing the program segment in the first **then** clause, if any, for which the preceding predicate is true and otherwise that of executing the program segment in the **else** clause, if one exists.

4.7.4. For statements

A **for** statement executes a program segment once for each value of a variable that satisfies a given condition. It starts with the keyword **for** followed by a variable, a clause describing a set of values for this variable, a **do** clause that contains a program segment, and the keyword **od**.

Figure 4.2 illustrates the use of a **for** statement in a high-level description of a multicast protocol that has no timing constraints. The figure begins with the definition of a vocabulary (i.e., a set of symbols) that can be used to describe packets sent by the protocol. The Packet data type to consist of triples [contents, source, dest], in which the contents field represents a message, the source field the Node sending the message, and the dest field the set of Nodes to which the message should be delivered. The state of the multicast algorithm consists of a multiset network, which represents the packets currently in transit, and an array queue, which represents, for each Node, the sequence of packets delivered to that Node, but not yet read by the Node.

The mcast action inserts a new packet in the network; the tuple data type provides the notation [m, i, I] and the multiset data type provides the insert operator by (see Section 4.3). The deliver action, which is described using a **for** statement, distributes a packet to all nodes in its destination set (by appending the packet to the queue for each destination node and then deleting the packet from the network). The read action receives the contents of a packet at a particular Node by removing that packet from its queue of delivered packets.

There are two ways to describe the set of values for the control variable in a **for** statement. The first (shown in Figure 4.2) consists of the keyword **in** followed by an expression denoting a set or multiset of values of the appropriate type, in which case the program segment in the **do** clause is executed once for each value in the set or multiset. The second consists of the keyword **where** followed by a predicate, in which case the program is executed once for each value satisfying the predicate. These executions of the program occur in an arbitrary order, and TIOA requires that the effect of a **for** statement be independent of the order in which executions of its program occur.

```
vocabulary Packet
  types Message, Node, Packet tuple [contents: Message, source: Node, dest: Set[Node]]

automaton Multicast
  imports Packet
  signature
   input     mcast(m: Message, i: Node, I: Set[Node])
   internal deliver(p: Packet)
   output   read(m: Message, j: Node)
  states
   network: Mset[Packet] := Ø,
   queue:   Array[Node, Seq[Packet]]
     initially "  i: Node (queue[i] = Ø)
  transitions
   input mcast(m, i, I)
     eff  network := insert([m, i, I], network)
   internal deliver(p)
     pre p  Î  network
     eff  for j: Node in p.dest do queue[j] := queue[j] ?   p od;
          network := delete(p, network)
   output read(m, j)
     pre queue[j] ¹  Ø  /\  head(queue[j]).contents = m
     eff  queue[j] := tail(queue[j])
```

Figure 4.2. TIOA description of a multicast protocol

4.7.5. Using predicates to constrain effects

The results of a program in the effects clause can be constrained by a predicate relating the values of state variables after a transition has occurred to the values of state variables before the transition began. For example, the transition definition for the swap action in the Arrange automaton (see page 11) can be rewritten as follows:

```
transitions output swap(i, j: Nat)
  eff A[i] := choose;
      A[j] := choose
      ensuring A[i] = A[j] /\ A[j] = A[i]
```

The assignment statements indicate that the array A may be modified at indices i and j, and the **ensuring** clause constrains the modifications. A primed state variable in this clause (i.e., A') indicates the value of the variable in the post-state; an unprimed state variable (i.e., A) indicates its value in the pre-state. This notation allows us to eliminate the local variable temp needed previously for swapping.

There are important differences between **where** clauses attached to nondeterministic **choose** operators and **ensuring** clauses. A **where** clause restricts the value chosen by a **choose** operator in a single assignment statement, and variables appearing in the **where** clause denote values in the state before the assignment statement is executed. An **ensuring** clause can be attached only to an entire **eff** clause; unprimed variables appearing in an **ensuring** clause denote values in the state before the transition represented by the entire **eff** clause occurs, and primed variables denote values in the state after the transition has occurred.

TIOA assumes that state variables do not change value during a transition unless they occur on the lhs in an assignment statement. Therefore, nondeterministic **choose** statements such as the ones shown here give a transition defined with an **ensuring** clause license to change the values of the state variables mention in both the **choose** statements and the **ensuring** clause.

Editorial note: The following sections need to be expanded and polished. At present, they contain snippets of text extracted from other documents.

