# Tempo: A Toolkit for The Timed Input/Output Automata Formalism

N. Lynch
CSAIL, MIT
The Stata Center, Building 32
32 Vassar Street
Cambridge, MA 02139
lynch@theory.csail.mit.edu

L. Michel
CSE, University of Connecticut
371 Fairfield Way
Storrs, CT 06269-2155
ldm@engr.uconn.edu

A.A. Shvartsman
CSE, University of Connecticut
371 Fairfield Way
Storrs, CT 06269-2155
aas@engr.uconn.edu

## ABSTRACT

Tempo is a simple formal language for modeling distributed, concurrent, and timed systems as collections of interacting state machines, called timed input/output automata. Tempo provides natural mathematical notations for describing systems, their intended properties, and intended relationships between their descriptions at varying levels of abstraction. The Tempo Toolkit is an implementation of the Tempo language and a suite of tools that supports a range of validation methods for descriptions of systems and their properties, including static analysis, simulation, and machine-checked proofs. This paper gives a brief overview of the Tempo language and illustrates its utility on selected examples of importance to distributed computing. The focus of the presentation is on the Tempo tools, and in particular, the simulator.

## Categories and Subject Descriptors

F.4.3 [**Formal Languages**]: Tempo; D.3 [**Programming Languages**]: Application; D.2.4 [**Software Engineering**]: Program Verification; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

## General Terms

Input Output Automata, Timed Input Output Automata, Distributed Algorithms, Specification, Verification

## 1. INTRODUCTION

Tempo is a simple formal language for modeling distributed systems as collections of interacting state machines called *Timed Input/Output Automata* [3]. Timed Input/Output Automata are often referred to as *Timed I/O Automata*, or just *TIOAs*. The distributed systems in question may have timing constraints, for example, bounds on the time when certain events may occur, or bounds on the rates of change of component clocks. They may use time in significant ways, for example, for timeouts, or for scheduling events to occur periodically. Timed I/O Automata formalism provides good support for describing these constraints and capabilities. Timed and untimed I/O Automata formalisms have been effectively used for specifying numerous distributed and concurrent algorithms [6]. The Tempo language provides simple formal notation for describing Timed I/O Automata precisely, based on the pseudocode notation that has been used in many research papers. It also allows specification of properties such as invariant assertions and relationships between automata at different levels of abstraction. The Tempo language is supported by an associated integrated development environment toolkit, also called Tempo, that provides an extensible framework supporting a range of integrated analysis and validation tools, including static analysis, simulation, model-checking, and theorem-proving. Additional tools under consideration include optimization of distributed deployment of systems specified in Tempo, and generation of distributed code from specifications.

Many distributed systems involve a combination of computer components and real-world, physical entities such as vehicles, robots, or medical devices. Systems involving interaction between computer and real-world components usually have strong safety, reliability, and predictability requirements, stemming from the requirements of real-world applications. This makes it especially important to have good methods for modeling the systems precisely and analyzing their behavior rigorously. Tempo provides a simple, elegant, and powerful mathematical foundation for analyzing a wide variety of systems, and it can be used to model both computer and real-world system components, as well as their interactions.

Tempo can be used to model practically any type of distributed system, including (wired and wireless) communication systems, real-time operating systems, embedded systems, automated process control systems, and even biological systems. The behavior of these systems generally includes both discrete state changes and continuous state evolution; Tempo is designed to express both kinds of changes.

The Tempo Toolkit was developed by VeroModo Inc., with support provided by an AFOSR technology transfer grant. The beta releases of the Tempo Toolkit for Linux, Windows, and Mac OS X platforms are available for download at www.veromodo.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SimulationWorks 2008, March 04, Marseille, France
Copyright © 2008 ICST 978-963-9799-20-2
DOI 10.4108/ICST.SIMUTOOLS2008.3105

Earlier work on a toolkit supporting specification in (untimed) Input/Output Automata was performed at the MIT Theory of Distributed Systems group [2]. The prototype toolkit supported a simulator [1], paired automata simulation [8], and simulations of composed automata [9].

In the rest of this paper we overview the Timed I/O Automata, the Tempo language, and toolset (Section 2), illustrate the capabilities of Tempo and its simulator (Sections 3 and 4), and describe the user interface of the Tempo integrated development environment (Section 5).

# 2. TEMPO OVERVIEW

We now discuss the Timed I/O Automata formalism that is the basis of the Tempo language, and summarize the capabilities of the Tempo toolkit.

