

# The TEMPO Simulator How-To

January 14, 2008

## 1 Getting Started

At this point we assume that the TEMPO toolkit has been installed properly along with the Simulator plugin. (The installation procedure can be found in the `INSTALL.pdf` document included in the distribution or at the distribution site of <http://www.veromodo.com/>.) We also assume that the user has some familiarity with the TEMPO model.

Formal correctness proofs for distributed systems can be long, hard or tedious to construct. Simulation can be used as a way of testing automata before development of correctness proofs begins. The execution of a TEMPO automaton either reveals bugs or increases the confidence that the automaton works as expected.

The Simulator also can assist users in constructing correctness proofs. By describing a system or an algorithm as a TEMPO program and simulating it, a user gains a better understanding of how it works. This can guide the strategy to be followed in proving correctness. Moreover, the invariants which are observed to be true for the simulated executions constitute candidates for useful lemmas in a full correctness proof.

Simulation in general is an efficient method for exposing possible deficiencies in the design of systems and algorithms which can lead to the correction of discovered errors, revision of proofs or tuning for better performance.

### 1.1 Current Limitations

The TEMPO Simulator runs selected executions of a Timed I/O Automaton on a single machine and displays information about the selected executions. The Simulator is currently under development, and some of the expected functionality has been temporarily restricted. These restrictions include the following:

- A specification must have a schedule block in order to be simulated. If there is only one schedule block, the simulator will execute this block. If there is more than one automaton with a schedule block or the automaton with a schedule is parameterized, a simulate block must be used to specify the automaton to be simulated and its parameters, if any.
- The predicate in a for-where statement must be a less-than predicate.
- The `..` set constructor is only supported for operands of type `Int`, `Nat`, and `Bool`.
- Quantified expressions are only defined for enumerable types, `Bool`, and `Nat`.
- Evolve statements must be of the form `d(x) = <literal value>;`, where the literal value must be an integer or a decimal value.
- Choose operators may only appear in state declarations and the bodies of transitions.
- Choose `det` blocks are not supported.
- The formal parameters of a transition must be identifiers, rather than arbitrary expressions.

- If a transition modifies any of its parameters, the corresponding actual parameters for that transition in a fire statement must be literal values, identifiers, tuple fields, or array elements.
- User-defined types are not supported, but user-defined shorthands of existing types are supported. User-defined vocabulary operators are supported if they only use built-in Tempo types.
- Type parameters are not supported for parametric automata.
- Components of composite automata must be given names and cannot be initialized as arrays.
- Hidden blocks in composite automata are ignored.
- Task blocks are ignored.
- Smart fire statements are ignored.
- Simulation relation proof programs may not include for loops, set-where constructors, quantifiers, or composite automata.

A complete list of restrictions can be found in [ReleaseNotes.pdf](#). Also, see Section 3.8 for an example of what happens when a specification with unimplemented TEMPO functionality is executed.

## 1.2 Running the Simulator

The TEMPO Simulator is a plug-in of the Front-End, which is responsible for parsing the source TEMPO specification. The Simulator can be invoked either via a command prompt or via the UI. For simplicity of this presentation, we assume a command prompt. In Section 4 we describe how to start the Simulator using the UI.

In order to simulate the input specification use the `plugin` option of the *Front-End*. For example, on Unix platform the complete command is:

```
tioa.sh -plugin=simNew PathAndFileName.tioa
```

On a MS Windows platform this task is accomplished via the following command:

```
tioa.bat -plugin=simNew PathAndFileName.tioa
```

For directions on how to run the Simulator on other supported platforms, we direct the reader to the [INSTALL.pdf](#) file. The full name of the executable file which launches the TEMPO tool depends on the platform; however, the use of the tool's options is platform-independent. In the remainder of this document we will assume a Unix platform.

Providing the `-plugin=simNew` option will indicate to the Front-End that the Simulator is the tool that will execute after the processing of the source specification. The file name (with extension `.tioa`) follows; if the file is not in the current working directory, then the complete path of the file must be supplied.

### 1.2.1 Simulation Steps

Simulation proceeds in *steps*, where a step represents an atomic execution of a transition or execution of a trajectory. Each simulation step is numbered, starting from one (1:), and is followed by the name of the simulated action or executed trajectory. After each step the state variables are printed. The order in which actions are *fired* (executed) is dictated by the schedule block, which must be provided for each simulated automaton.