4.8. Trajectories

The dynamic type of a discrete variable is the set of piecewise continuous functions from real time to values in the static type of the variable. The dynamic type of an analog variable is the set of piecewise continuous functions from real time to values in the static type of the variable. In TIOA, variables of type Real are *analog*; all other variables are *discrete*.

A *trajectory* for a variable v describes the evolution of its value over time. Formally, it is a function τ_v in the dynamic type of v whose domain is an interval I of time starting with 0.

A *trajectory* for a set V of variables describes the evolution of each of their values over time. Formally, it is a function τ that assigns to each i in an interval I of time starting with 0 a valuation f_i of V such that, for any $v \in V$, the function τ_v defined by $\tau_v(i) = f_i(v)$ is a trajectory of v.

Trajectories are defined using invariants, algebraic and differential equations, and ``urgency'' conditions that specify when time must stop to allow a discrete action to occur.

Trajectories of an automaton are defined following the keyword **trajectories**. A trajectory definition consists of the keyword **trajdef** followed by a name, an invariant, an **evolve** clause, and a stopping condition. More than one trajectory definition can be used to define trajectories of an automaton. For example, the automaton Timeout in Figure~\ref{code:timeout} has two.

Each **trajdef** defines a set of trajectories; the set of all trajectories for an automaton is the concatenation closure of all of these sets (see~\cite{KLSV03b} for the definition of concatenation for trajectories).%SJG: We need to say more here. A trajectory belongs to the set of trajectories defined by a trajectory definition if it satisfies the predicate in its invariant clause, the differential equations in the evolve clause and the stopping condition expressed by the stop when clause. The stopping condition is satisfied by a trajectory if the only state in which the condition holds is the last state of that trajectory. In other words, time cannot advance once the stopping condition becomes true.

\caption{Example showing trajectory definitions}

The algorithm ClockSync is based on the exchange of physical clock values between different processes in the system. The parameter u determines the frequency of sending messages. Processes in the system are indexed by the elements of the type Index, which we assume to be pre-defined. ClockSync has a physical clock physclock, which may drift from the real time with a drift rate bounded by r. It uses the variable maxother to keep track of the largest physical clock value of the other processes in the system. The variable nextsend records when it is supposed to send its physical clock to the other processes. The logical clock, logclock, is defined to be the maximum of maxother and physclock. Formally, logclock is a *derived variable*, which is a function whose value is defined in terms of the state variables.

The unique trajectory definition in this example shows that the variable physclock drifts with a rate that is bounded by r. The periodic sending of physical clocks to other processes is enforced through the stopping condition in the trajectory specification. Time is not allowed to pass beyond the point where physclock = nextsend.

4.9. Operations on automata

The operation of *composition* allows an automaton representing a complex system to be constructed by composing automata representing individual system components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving an action π, so do all component automata that have π in their signatures.

The *hiding* operation ``hides'' output actions of an automaton by reclassifying them as internal actions; this prevents them from being used for further communication and means that they are no longer included in traces.

The *renaming* operation changes the names of an automaton's actions, to facilitate composing that automaton with others that were defined with different naming conventions. The TIOA language does not currently support this operation.

5. Properties of automata

5.1. Executions and traces

A simple mathematical object called a *trace*, which is essentially a sequence of actions interspersed with time-passage steps, defines the external behavior of a timed automaton.

An *execution fragment* of a timed automaton is a sequence τ_0, π_1, τ_1, π_2, \ldots of alternating trajectories τ_i and actions π_i such that, if τ_i is not the last trajectory in the sequence, then the domain of τ_i is a closed interval $[0, t_i]$ of time and $(\tau_i(t_i), \pi_{i+1}, \tau_{i+1}(0))$ is a transition of the automaton. An *execution* is an execution fragment such that $\tau_0(0)$ is an initial state. A state is *reachable* if it occurs in some execution.

If a program consists of more than a single assignment statement, then the states before and after the assignment statements in the program may be intermediate states, which do not appear in the execution fragments of the automaton.

The *trace* of an execution is the alternating sequence of external actions trajectories for the empty set of variables in that execution. Thus, the only information conveyed by these trajectories consists of their domains, that is, ob the amount of time that passes.

Editorial note: Give examples of traces for the alarm clock.

5.2. Invariants

An *invariant* of an automaton is a property that is true in all reachable states of the automaton.

Editorial note: Give examples.