## 2.1 Timed I/O Automata

The Timed I/O Automata [3] mathematical framework is an extension of the classical I/O Automata framework [5, 6], which for many years has been successfully used in the theoretical distributed computing research community to specify and reason about distributed and concurrent algorithms. I/O Automata are very simple interacting asynchronous state machines, without any support for describing timing features. Although they are simple, I/O Automata provide a rich set of capabilities for modeling and analyzing distributed algorithms. I/O Automata support description of many properties that distributed algorithms are required to satisfy, and mathematical proofs that the algorithms in fact satisfy their required properties. These proofs are based on methods such as invariant assertions and compositional reasoning. I/O Automata also support representation of algorithms at different levels of abstraction, and proofs of consistency relationships between algorithm representations at different levels. Because of these capabilities, I/O Automata have been used fairly extensively for modeling and analyzing asynchronous distributed algorithms, and even for proving impossibility results about computability in asynchronous distributed settings.

However, ordinary I/O Automata cannot be used to describe distributed algorithms that use time explicitly, for example, those that use timeouts or schedule events periodically. And they do not provide explicit support for describing timing constraints such as bounds on message delay or clock rates. Moreover, without support for timing, I/O Automata could not be used for other applications such as practical communication protocols. These limitations led to the development of Timed I/O Automata, which include new features—most notably, *trajectories*—specifically designed for describing timing aspects of systems.

Like ordinary I/O Automata, Timed I/O Automata are simple interacting state machines and have a well-developed, elegant theory, presented in [3]. Like I/O Automata, Timed I/O Automata provide a rich set of capabilities for system modeling and analysis. Methods used for analyzing timed I/O automata are essentially the same as those used for ordinary I/O automata: invariant assertions, compositional reasoning, and correspondences between levels of abstraction.

## 2.2 The Tempo language and tools

I/O Automata and Timed I/O Automata are fine mathematical modeling frameworks for distributed systems and have been used, by hand, to describe and analyze distributed algorithms, communication protocols, and embedded systems. Yet, computer support could make these tasks quite a bit easier. The Tempo Language and Toolkit is an attempt at providing a broad set of tools to support these activities.

The Tempo toolkit contains tools to support analysis of systems. These include a compiler that checks syntax and perform static semantic analysis; a simulator to produce and explore execution traces for an automaton; a translation module to the UPPAAL model-checker [4]; and a translation module to the PVS interactive theorem-prover [7]. The overall architecture of the Tempo toolkit has been designed to facilitate incorporation of other validation tools in the future.

The Tempo language has a rather minimal syntax, which closely matches the simple semantics of the Timed I/O Automata mathematical framework. In fact, the mapping between a Tempo automaton description and the Timed I/O Automata that it denotes is pretty transparent. For example, an automaton's discrete transitions and continuous evolutions are described directly in Tempo, by "transitions" and "trajectories", respectively. The minimality of the Tempo language does not limit its expressive power: Tempo is capable of describing very general systems of Timed I/O Automata. Of course, many analysis tools—especially automated ones like model-checkers—are not capable of handling fully general Tempo programs. In contrast with the conventional approach taken by developers of automated tools, Tempo does not outright limit the expressive power of the language and opts instead for the definition of *sublanguages* that are suitable for use with particular tools.

# 3. THE DRIVING EXAMPLE.

To illustrate the capabilities of Tempo and its simulator, we will be using the Fischer Timed Mutual Exclusion Algorithm. It has become famous as a standard test example for formal methods for modeling and analyzing timed systems. An informal description of the example appears in [6], Chapter 24.

## 3.1 The Tempo specification

This example illustrates most of the basic constructs needed for writing a Tempo program for a single Timed I/O Automaton modeling a shared-memory system. The example also demonstrates how to express invariants using Tempo, including invariants that involve time.

