TIOA Tutorial

Stephen J. Garland, Dilsun Kaynar, Nancy A. Lynch,
Joshua A. Tauber, and Mandana Vaziri
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

May 22, 2005

The timed input/output (TIOA) modeling framework [2, 3] is a mathematical framework that supports the description and analysis of timed systems. The TIOA language is a modeling language that provides notations for describing timed I/O automata precisely.

The TIOA language is a variant of the IOA language [1], which can be used to describe basic I/O automata with no timing information. TIOA extends and formalizes the descriptive notations used in [2, 3] and supports a variety of analytic tools. These tools range from light-weight tools, which check the syntax of automaton descriptions, to medium-weight tools, which simulate the action of an automaton, and to heavier-weight tools, which provide support for proving properties of automata.

1 Introduction

Timed input/ouput automata provide a mathematical model suitable for describing time-dependent behavior in concurrent systems. The model provides a precise way of describing and reasoning about system components that interact with each other through discrete actions as well as the continuous evolution of internal state components over time.

2 Timed input/output automata

The fundamental object in the TIOA framework is a *timed (I/O) automaton*, which is a kind of nondeterministic, possibly infinite-state, state machine. The state of a timed automaton is described by a valuation of state variables that are internal to the automaton. The state of a timed automaton can change in two ways: instantaneously, by the occurrence of a *discrete transition*, or over an interval of time via a *trajectory*, which is a function that describes the evolution of the state variables. Trajectories may be continuous or discontinuous functions.

TIOA transitions are associated with named *actions*, which are classified as *input*, *output*, or *internal.* Input and output actions are used for communication with the automaton's environment, whereas internal actions are visible only to the automaton itself. The input actions are assumed not to be under the automaton's control, whereas the automaton itself controls which output and internal actions should be performed.

The communication of a timed automaton with its environment is limited to discrete transitions associated with actions shared between the automaton and its environment. The TIOA framework does not model continuous information flow between a timed automaton and its environment; this kind of modeling would require a ful-fledged hybrid automaton model [4].

The time domain in TIOA is the set of real numbers (in [3] additional time domains are considered). States of automata consist of valuations of *variables.* Each variable has both a *static type*, which defines the set of values it may assume, and a *dynamic type*, which gives the set of trajectories it may follow. We assume that dynamic types are closed under some simple operations: shifting the time domain, taking subintervals, and pasting together intervals.

A *trajectory* for a set V of variables describes the evolution of the variables in V over time; formally, it is a function from a time interval that starts with 0 to valuations of V; that is, a trajectory defines a value for each variable at each time in the interval.

A typical example of a timed automaton is a time-bounded FIFO message channel.

Formally, a *timed (I/O) automaton A* consists of the following six components:

- A set X of *internal variables.*

- A set Q, which is a subset of all possible valuations of X.

- A set of initial states, which is a non-empty subset of the set of all states.

- A signature, which lists disjoint sets of input, output, and internal actions of A.

- A discrete transition relation, which contains triples of the form (state, action, state), and

- A set of trajectories for X such that $\tau(t) \in Q$ for every $\tau \in \mathcal{T}$ and every t in the domain of τ.

Dilsun: Do we need to call the variables "internal"? Is Q the set of states? Why doesn't it contain all valuations?

An action π is said to be *enabled* in a state s if there is another state s' such that (s, π, s') is a transition of the automaton. Input actions are enabled in every state; i.e., automata are not able to "block" input actions from occurring. The *external* actions of an automaton consist of its input and output actions.

The transition relation is usually described in *precondition-effect* style, which groups together all transitions that involve a particular type of action into a single piece of code. The precondition is a predicate (that is, a boolean-valued expression) on the state indicating the conditions under which the action is permitted to occur. The effect 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). Actions are executed indivisibly.

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.

3 Using TIOA to formalize descriptions of timed I/O automata

We illustrate the nature of timed automata, as well as the use of TIOA to define the automata, by a few simple examples. Figure 3.1 contains a simple TIOA description for an automaton, Timeout(u:Real, M:Type), that awaits the receipt of a message of type M from another process. If u time units elapse without such a message arriving, the automaton performs a timeout action, thereby "suspecting" the other process. When a message arrives, it "unsuspects" the other process. The automaton may suspect and unsuspect repeatedly. The automaton is parameterized by the timeout period u and the type M of the messages received by Timeout(u:Real, M: Type).

The automaton Timeout has two state variables: suspected is a boolean that is set to true when a timeout occurs, and clock is a real number that represents a timer running at the same speed as realtime. The initial value of suspected and clock are \ioafalse' and 0. The value of the automaton parameter u is constrained to be strictly greater than 0.