It is possible to write schedules that are infinite or finite. A finite schedule will run until the scheduler runs out of actions to schedule and the simulation will terminate. An infinite schedule can be constructed using a `while (true) do ... od` construction. Unless the simulated automaton terminates, the simulation will continue to run forever. To prevent such behavior the Simulator will perform up to fifty steps by default and

then terminate the simulation. However, the Simulator allows users to specify their own maximum number of steps. This is done via `-steps` option. For example to allow a maximum of two steps we would perform (on Unix):

```
tioa.sh -plugin=simNew -steps=2 PathAndFileName.tioa
```

To allow a maximum of one thousand steps, we would perform (on Unix):

```
tioa.sh -plugin=simNew -steps=1000 PathAndFileName.tioa
```

Note that these are only the maximum number of steps and if the simulation self-terminates before the specified maximum number, then so does the simulation. Moreover, if the schedule is finite, but allows more than fifty steps and the user does not use the `steps` option, then the Simulator will stop after the fiftieth step.

## 2 Understanding the Simulator's Output

To illustrate the usage of the TEMPO Simulator, we will use a distribution-supplied example:

```
[loc of distribution]/common/examples/simulator/FailureDetectionExpanded.tioa.
```

This example illustrates manipulation of variables that are of tuple type, use of invariants, and a complex schedule which includes the use of schedule-defined state variables. This specification also involves an automaton with formal parameters. In order to simulate this automaton, these parameters must be bound to specific values with a `run` statement in a `simulate` block. The specification is as follows:

```
vocabulary Composition
  types M : Enumeration[nil, m1],
         TimedM : Tuple [message: M, timestamp: AugmentedReal],
         PeriodicSend2 : Tuple [failed: Bool, clock: AugmentedReal],
         TimedChannel : Tuple [queue: Seq[TimedM], now: AugmentedReal],
         Timeout : Tuple [suspected: Bool, clock: AugmentedReal]
end

% This system composes PeriodicSend2, TimedChannel and Timeout.
% A timeout action occurs after a failure of the PeriodicSend2
% automaton

automaton FDIml (u1, u2, b: Real) where (u1 + b) < u2
  imports Composition
  signature
    internal fail, send(m: M), receive(m: M)
    output timeout
  states
    P: PeriodicSend2 := [false, 0];
    C: TimedChannel := [{}, 0];
    T: Timeout := [false, 0];
  transitions
    internal send(m)
      pre ~P.failed /\ P.clock = u1;
      eff P.clock := 0;
         C.queue := C.queue |- [m, C.now + b];
    internal fail
      eff P.failed:= true;
    internal receive(m)
      pre head(C.queue).message = m;
      eff C.queue := tail(C.queue);
         T.clock:=0;
         T.suspected:= false;
    output timeout
```

```

    pre ~T.suspected /\ T.clock = u2;
    eff T.suspected := true;
trajectories
  trajdef traj
    stop when (C.queue ~= {} /\ (head(C.queue)).timestamp = C.now) \/
      (~T.suspected /\ T.clock = u2) \/
      (~P.failed /\ P.clock = u1);

    evolve
      d(P.clock) = 1;
      d(C.now) = 1;
      d(T.clock) = 1;
  schedule
    states
      count: Nat := 0;
      n: Nat :=2;
    do
      follow traj duration u1;
      % Send n rounds of messages before failing
      while (count < n) do
        fire internal send(m1);
        follow traj duration b;
        fire internal receive(m1);
        follow traj duration (u1-b);
        count := count + 1;
      od;
      % failure
      fire internal fail;
      follow traj duration u2 - (u1-b);
      % detection
      fire output timeout;
      follow traj duration \infty;
    od

invariant I of FDIml:
  C.now >= 0;
  C.now >= 0 /\ C.queue ~= {}
  => C.now <= (head(C.queue)).timestamp;
  (C.now + u2) >= 0 /\ ~T.suspected
  => T.clock ~= \infty /\ T.clock <= u2;
  (C.now + u1) >= 0 /\ ~P.failed
  => P.clock ~= \infty /\ P.clock <= u1;
  %% cannot be simulated because \A will iterate n for all Nats
  %%\A n: Nat
  %% (n < len(C.queue) =>
  %%   C.queue[n].timestamp <= (C.now + b));
  b >= 0 /\ ~ P.failed
  =>
  (if C.queue ~= {}
   then (head(C.queue)).timestamp < (T.clock + (C.now + u2))
   else (P.clock + b) < (T.clock + (C.now + u2)));
  T.suspected => P.failed;

simulate do
  run FDIml(5,8,2);
od