The Tempo model shown in Code 1,2 describes the entire system as a single Timed I/O Automaton. The **vocabulary** section declares the data types used in the algorithm, namely, the abstract data type *process* and the program counter abstract data type *PcValue* (an enumerated type) to represent the exact location of each process in its program. Each process could be in its *remainder region* (program counter = *pc_rem*), where it is not engaged in trying to enter the critical region. Or, it could be about to test, set, or check the *turn* variable. Or, it could be in various stages of entering or leaving the critical region—the model uses separate program

```
vocabulary fischer_types
  types process,
  PcValue : Enumeration [pc_rem, pc_test, pc_set, pc_check,
            pc_leavetry, pc_crit, pc_reset, pc_leaveexit]
end

automaton fischer(l_check, u_set: Real)
  where u_set < l_check ∧u_set ≥0 ∧l_check ≥0
  imports fischer_types

  signature
    output try(i: process)
    output crit(i: process)
    output exit(i: process)
    output rem(i: process)
    internal test(i: process)
    internal set(i: process)
    internal check(i: process)
    internal reset(i: process)

  states
    turn: Null[process] := nil;
    pc: Array[process, PcValue] := constant(pc_rem);
    now: Real := 0;
    last_set: Array[process, AugmentedReal] := constant(∞);
    first_check: Array[process, DiscreteReal] := constant(0);

  transitions
    output try(i)
      pre pc[i] =pc_rem;
      eff pc[i] := pc_test;
    internal test(i)
      pre pc[i] =pc_test;
      eff if turn =nil then
            pc[i] := pc_set;
            last_set[i] := (now + u_set);
          fi;
    internal set(i)
      pre pc[i] =pc_set;
      eff turn := embed(i);
          pc[i] := pc_check;
          last_set[i] := ∞;
          first_check[i] := now + l_check;
```

**Code 1: Tempo spec. of the Fischer algorithm (I)**

```
    internal check(i)
      pre pc[i] =pc_check ∧first_check[i] ≤now;
      eff if turn =embed(i) then
            pc[i] := pc_leavetry;
          else
            pc[i] := pc_test;
          fi;
          first_check[i] := 0;
    output crit(i)
      pre pc[i] =pc_leavetry;
      eff pc[i] := pc_crit;
    output exit(i)
      pre pc[i] =pc_crit;
      eff pc[i] := pc_reset;
    internal reset(i)
      pre pc[i] =pc_reset;
      eff pc[i] := pc_leaveexit;
          turn := nil;
    output rem(i)
      pre pc[i] =pc_leaveexit;
      eff pc[i] := pc_rem;
  trajectories
    trajdef traj
      stop when
        ∃i: process (now =last_set[i]);
      evolve
        d(now) =1;
```

**Code 2: Tempo spec. of the Fischer algorithm (II)**

counter values to represent situations where the process has successfully completed the trying protocol, where it is actually in the critical region, where it is about to reset the *turn* variable upon leaving, and where it has successfully completed the exit protocol.

The actual automaton description begins with the name of the automaton, with formal parameters *l_check* and *u_set*. These are real numbers representing, respectively, a lower bound on the time between setting and checking, and an upper bound on the time between checking and setting. The **where** clause specifies restrictions imposed on the parameters saying (most importantly) that *u_set* must be strictly less than *l_check*. The automaton **imports** the vocabulary to make its definition available to the remainder of the specification.

The automaton's **signature**, describe its actions. Actions are classified as **input**, **output**, or **internal**. Here, no input actions are used, i.e., the system is "closed". Since the entire system is being modelled by a single automaton, each type of action is parameterized by the name of the process that performs it. In this model, the internal actions are associated with shared-variable accesses—the steps that test, set, check, and reset the *turn* variable. The output actions are those that mark processes' progress through the various high-level regions of their code: The *try*($i$) action describes process $i$ moving from its remainder region to its *trying region*, in which it executes a protocol to try to reach the critical region. The *crit*($i$) action describes passage from the trying region to the critical region, and the *exit*($i$) action describes passage from the critical region to the *exit region*, where process $i$ performs its exit protocol. Finally, the *rem*($i$) action describes passage from the exit region back to the remainder region.

The automaton's state is specified in the **states** section. The shared variable *turn* has type **Null**[*process*], which indicates that its value can either be a process or the special value *nil* to indicate the absence of value. *turn* is initially set to *nil*. The variable *pc*, represents the program counters for all of the processes in an array of *PcValue* indexed by processes. Initially, all of the program counter values are set to *pc_rem*, which means that all of the processes start out in the remainder region.

The remaining three variables are introduced solely to express the needed timing constraints. First, the variable *now* is used to represent the real time. It is initialized at 0.

Second, the variable *last_set* is an array containing absolute real time upper bounds (*deadlines*) for the processes to perform **set** actions. A deadline will be in force for a process $i$ only when its program counter is equal to *pc_set*, that is, when it is in fact ready to set the *turn* variable. In this case, the value of *last_set*[$i$] will be a nonnegative real number; oth-

erwise, that is, if the program counter is anything other than *pc_set*, the value will be $\infty$, representing the absence of any such deadline. The elements of the *last_set* array are defined to be of type *AugmentedReal*: a type that includes all (positive and negative) real numbers, plus two values corresponding to positive and negative infinity. Initially, since none of the program counters is *pc_set*, the values in the array are all $\infty$.