The transitions of the automaton Timeout are given in precondition/effect style. The input action receive (m) has no precondition, which is equivalent to having true as its 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 receive is to reset clock to 0 and to set suspected to false (in case it had been true before). The output action timeout can occur only when it is enabled, that is, only in states in which suspected is false and clock =u. Its effect is to set suspected to true.

```
automaton Timeout(u: Real, M: type) where u > 0
  signature
    input receive(m: M)
    output timeout
  states
    suspected: Bool := false,
    clock Real := 0
  transitions
    input receive(m)
      eff clock := 0;
          suspected:= false
    output timeout
      pre  ¬suspected ∧ clock = u
      eff  suspected := true
  trajectories
    trajdef suspected
      invariant suspected
      evolve  d(clock) = 1
    trajdef notsuspected
      invariant ¬suspected
      stop when clock = u
      evolve d(clock) = 1
```

Figure 3.1: TIOA description of a timeout process

The two trajectory definitions suspected and notsuspected correspond to two "modes" of the Timeout automaton. While the suspected flag is false, the clock advances with rate 1, that is, with the same rate as realtime, and time cannot go beyond the point at which clock =u. While the suspected flag is true there is no condition on time-passage; the clock may keep advancing with rate 1. Note that trajectories do not need to be followed until a stopping condition is reached; however, if a stopping condition is reached then time must stop. At this point, a discrete action may occur if it is enabled.

4 Data types in TIOA descriptions

[[Have to incorporate material on dynamic types, AugmentedReal etc.]]

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

The data types Bool, Int, Nat, Real, Char, and String can appear in TIOA descriptions without explicit definition. Compound data types can be constructed using the following type constructors and used without explicit definition.

- Array[I1, ... , In, E] is an n-dimensional array of elements of type E indexed by elements of types I1, ..., In.
- Map[D1, ... , Dn, R] is a finite partial mapping of elements of an n-dimensional domain with type D1 × ⋯ × 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).
- Seq[E] is a finite sequence of elements of type E.
- Set[E] is a finite set of elements of type E.
- Mset[E] is a finite multiset of elements of type E.
- Null[E] is isomorphic to E extended by a single element nil.

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 provides 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
\end(IOA}

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

A vocabulary can define a type constructor, as in the following built-in
vocabulary for the \ioa'Null' constructor.
\begin{IOA}
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 4.1 shows the input conventions for entering these symbols.

5 TIOA descriptions for primitive automata

Explicit descriptions of primitive automata specify their names, action signatures, state variables, transition relations, and trajectories. All but the last of these elements must be present in every primitive automaton description.

Logical Operator			Datatype Operator		
Symbol	Meaning	Input	Symbol	Meaning	Input
$\forall$	For all	`\A`	$\leq$	Less than or equal	`<=`
$\exists$	There exists	`\E`	$\geq$	Greater than or equal	`>=`
$\neg$	Not	`~`	$\in$	Member of	`\in`
$\neq$	Not equals	`~=`	$\notin$	Not a member of	`\notin`
$\wedge$	And	`/\`	$\subset$	Proper subset of	`\subset`
$\vee$	Or	`\/`	$\subseteq$	Subset of	`\subseteq`
$\Rightarrow$	Implies	`=>`	$\supset$	Proper superset of	`\supset`
$\Leftrightarrow$	If and only if	`<=>`	$\supseteq$	Superset of	`\supseteq`
			$\vdash$	Append element	`|-`
			$\dashv$	Prepend element	`-|`

Table 4.1: Typographical conventions

5.1 Automaton names and parameters

The first line of an automaton description consists of the keyword **automaton** followed by the name of the automaton. As illustrated in Figure 3.1, the name may be followed by a list of formal parameters enclosed within parentheses. There are two kinds of automaton parameters. An individual parameter (such as u:Real) consists of an identifier with its associated type, and it denotes a fixed element of that type. A type parameter (such as M:**type**) consists of an identifier followed by the keyword **type**, and it denotes a type.Example of nondeterministic choice of initial value for state variable

5.2 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. A varying parameter consists of an identifier with its associated type, and it denotes an arbitrary element of that type. A fixed parameter consists of the keyword **const** followed by a term denoting a 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.

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. Such constraints restrict the set of actions denoted by the entry.

5.3 State variables

As in the above examples, state variables are declared using the keyword **states** followed by a comma-separated list of state variables and their types. State variables can be initialized using the assignment operator := followed either by an expression or by a nondeterministic choice. The order in which state variables are declared makes no difference: state variables are initialized simultaneously, and their initial values cannot refer to the value of any state variable.