```

Assuming that the tool will be executed in the directory where the example is found, invoking the following options will launch the Simulator (on Unix):

```
tioa.sh -plugin=simNew FailureDetectionExpanded.tioa
```

The simulation begins with the initialization of the automaton's state variables. More specifically, the assignments of values in the automaton's state variable block are now executed. The following is a snippet of the Simulator-generated trace.

```

Instantiate automaton FDImp1
State variables for automaton FDImp1
  u1 : 5
  u2 : 8
  b : 2
  P : [0,0]
  C : [ {},0]
  T : [0,0]

```

Once the state variables are initialized, simulation begins. The first step is to follow the trajectory `traj` for `u1` units. Variable `u1` is one of the automaton's formal parameters and is bound to five by the `run` command. Hence, the trajectory will be followed for five units. As a result the `clock` fields of state variables `P`, `C`, and `T` are modified. The printout following the first step depicts the new state values.

```

1 : follow FDImp1.traj duration 5.0
State variables for automaton FDImp1
  u1 : 5
  u2 : 8
  b : 2
  P : [0,5.0]
  C : [ {},5.0]
  T : [0,5.0]

```

The next event to be simulated is the `send(m1)` action (i.e. `fire output send(m1)`). This action results in modification of the state variables of `P` and `C` only:

```

2 : fire internal send(m1)
State variables for automaton FDImp1
  u1 : 5
  u2 : 8
  b : 2
  P : [0,0]
  C : [ {[m1,7.0]},5.0]
  T : [0,5.0]

```

Note that if a state variable is modified more than once in the code of the fired transition, only the final value will be printed following completion of the step.

In total there are thirteen steps until the end of the simulation is reached. The rest of the output is as follows:

```

3 : follow FDImp1.traj duration 2.0
State variables for automaton FDImp1
  u1 : 5
  u2 : 8
  b : 2
  P : [0,2.0]
  C : [ {[m1,7.0]},7.0]
  T : [0,7.0]

```

```

4 : fire internal receive(m1)
State variables for automaton FDImp1
  u1 : 5
  u2 : 8
  b : 2
  P : [0,2.0]
  C : [ {},7.0]
  T : [0,0]

```

```

5 : follow FDImp1.traj duration 3.0
State variables for automaton FDImp1
  u1 : 5
  u2 : 8

```

```

b : 2
P : [0,5.0]
C : [{}],10.0]
T : [0,3.0]

6 : fire internal send(m1)
State variables for automaton FDImp1
u1 : 5
u2 : 8
b : 2
P : [0,0]
C : [{}],10.0]
T : [0,3.0]

7 : follow FDImp1.traj duration 2.0
State variables for automaton FDImp1
u1 : 5
u2 : 8
b : 2
P : [0,2.0]
C : [{}],12.0]
T : [0,5.0]

8 : fire internal receive(m1)
State variables for automaton FDImp1
u1 : 5
u2 : 8
b : 2
P : [0,2.0]
C : [{}],12.0]
T : [0,0]

9 : follow FDImp1.traj duration 3.0
State variables for automaton FDImp1
u1 : 5
u2 : 8
b : 2
P : [0,5.0]
C : [{}],15.0]
T : [0,3.0]

10 : fire internal fail
State variables for automaton FDImp1
u1 : 5
u2 : 8
b : 2
P : [1,5.0]
C : [{}],15.0]
T : [0,3.0]

11 : follow FDImp1.traj duration 5.0
State variables for automaton FDImp1
u1 : 5
u2 : 8
b : 2
P : [1,10.0]
C : [{}],20.0]
T : [0,8.0]

12 : fire output timeout
State variables for automaton FDImp1
u1 : 5
u2 : 8
b : 2
P : [1,10.0]

```

```
C : [{}],20.0]
T : [1,8.0]
```

End of simulation reached.

```
ERROR C:\eclipse\workspace\TIOA_Distribution\common\examples\simulator\FailureDetectionExpanded.tioa[65:7]
Runtime Error: follow statement evolved for an infinite amount of time
Finished checking specifications: FailureDetectionExpanded.tioa
Messages Reported: 1
```

### 3 Other Examples

In this section we present the remaining Simulator examples found in the distribution and the expected output of each.