Third and finally, the variable *first_check* is an array containing absolute real time lower bounds (*earliest times*) for the processes to perform *check* actions, when their program counters are equal to *pc_check*. The elements of *first_check* are of type *DiscreteReal*, which means that they always have *Real* values, and moreover, they do not change between discrete actions.

The detailed description of the transitions of the automaton follows in the **transitions** section. Transitions are (state, action, state) triples. The transitions are described in *guarded command* style, using small pieces of code called *transition definitions*. Each transition definition denotes a collection of transitions, all of which share a common action name.

Each transition has a name, list of parameters, a *precondition* that indicates when the action is enabled and finally, an effect clause that describes the changes to the state when that accompany the action. Input actions are always enabled, reflecting the assumption that Timed I/O Automata are *input-enabled*. Notionally, input actions have no preconditions, as a shorthand for the precondition being true.

The *try*($i$) transition represents an entrance by process $i$ into its trying region. The transition is allowed to occur whenever $pc[i] = pc\_rem$, that is, whenever process $i$ is in its remainder region. The effect is simply to advance the program counter to *pc_test* to indicate that process $i$ is ready to test the *turn* variable.

The *test*($i$) transition represents process $i$ testing the *turn* variable. It is allowed to occur whenever $pc[i] = pc\_test$. The transition can either find the *turn* variable equal to *nil* at which point it moves to take the turn (by setting the program counter to *pc_set*) and saves in $last\_set[i]$ the deadline for the **set** action to occur at the latest in $u\_set$ time steps in the future (away from *now*). The transition can also find that *turn* is not *nil* and simply takes no action to remain in the state, ready to test again.

The **set**($i$) transition represents process $i$ setting the *turn* variable to its own index. This is allowed to occur whenever $pc[i] = pc\_set$. The effects are given as straight-line code in which process $i$ simply sets *turn* to its own index (the *embed* call is necessary to store the value into an object of type **Null**[*process*]). The code then sets the program counter to *pc_check* to enable the *check*($i$) transition that will verify the *turn* variable. Now that the **set**($i$) action has occurred, the $last\_set[i]$ deadline is reset to its default value, $\infty$. The code also records the earliest time when process $i$ could recheck the *turn* variable based on the current clock *now* and the lower bound $l\_check$.

The *check*($i$) transition is enabled when process $i$'s program counter is set to *pc_check* and its earliest checking time has passed ($first\_check[i] \leq now$). When the transition executes, two interesting cases may arise: If process $i$ finds that *turn* is still equal to $i$, it leaves the trying region and enters the critical region. On the other hand, if it finds the *turn* variable equal to anything else, it gives up the current attempt and goes back to the testing step. In either case, $first\_check[i]$ is reset to its default, 0.

The subsequent transitions are quite straightforward. A *crit*($i$) transition represents process $i$ moving into the critical region, and an *exit*($i$) transition represents process $i$ leaving the critical region. A *reset*($i$) transition represents process $i$ resetting the *turn* variable to its default value *nil*, and a *rem*($i$) transition represents process $i$ returning to its remainder region.

The final part of the automaton description is the set of **trajectories**, that is, the functions from time to states that describe how the state is permitted to evolve between discrete steps. This model specifies one trajectory definition, named *traj*. This definition describes the evolution of the state in a way that allowed the current time *now* to increase at rate 1. All of the other state variables are of types that are defined to be discrete; these, by default, are not allowed to change during trajectories. The **stop when** condition says that a trajectory must stop if the state ever reaches a point where the current time *now* is equal to a specified deadline $last\_set[i]$, for any $i$. That is, time is not "allowed to pass" beyond any deadline currently in force.

This **stop when** condition is an example of a phenomenon whereby an automaton can prevent the passage of time. This may look strange (at first) to some programmers, since programs of course cannot prevent time from passing. However, appearances can be deceiving and the Fischer automaton is not exactly a program; it is a *descriptive model* that expresses both the usual sort of behavior expressed by a program, plus additional timing assumptions that might be expressed in other ways.