A nondeterministic choice (of the form **choose** *variable* **where** *predicate*) selects an arbitrary value of the variable that satisfies the predicate. When a nondeterministic choice is used to initialize a state variable, there must be some value of the variable that satisfies the predicate. 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.

```
automaton Choice
  signature output result(i: Int)
  states num: Int := choose n where 1 ≤ n ∧ n ≤ 3
  transitions
    output result(i)
      pre i = num
```

Figure 5.2: Example of nondeterministic choice of initial value for state variable

For example, in the automaton Choice (Figure 5.2), the state variable num is initialized nondeterministically to some value n that satisfies the predicate $1 \leq n \land n \leq 3$, i.e., to one of the values 1, 2, or 3 (the value of n must be an integer because it is constrained to have the same type, Int, as the variable num to which it will be assigned).

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), as illustrated by the definition of the automaton Shuffle in Figure 5.3.[1]

In Figure 5.3, the initial values of the variable cut and the array name of strings are constrained so that name[1], . . . , name[52] are sorted in two pieces, each in increasing order, with the piece after the cut containing smaller elements than the piece before the cut. The constraint following **initially** limits only the initial values of the state variables, not their subsequent values. (Note that the scope of the **initially** clause is the entire set of state variable declarations.) The type Array[cardIndex, String] of the state variable name consists of arrays indexed by elements of type cardIndex and containing elements of type String (see Section 4). The swap actions transpose pairs of strings until a deal action announces the contents of the array; then no further actions occur.

When the type of a state variable is an Array, Map, or **tuple** (Section 4), TIOA also treats the elements of the array or mapping, or the fields in the tuple, as state variables, to which values can be assigned without affecting the values of the other elements in the array or mapping or of the fields in the tuple.

[1] At present, users must expand the . . . in the definition of the type cardIndex by hand. In the future, TIOA may provide more convenient notations for integer subranges.

```
vocabulary cardDeck
  types cardIndex enumeration [1, 2, 3, ..., 52]

automaton Shuffle
  imports cardDeck
  signature
    internal swap(i, j: cardIndex)
    output   deal(a: Array[cardIndex, String])
  states
    dealt: Bool := false,
    name:  Array[cardIndex, String],
    cut:   cardIndex
    initially
      ∀ i: cardIndex (i ≠ 52 ∧ i ≠ cut ⇒ name[i] < name[succ(i)])
              ∧ name[52] < name[1]
  transitions
    internal swap(i, j; local temp: String)
      pre ¬dealt
      eff temp := name[i];
          name[i] := name[j];
          name[j] := temp
    output deal(a)
      pre ¬dealt ∧ a = name
      eff dealt := true
```

Figure 5.3: Example of a constraint on initial values for state variables

5.4 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, an optional **where** clause, an optional precondition, and an optional effect.

More than one transition definition can be given for an entry in an automaton's signature. For example, the transition definition for the swap actions in the Shuffle automaton (Figure 5.3) can be split into two parts:

```
internal swap(i, j; local temp: String) where i ≠ j
  pre ¬dealt
  eff temp := name[i];
      name[i] := name[j];
      name[j] := temp
internal swap(i, i)
  pre ¬dealt
```

The second of these two transition definitions does not change the state, because it has no **eff** clause.

5.5 Transition parameters

Two kinds of parameters can be specified for a transition: ordinary parameters, corresponding to those in the automaton's signature, and additional local parameters. The ordinary parameters accompanying an action name in a transition definition must match those accompanying the action name in the automaton's signature, both in number and in type. However, the keyword **const** does not appear in transition parameters, and all transition parameters are treated as terms.

The simplest way to formulate the ordinary parameters for an action in a transition definition is to erase the keyword **const** and the type modifiers from the parameter list in the signature;

In addition to these ordinary parameters, a transition definition can contain *local variables*, which are specified after the ordinary parameters and the keyword **local**. Local variables can be used for two purposes. As illustrated in Figure 5.3, they can be used as temporary state variables. In addition, they can be used to relate the postcondition for a transition to its precondition.

5.6 Preconditions

A precondition can be defined for a transition of an output or internal action using the keyword **pre** followed by a predicate. Preconditions cannot be defined for transitions of input actions. All variables in a precondition must be parameters of the automaton, be state or local variables, appear in parameters for the transition definition, or be quantified explicitly in the precondition. If no precondition is given, it is assumed to be true.

An action is said to be *enabled* in a state if there are some values for the local variables of one of its transition definitions that satisfy both the **where** clause and, together with the state variables, the precondition for that transition definition. Input actions, whose transitions have no preconditions, are always enabled.