#### 3.1 PeriodicSend.tioa

This example illustrates a non-finite schedule, where two events are repeated indefinitely. The Simulator will not enter an infinite loop, but instead will terminate after either fifty or a user-defined number of steps have been completed (a step is either a fire or a follow event).

```
vocabulary Messages
  types M : Enumeration[nil, m1]
end

automaton PeriodicSend(u1: Real) where u1 > 0
  imports Messages
  signature
    output send(m: M)
  states
    clock: AugmentedReal := 0;
  transitions
    output send(m)
    pre clock = u1;
    eff clock := 0;
  trajectories
    trajdef traj
      stop when clock = u1;
      evolve d(clock) = 1;
  schedule do
    while (true) do
      follow traj duration u1;
      fire output send(m1);
    od;
  od

invariant Test of PeriodicSend:
  u1 > 0;

simulate do
  run PeriodicSend(5);
od
```

Assume that the user is interested only in observing the output up to the fifth step of the simulation. The following is the output as a result of simulating `PeriodicSend.tioa` up to 5 steps only, i.e. executing the following command (on Unix):

```
tioa.sh -plugin=simNew -steps=5 PeriodicSend.tioa
```

```
Instantiate automaton PeriodicSend
State variables for automaton PeriodicSend
```

```

u1 : 5
clock : 0

1 : follow PeriodicSend.traj duration 5.0
State variables for automaton PeriodicSend
u1 : 5
clock : 5.0

2 : fire output send(m1)
State variables for automaton PeriodicSend
u1 : 5
clock : 0

3 : follow PeriodicSend.traj duration 5.0
State variables for automaton PeriodicSend
u1 : 5
clock : 5.0

4 : fire output send(m1)
State variables for automaton PeriodicSend
u1 : 5
clock : 0

5 : follow PeriodicSend.traj duration 5.0
State variables for automaton PeriodicSend
u1 : 5
clock : 5.0

End of simulation reached.
WARNING C:\eclipse\workspace\TIOA_Distribution\common\examples\simulator\PeriodicSend.tioa[27:5]
Simulation has stopped (too many steps)
Finished checking specifications: PeriodicSend.tioa
Messages Reported: 1

```

### 3.2 PeriodicSend2.tioa

This example is a variation of the `PeriodicSend.tioa` example, where the schedule is modified to have the automaton fail after a specific number of messages are sent and then follow the trajectory forever.

```

vocabulary Messages
  types M : Enumeration[nil, m1]
end

automaton PeriodicSend2(u1: Real) where u1 > 0
  imports Messages
  signature
    input fail
    output send(m: M)
  states
    failed: Bool := false;
    clock: AugmentedReal := 0;
    initially u1 >= 0;
  transitions
    output send(m)
      pre ~failed /\ clock = u1;
      eff clock := 0;
    input fail
      eff failed:= true;
  trajectories
    trajdef traj
      stop when ~failed /\ clock = u1;
      evolve d(clock) = 1;
  schedule
    states count: Nat := 0; n: Nat :=2;

```

```

do
  % Send n rounds of messages before failing
  while (count < n) do
    follow traj duration u1;
    fire output send(m1);
    count := count + 1;
  od;
  fire input fail;
  follow traj duration \infty;
od

simulate do
  run PeriodicSend2(5);
od

```

The expected output is:

```

Instantiate automaton PeriodicSend2
State variables for automaton PeriodicSend2
u1 : 5
failed : 0
clock : 0

1 : follow PeriodicSend2.traj duration 5.0
State variables for automaton PeriodicSend2
u1 : 5
failed : 0
clock : 5.0

2 : fire output send(m1)
State variables for automaton PeriodicSend2
u1 : 5
failed : 0
clock : 0

3 : follow PeriodicSend2.traj duration 5.0
State variables for automaton PeriodicSend2
u1 : 5
failed : 0
clock : 5.0

4 : fire output send(m1)
State variables for automaton PeriodicSend2
u1 : 5
failed : 0
clock : 0

5 : fire input fail
State variables for automaton PeriodicSend2
u1 : 5
failed : 1
clock : 0

End of simulation reached.
ERROR C:\eclipse\workspace\TIOA_Distribution\common\examples\simulator\PeriodicSend2.tioa[34:7]
Runtime Error: follow statement evolved for an infinite amount of time
Finished checking specifications: PeriodicSend2.tioa
Messages Reported: 1