## 3.2 Properties of the algorithm

Tempo can be used to describe not just algorithms, but also properties that we would like the algorithms to satisfy. For example, the Fischer algorithm is supposed to satisfy the *mutual exclusion* property, saying that no two processes can simultaneously reside in their critical regions. This is a claim that the mutual exclusion is an *invariant* of the Fischer algorithm, that is, that it is true in all reachable states of the *fischer* automaton. This claim can be expressed in Tempo with a block

**invariant of** *fischer*:
$\forall i$: *process* $\forall j$: *process*
$(i \neq j \Rightarrow (pc[i] \neq pc\_crit \vee pc[j] \neq pc\_crit))$;

This invariant definition claims that, in any reachable state of the automaton, any two processes cannot simultaneously be in the critical section. This formal statement must, of course, be verified with a tool in order to formally prove that the algorithm is correct. For instance, one could use an interactive theorem prover such as PVS, a model-checker like UPPAAL, or run simulations of the protocol and require the simulator to check the assertions after every single step of the simulations.

# 4. SIMULATION

In this section we illustrate the use of the Tempo simulator on the Fischer Mutual Exclusion example. We also describe language extensions designed to enable simulations of Tempo specifications: schedules, simulations, and simulation relations.

## 4.1 Schedules

Timed Input/Output automata are non-deterministic machines. Indeed, at any point in time, multiple transitions may be enabled and ready to fire. The simulation of a non-deterministic computation is delicate as the simulator must resolve the non-determinism and produce a total ordering over the events by deciding which enabled action to run next. A priori, a simulator may not be able to determine which total ordering among all the possible options is worth executing.

The Tempo simulator addresses this issue by putting the modeler in charge of resolving the non-determinism with a **schedule**. A Tempo schedule is an imperative code fragment that programmatically specifies the sequence of actions that the automaton should undergo. Schedules can have local state, can observe the state of the automaton and are responsible for deciding the duration of trajectories, sequence of transitions as well as the actual arguments for these transitions. Schedules are not required to completely eliminate all non-determinism, but can limit themselves to reducing it and still rely on randomization for specific decision (e.g., the duration of a trajectory).

The code fragment in Figure 1 depicts a particular schedule for the *fischer* automaton. The local variable *dur* stores the duration of a trajectory segment. The schedule body is an imperative program that iterates through all the processes. Iteration $i$ focuses on process $i$ and starts by choosing, uniformly at random from the range [1..10], the amount of time that should pass for the system before process $i$ initiates the sequence of transitions to acquire the lock and enter the critical section.

```
schedule
  states
    dur : AugmentedReal;
  do
    while (true) do
      for i in process do
          dur := choose n where 1 ≤n ∧n ≤10;
          follow traj duration dur;
          fire output try(i);
          fire internal test(i);
          fire internal set(i);
          follow traj duration 100;
          fire internal check(i);
          fire output crit(i);
          fire output exit(i);
          fire internal reset(i);
          fire output rem(i);
      od
    od
  od
```

**Figure 1: A schedule for the *fischer* automaton**

A **fire** statement triggers the named action and provides the actual values. When a **fire** event occurs, the Tempo simulator identifies all the transitions that match the event. Indeed, several transitions could apply and the simulator must determine which transitions are enabled (their preconditions are true). Once selected for execution the transition effects clause runs to update the state variable of the automaton. When all the transitions have fired, the control is returned to the schedule.

When simulating a composite automaton, the **fire** event may correspond to a handshake between an output transition of a component and one or more input transitions of other components. The simulator will execute all the matching transitions starting with the output and unify the arguments to pass actual values from the output to the inputs. For instance, in the model

```
automaton A
 signature output foo(n:Int)
 states x : Int := 10;
 transitions
   output foo(n)
   eff
     n := x;

automaton B(k : Int)
 signature input foo(n:Int)
 states y : Int := 0;
 transitions
   input foo(n)
   eff
     y := n + k;

automaton C
 components a:A; b1:B(0);b2:B(4);
 schedule
        states n : Int := 5;
 do
    fire output a.foo(n);
    print n;
 od
```

the automaton C has three components *a*, *b1* and *b2*. When the schedule of *C* executes, it declares a local variable $n$, sets it to 5 and fires the *foo* output action of component *a*. Given that the two other components (*b1* and *b2*) have matching input actions, all three are scheduled for execution starting with the output action. The output of component *a* copies the value of its state variable in the output formal $n$. The matching input executes next (in any order) and the handshake passes the value of the formal into the input action. Consequently, the input action of *b1* alters *b1*'s state variable $y$ and sets it to $10+0$. Similarly, when the input of *b2* executes, it sets its own state variable $y$ to $10+4$. Finally the control returns to the schedule, which prints the value of variable $n$, now bound to 10.