5.3. Simulation relations

It is often useful to view timed systems at multiple levels of abstraction, starting with a high-level version that describes required properties, and ending with a low-level version that describes a detailed design or implementation. An automaton A is said to *implement* an automaton B provided that A and B have the same input and output actions and that every trace of A is also a trace of B. When A implements B, it might be more deterministic than $\B$, in terms of either discrete transitions or trajectories. For instance, $\B$ might be allowed to perform an output action at an arbitrary time before noon, whereas $\A$ guarantees that this action occurs between 10 and 11AM.

Editorial note: The rest of this section needs to be updated to describe simulation relations between timed automata, not just simulation relations between untimed automata.

The notion of a *simulation relation* between the states $\A$ and the states of $\B$ provides a sufficient condition for demonstrating that $\A$ implements $\B$. A simulation relation must satisfy three conditions, one relating initial states, one relating discrete transitions, and one relating trajectories of $\A$ and $\B$; loosely speaking, every initial state of A is related to an initial state of B and every reachable state of A is related to a state of B reached by the same series of external actions.

\caption{Forward simulation relation}

For the purpose of a formal definition, we assume that A and B have the same input and output actions. A relation R between the states of A and B is a *forward simulation* with respect to invariants I_A and I_B of A and B if and only if (as illustrated in Figure~\ref{fig:forwardSimulation}) \begin{itemize} \item every initial state of A is related (via R) to an initial state of B, and \item for all states s of A and u of B satisfying the invariants I_A and I_B such that $R(s, u)$, and for every step (s, π, s') of A, there is an execution fragment α of B starting with u that contains the same external actions as π and ends with a state u' such that $R(s', u')$. \end{itemize} A general theorem is that A implements B if there is a forward simulation from A to B.

Similarly, a relation R between the states of A and B is a *backward simulation* with respect to invariants I_A and I_B of A and B if \begin{itemize} \item every state of A that satisfies I_A corresponds (via R) to some state of B that satisfies I_B, \item if an initial state s of A is related (via R) to a state u of B that satisfies I_B, then u is an initial state of B, and \item for all states s, s' of A and u' of B satisfying the invariants such that $R(s', u')$, and for every step (s, π, s') of A, there is an execution fragment α of B ending with u' that contains the same external actions as π and that starts with a state u satisfying I_B such that $R(s, u)$. \end{itemize} \noindent Another general theorem is that A implements B if there is an *image-finite* backward simulation from A to B. Here, a relation R is image-finite provided that for any x there are only finitely many y such that $R(x, y)$. Moreover, the existence of any backward simulation from A to B implies that all finite traces of A are also traces of B.

6. Data types in TIOA

TIOA enables users to define the actions and states of I/O automata abstractly, using mathematical notations, without having to provide concrete representations for these abstractions. Some mathematical notations are built into TIOA; the user can define others.

Notations for the primitive data types Bool, Nat, Int, Real, AugmentedReal (which adds two elements, ∞ and $-\infty$ to the set of reals), Char, and String can appear in TIOA descriptions without explicit definition.

Notations for compound data types that result from using the following type constructors can also appear in TIOA descriptions without explicit definition. \begin{itemize} \item \mbox{}\ioa`Array[I1, `\ldots , In, E] is an n-dimensional array of elements of type E indexed by elements of types \ioa`I1`, \ldots, In. \item \mbox{}\ioa`Map[D1, `\ldots , Dn, R] is a finite partial mapping of elements of an n-dimensional domain with type $\ioa`D1` \times \cdots \times Dn$ to elements of a range with type R. Mappings differ from arrays in that they are defined only for finitely many elements of their domains (and hence may not be totally defined). \item \mbox{} Seq[E] is a finite sequence of elements of type E. \item \mbox{} Set[E] is a finite set of elements of type E. \item \mbox{} Mset[E] is a finite multiset of elements of type E. \item \mbox{} Null[E] is isomorphic to E extended by a single element nil. \end{itemize} %In this tutorial, we describe operators on the built-in data types informally %when they first appear in an example.