5.7 Effects

The effect of a transition, if any, is defined following the keyword **eff**. This effect is generally defined in terms of a (possibly nondeterministic) program that assigns new values to state variables. The amount of nondeterminism in a transition can 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.

If the effect is missing, then the transition has none; i.e., it leaves the state unchanged.

5.7.1 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.

Assignment statements An assignment statement changes the value of a state or local variable. The statement consists of the state or local variable followed by the assignment operator $:=$ and either an expression or a nondeterministic choice (indicated by the keyword **choose**). As noted in Section 5.3, the elements in an array or mapping, or the fields in a tuple, used as a state or local variable, are themselves considered as separate variables and can appear on the left side of the assignment operator.

The expression or nondeterministic choice in an assignment statement must have the same type as the state or local variable. The value of the expression is defined mathematically, rather than computationally, in the state before the assignment statement is executed.[2] The value of the expression then becomes the value of the state or local variable in the state after the assignment statement is executed. 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.

Conditional statements A conditional statement is used to select which of several program segments to execute in a larger program. It starts with the keyword **if** followed by a predicate, the keyword **then**, and a program segment; it ends with the keyword **fi**. In between, there can be any number of **elseif** clauses (each of which contains a predicate, the keyword **then**, and a program segment), and there can be a final **else** clause (which also contains a program segment).

For ***statements*** A **for** statement is used to perform 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, the keyword **do**, a program segment, and the keyword **od**.

[2] 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.

```
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] := {},
    queue:    Array[Node, Seq[Packet]]
    initially ∀ i: Node (queue[i] = {})
  transitions
    input mcast(m, i, I)
      eff network := insert([m, i, I], network)
    internal deliver(p)
      pre p ∈ 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] ≠ {} ∧ head(queue[j]).contents = m
      eff queue[j] := tail(queue[j])
```

Figure 5.4: Example showing use of a **for** statement

Figure 5.4 illustrates the use of a **for** statement in a high-level description of a multicast algorithm that has no timing constraints. Its first line defines the Packet data type to consist of triples [contents, source, dest], in which contents represents a message, source the Node from which the message originated, and dest 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 notation [m, i, I] is defined by the tuple data type (Section 4) and the insert operator by the multiset data type (Section 4). 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 node in the destination set 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 the queue of delivered packets at that Node.

There are two ways to describe the set of values for the control variable in a **for** statement. The first 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 following the keyword **do** 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. However, TIOA requires that the effect of a **for** statement be independent of the order in which executions of its program occur.

Using predicates on states to specify effects The results of a program 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. Such a predicate is particularly useful when the program contains the nondeterministic **choose** operator. For example,

```
eff name[i] := choose;
    name[j] := choose
    ensuring name'[i] = name[j] ∧ name'[j] = name[i]
```

is an alternative way of writing the effect clause of the swap action in Shuffle (Figure 5.3). The assignment statements indicate that the array name may be modified at indices i and j, and the **ensuring** clause constrains the modifications. A primed state variable in this clause (i.e., name′) indicates the value of the variable in the post-state; an unprimed state variable (i.e., name) 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 those attached to **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 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.

5.8 Trajectories

Trajectories of an automaton are defined following the keyword **trajectories**. A trajectroy definition consists of 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 3.1 has two trajectory definitions.

Each trajectory definition defines a set of trajectories; the set of all trajectories for an automaton is the concatenation closure of all of these sets (see [3] for the definition of concetanation for trajectories).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.

```
automaton ClockSync(u,r: Real, i: Index)
  signature
     input receive(m: Real, j: Index, const i: Index) where j ≠ i,
     output send(m: Real, const i: Index)
  states
     nextsend: Real := 0,
     maxother: Real := 0,
     physclock: Real := 0
     initially u > 0 ∧ (0 ≤ r < 1)

  let logclock = max(maxother, physclock)

  transitions
     input  receive(m,j,i)
       eff maxother := max(maxother,m)
     output send(m,i)
       pre m = physclock ∧ physclock = nextsend
       eff nextsend := nextsend + u
  trajectories
     trajdef always
        stop when physclock = nextsend
        evolve (1 − r) ≤ d(physclock) ≤ (1 + r)
```

Figure 5.5: 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.

References

[1] S. Garland, N. Lynch, and M. Vaziri. *IOA: A Language for Specifying, Programming, and Validating Distributed Systems.* MIT Laboratory for Computer Science, Cambridge, MA, 2001. Available at `http://theory.lcs.mit.edu/tds/ioa.html`.

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

[3] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. Technical Report MIT/LCS/TR-917a, MIT Laboratory for Computer Science, 2004. Available at `http://theory.lcs.mit.edu/tds/reflist.html`.

[4] 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.