A Timed Input/Output automaton trace should feature a strict alternation of transitions and trajectories, yet, trajectories can be instantaneous (0 duration) and the Tempo simulator automatically inserts such a 0-duration trajectory between transitions as needed.

## 4.2 Simulations

Tempo supports parametric automata definitions. For instance, in the code fragment 1, the *fischer* automaton is parameterized with the lower bound on the waiting delay and

the upper bound on trajectory durations. Parametric definitions are convenient to define an entire family of automata, and once again, a simulator must bind these parameters to specific values to execute a simulation. From a simulation standpoint, it may even be desirable to execute many simulations with different parameter instantiations. Tempo addresses both needs with a *scripting* capability in the form of a **simulate** block.

For instance, to execute a single simulation of the Fischer automaton, one can write

```
simulate do
 run fischer(4,2);
od
```

and Tempo will execute the schedule associated to the *fischer* automaton with *⌊check* bound to 4 and *u_set* bound to 2.

Simulate blocks have a simple structure and can use conditionals, loops and run statements to construct scripts that perform several simulations. The scripts are not limited to a single automaton and can use multiple **run** statements to instantiate and simulate several automata. For instance, the fragment

```
simulate do
   for i in {1..4} do
       run fischer(4,i);
   od
od
```

performs a sequence of four simulations with an increasingly larger upper-bounds. Note that the last simulation will be aborted prematurely given that the actual arguments passed in do not satisfy the **where** restriction imposed by the *fischer* automaton.

## 4.3 Forward Simulation Relation

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 $B$. In order to show that $A$ implements $B$, one can use a *simulation relation* between states of $A$ and states of $B$.

Suppose 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* if

- every start state of $A$ is related (via $R$) to some start state of $B$,
- for every state $s$ of $A$ and every state $u$ of $B$ such that $R(s,u)$, and for every discrete step $(s,\pi,s')$ of $A$, there is an execution fragment $\alpha$ of $B$ starting with $u$, that has the same trace as $\pi$ and that ends with a state $u'$ such that $R(s',u')$, and
- for every state $s$ of $A$ and every state $u$ of $B$ such that $R(s,u)$, and for every trajectory $\tau$ of $A$ starting with $s$, there is an execution fragment $\alpha$ of $B$ starting with $u$ that has the same trace as $\tau$ and that ends with a state $u'$ such that $R(s',u')$.

A general theorem is that $A$ implements $B$ if there is a forward simulation from $A$ to $B$ (see Chapter 4 of [3]).

The specification of a forward simulation begins with the keywords **forward simulation**, followed by a name for the simulation relation, optional formal parameters and possibly a **where** clause constraining these parameters. It continues with descriptions of the two automata involved in the simulation. The "lower-level" automaton ($A$ in the forward simulation definition above) is specified using the keyword **from**, followed by a short name for the automaton, a colon and a description of the automaton. Similarly, the "higher-level" automaton ($B$ above) is specified using the keyword **to**, followed by a short name, a colon, and a description of the automaton.

The specification of a forward simulation continues with the keyword **mapping**, followed by a first-order predicate involving the formal parameters of the forward simulation and the state variables of the two automata. The mapping states an invariant property that must be true at every step of the forward simulation.

The **proof** section specifies the correspondence between the transitions and trajectories of the low-level and high-level automata. It is used by the simulator to drive the transition in the "high-level" automaton in response to the execution of transition and trajectories in the "low-level" automaton. Each correspondence clause can alter the parameters and remap the transition as necessary.

Consider the example in Figure 2. It features a forward simulation between two instances of *TimedChannel* with different actuals for the deadline argument that bounds the amount of time that elapses between the placement of a packet in the queue and its removal. Clearly, any trace with a deadline of 2 is a valid implementation for a TimedChannel with a looser deadline (e.g., 3) in which packets are allowed to remain in the queue for up to 3 time units. This implementation relationship can be proved (say with PVS) by showing the existence of a forward simulation relation from *TimedChannel*(2) to *TimedChannel*(3).

The Tempo simulator is used here for a *paired simulation* of two automata, a "lower-level" implementation automaton and a "higher-level" specification automaton.