Users can introduce additional data types and type constructors by defining *vocabularies* for them. Each vocabulary introduces notations for a set of types and a set of operators. In fact, each of the built-in data types is defined by a built-in vocabulary. For example, the following built-in vocabularies provide notations for the Real data type and its associated operators. Each operator has a *signature* that specifies the types of its arguments and the type of its result. Infix, prefix, postfix, and mixfix operators are named by sequences of non-letter characters and are defined using placeholders __ to indicate the locations of their arguments. Operators used in functional notation (e.g., in $max(a, b)$) are named by simple identifiers.

```
vocabulary Real
  imports NumericOps(type Real, type Real, type Real)
   operators
    -__, abs: Real -> Real
    __**__: Real, Int -> Real
    int2real: Int -> Real

vocabulary NumericOps(T1, T2, T3: type)
  types T1 T2 T3
  operators
    __+__, __-__, __*__, __/__, min, max: T1, T2 -> T3
    __<__, __<=__, __>__, __>=__, __=__, __~=__: T1, T2 -> Bool
```

As these examples illustrate, a vocabulary can import notations from other vocabularies, and it can be parameterized to make operator notations such as __<__:Real,Real® Bool available for the Real data type.

A vocabulary can define a type constructor, as in the following built-in vocabulary for the Null constructor

```
vocabulary Null defines Null[T]
  operators
    nil : ® Null[T]
    embed : T ® Null[T]
    __.val : Null[T] ® T
```

The identifier T in this vocabulary is a type parameter, which is instantiated any time the constructor Null is used to provide operator notations appropriate for that use. Thus, if x is a variable of type Null[Int], then one can write embed(x).val = x.

User-defined vocabularies can introduce notations for enumeration, tuple, and union types analogous to those found in many common programming languages. For example,

```
vocabulary sampleVocab
  types Color enumeration [red, white, blue],
      Msg   tuple [source, dest: Process, contents: String],
      Fig   union [sq: Square, circ: Circle]
```

can be imported by the definition of any other vocabulary or automaton to provide notations for three data types it describes.

In this tutorial, some operators are displayed using mathematical symbols that do not appear on the standard keyboard. Table~\ref{table:symbols} shows the input conventions for entering these symbols.

Logical Operator		
Symbol	**Meaning**	**Input**

"	Forall	\A
$	There exists	\E
Ø	Not	~
¹	Not equals	~=
Ù	And	/\
Ú	Or	\/
Þ	Implies	=>
Û	If and only if	<=>
	Operator Signature	
Symbol	**Meaning**	**Input**
®	Domain ® Range	->
	Datatype Operator	
Symbol	**Meaning**	**Input**
£	Less than or equal	<=
³	Greater than or equal	>=
Î	Member of	\in
Ï	Not a member of	\not \in
?	Proper subset of	\subset
Í	Subset of	\subseteq
É	Proper superset of	\supset
Ê	Superset of	\supseteq
?	Append element	|-
?	Prepend element	-|
8	Infinity	\infty
?	Bottom element	\bot
?	Top element	\top
Ø	Empty set or sequence	{}

\caption{Typographical conventions}

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicolin, A. Olivero, J. Sifakis, and S. Yovine, “The algorithmic analysis of hybrid systems,” *Theoretical Computer Science*, 138:3-34, 1995.

[2] Stephen J. Garland, Nancy A. Lynch, Joshua A. Tauber, Mandan Vaziri, *IOA User Guide and Reference Manual*, MIT Computer Science and Artificial Intelligence Laboratory}, Cambridge, MA, 2003. Available at http://theory.csail.mit.edu/tds/ioa/manual.ps.

[3] D. Kaynar, N. Lynch, R. Segala, and F.Vaandrager, “Timed I/O automata: a mathematical framework for modeling and analyzing real-time systems,” *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, 2003, pages 166-177. IEEE Computer Society. Full version available as Technical Report MIT/LCS/TR-917.

[4] Dilsun Kaynar, Nancy Lynch, Sayan Mitra, Stephen Garland, *The TIOA Language, Version 0.21*, MIT Computer Science and Artificial Intelligence Laboratory, unpublished manuscript, May 22, 2005.

[5] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager, “The theory of timed I/O automata,” Technical Report MIT/LCS/TR-917a, MIT Computer Science and Artificial Intelligence Laboratory, 2004. Available at http://theory.csail.mit.edu/tds/reflist.html.

[6] N. A. Lynch, R. Segala, and F. W. Vaandrager, “Hybrid I/O automata,” *Information and Computation*, 185(1):105-157, 2003. Also Technical Report MIT-LCS-TR-827d, MIT Laboratory for Computer Science.

[7] Sayan Mitra, “HIOA+: Specification language and proof tools for hybrid systems, unpublished manuscript.

[8] Sayan Mitra, Yong Wang, Nancy Lynch, and Eric Feron, “Safety verification of model helicopter controller using hybrid input/output automata, *HSCC'03, Hybrid System: Computation Control*, Prague, the Czech Republic, April 3-5, 2003.