The simulator starts, as usual, in the **simulate** block and proceeds with the paired simulation of *F*. To *drive* this simulation, it uses the schedule associated with the **from** automaton. This schedule simply repeats a sequence of five transitions. Each transition is chosen uniformly at random with the $x$ := **choose** *k*:*Bool* statement. If $x$ is true, the schedule fires the input transition *send* to insert a message into the channel. Otherwise, it fires the output transition *receive* to retrieve a message *m*. When a transition fires in the source automaton, the **proof** specification is used to find the matching transition in the to automaton. Once identified, the transition is fired there as well. Finally, the assertion in the mapping is verified and, provided that it is satisfied, the simulation resumes in the source automaton. The same logic is used when a trajectory is followed in the source automaton (the matching trajectory is identified and followed

```
vocabulary Message(M: Type)
 types Packet : Tuple[message: M, deadline: Real]
end

automaton TimedChannel(b: Real) where b ≥0
 imports Message(Type String)
 signature
    input send(m:String)
    output receive(m:String)
 states
    queue: Seq[Packet] : = ∅;
    now: Real : = 0;
 transitions
    input send(m)
      eff queue : = queue ⊢[m,now+b];
    output receive(m)
      pre queue ≠∅∧head(queue).message =m;
      eff queue : = tail(queue);

 trajectories
    trajdef traj
       stop when queue ≠∅∧now =head(queue).deadline;
       evolve d(now) =1;

 schedule states x: Bool;m : String : = "helo"; do
    for i in (1..5):Set[Nat] do
        x : = choose k:Bool;
        if x =true
         then fire input send("helo");
         else fire output receive(m);
      fi
        follow traj duration 20;
    od
 od

forward simulation F
        from TC1 : TimedChannel(2)
        to TC2 : TimedChannel(3)
  mapping
     TC1.now =TC2.now
     ∧len(TC1.queue) =len(TC2.queue)
     ∧∀i:Nat (1 ≤i ≤len(TC1.queue)
             ∧TC1.queue[i].deadline ≤TC2.queue[i].deadline
             ∧TC1.queue[i].message =TC2.queue[i].message);
 proof
    for input send(m) do
       fire input send(m);
    od
    for output receive(m) do
       fire output receive(m);
    od
    for trajectory traj duration k do
        follow traj duration k;
    od
end

simulate do
  run F;
od
```

**Figure 2: Forward Simulation Example.**

in the target automaton).

# 5. USER INTERFACE

The Tempo simulator is embedded in a graphical user interface implemented on the Eclipse Rich Client Platform. The simulator presents a debugger-like interface where end-users can set breakpoints within simulate blocks, schedules or even transitions. The users can then execute the simulation as a whole or until the control hits a breakpoint. When it does, an inspector panel presents a view showing the state variables of the automata involved as well as local variables or formals of routines or transitions currently executing. The execution can be resumed through single stepping, stepping over instructions or executing until the next breakpoint.

When the execution completes, the user-interface also provides a slider to inspect prior computation states. This can become particularly handy to review the execution trace in a more intuitive way.

Figure 3 shows the editor window with the control stopped at the first line of the effect of the *test*(*i*) transition. The bottom panel contains a textual trace of the execution. The snapshot was obtained when on the third step of the simulation. The visible part of the console shows all the state variables of the automaton which clearly indicates that the program counter is set to *pc_set* for process *P1*.

Figure 4 displays the state variables interactively when the control stops at a breakpoint. If the top slider is used, previous states of the computation can be restored and inspected. Displays like `[P1]:pc_test,*:pc_rem` provide a condensed representation of the array. The entry for `P1` is `pc_test` while all the other entries are set to `pc_rem`.

# 6. CONCLUSION

In this paper we overviewed the language Tempo based on Timed Input/Output Automata formalism, and the integrated toolkit that supports the specification, simulation, and analysis of distributed, concurrent, and timed systems expressed as Tempo specifications. The Tempo simulator is a powerful tool designed to simulate executions of Tempo specifications and to provide linked simulations of pairs of specifications, where one specification gives an abstract definition and the other is a more concrete specification that is supposed to implement the abstract definition. Together with interfaces to model-checking and theorem-proving tools, the Tempo toolset provides a comprehensive integrated development environment for complex distributed systems. Current work on future extensions for the toolset is funded by AFOSR and NSF, and includes automated distributed code generation from Tempo specifications and optimization of distributed system deployment in target network platforms.

# 7. ACKNOWLEDGMENTS

The work described in this paper was funded by a grant from AFOSR and partially supported by NSF award #0702670. Earlier work on a prototype simulator was performed by Anna Chefter, Stephen Garland, Dilsun Kaynar, Panayiotis Mavrommatis, Antonio Ramirez, and Edward Solovey. Major parts of the Tempo Toolkit were developed by Carleton Coffrin and Peter Musial.

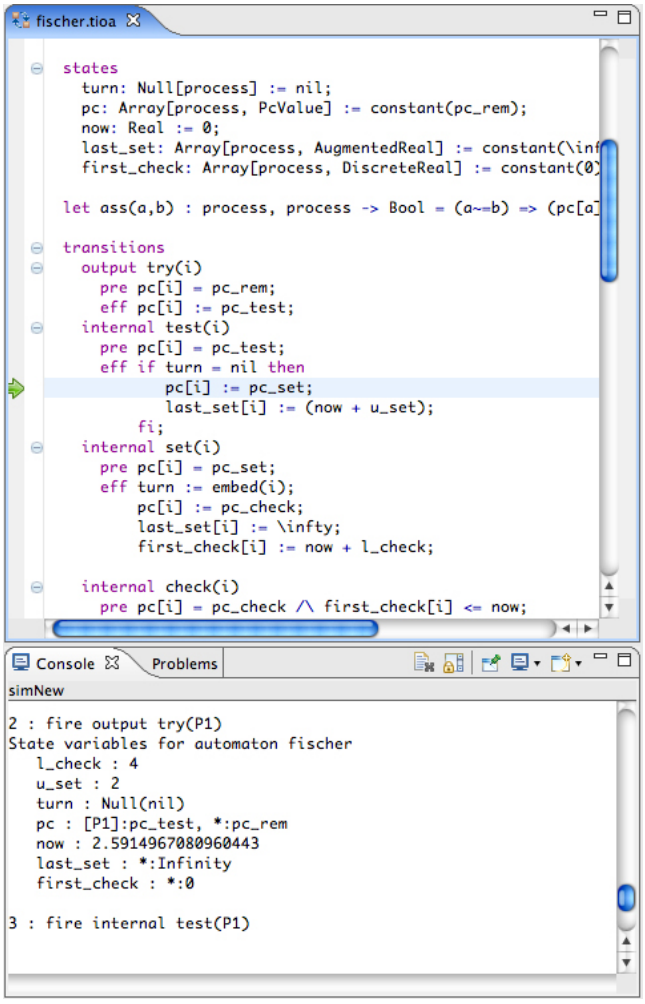


**Figure 3: The Editor window**

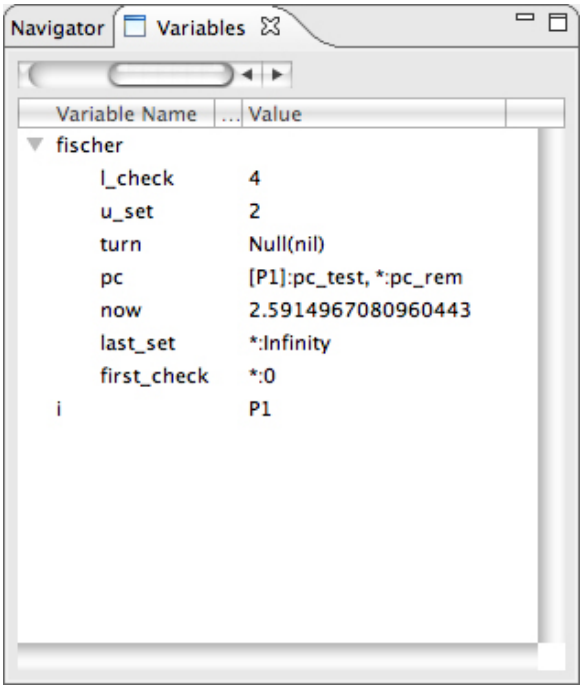


**Figure 4: The variables window**

## 8. REFERENCES


[1] Anna E. Chefter. A simulator for the IOA language. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 1998

[2] S. Garland, N. Lynch, Joshua Tauber, and M. Vaziri. *IOA User Guide and Reference Manual*. MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, 2003. Available at http://theory.lcs.mit.edu/tds/ioa.html.

[3] Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. The Theory of Timed I/O Automata. Synthesis Lectures on Computer Science, Morgan and Claypool Publishers, 123 pages, 2006. ISBN 159829010X.

[4] Kim G. Larsen, Paul Pettersson and Wang Yi. UPPAAL in a Nutshell. In Springer International Journal of Software Tools for Technology Transfer 1(1+2), pp. 134–152, 1997.

[5] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[6] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.

[7] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV '96*, LNCS 1102, pages 411–414. Springer Verlag, 1996.

[8] J. Antonio Ramirez-Robredo. Paired simulation of I/O automata. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2000.

[9] Edward Solovey. Simulation of composite I/O automata. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 2003