```

### 3.3 Spec.tioa

This is another example that uses invariants. However, this time the automaton and invariant are specified in such a way as to make the invariant fail.

```

automaton FDSpec(u1, u2, b: Real) where (u1 + b) < u2
signature
  internal fail
  output timeout
states
  last_timeout : AugmentedReal := 0;
  now : Real := 0;
  suspected: Bool := false;
  failed: Bool := false ;
transitions
  internal fail
    pre ~failed;
    eff failed := true;
    last_timeout := now + u2 + b;

  output timeout
    pre failed /\ ~suspected;
    eff suspected := true;
    last_timeout := \infty;

trajectories
  trajdef traj
    stop when failed /\ ~suspected /\ now = last_timeout;
    evolve d(now) = 1;

schedule do
  fire internal fail;
  fire output timeout;
od

invariant S of FDSpec:
  now >= 0;
  suspected => failed;
  failed /\ ~suspected <=> \infty ~ = last_timeout;
  now >= 0 => now <= last_timeout;
  (now + u2 + b) >= 0 /\ \infty ~ = last_timeout
  => last_timeout <= (now + u2 + b);

simulate do
  run FDSpec(5,8,2);
od

```

The expected output is:

```

Instantiate automaton FDSpec
State variables for automaton FDSpec
  u1 : 5
  u2 : 8
  b : 2
  last_timeout : 0
  now : 0
  suspected : 0
  failed : 0

```

>>>> Invariant expression at [34:5] evaluates to false prior to fire event located at [27:6]

```

1 : fire internal fail
State variables for automaton FDSpec
  u1 : 5
  u2 : 8
  b : 2
  last_timeout : 10.0
  now : 0
  suspected : 0
  failed : 1

```

```

2 : fire output timeout
State variables for automaton FDSpec
  u1 : 5
  u2 : 8
  b : 2
  last_timeout : Infinity
  now : 0
  suspected : 1
  failed : 1

End of simulation reached.
Finished checking specifications: Spec.tioa

```

### 3.4 arrays-and-shortcuts.tioa

This example demonstrates the use of arrays in conjunction with a vocabulary shortcuts. Within the vocabulary block three types are defined: tuple, enumeration, and union. This example also illustrates how to initialize arrays and use them within a transition. Notice the very simple schedule, which contains a single 'fire' call. Hence the simulation only has a single step.

```

vocabulary Shortcuts
  types tup : Tuple [i: Int, j: Int],
         enum : Enumeration [x, y, z],
         uni : Union [ff: Int, gg: Real]
end
automaton ArraysAndShortcuts
  imports Shortcuts
  signature
    output out
  states
    a: Array[Int, tup] := constant([0,0]);
    b: Array[Int, Int, tup] := constant([0,0]);
    c: Array[Int, enum] := constant(x);
    e: Array[Int, uni] := constant(ff(9));
  transitions
    output out
    eff a[1] := [3,4];
        b[0,8] := [9, 10];
        c[5] := x;
        e[3] := ff(4);
  schedule do fire output out; od

simulate do
  run ArraysAndShortcuts;
od

```

The expected output is:

```

Instantiate automaton ArraysAndShortcuts
State variables for automaton ArraysAndShortcuts
  a : *: [0,0]
  b : *: [0,0]
  c : *: x
  e : *: ff(9)

1 : fire output out
State variables for automaton ArraysAndShortcuts
  a : [1]: [3,4], *: [0,0]
  b : [0,8]: [9,10], *: [0,0]
  c : [5]: x, *: x
  e : [3]: ff(4), *: ff(9)

End of simulation reached.
Finished checking specifications: arrays-and-shortcuts.tioa

```

### 3.5 infinite-trajectory.tioa

This specification is another example of an infinite trajectory. A follow statement is used on the trajectory for an infinite number of steps. As a consequence, no other event may occur after this point.

```
automaton InfiniteTrajectories
signature
  input fail
  output send(m: Int)
states
  u: Real := 5;
  failed: Bool := false;
  clock: AugmentedReal := 0;
transitions
  output send(m)
    pre ~failed /\ clock = u;
    eff clock := 0;
  input fail
    eff failed:= true;
trajectories
  trajdef traj
    stop when ~failed /\ clock = u;
    evolve d(clock) = 1;
schedule
  states count: Nat := 0; n: Nat :=2;
  do
    follow traj duration u;
    fire output send(1);
    fire input fail;
    follow traj duration \infty;
    follow traj duration 5; % should not be followed
    fire input fail;      % should not be fired
  od

simulate do
  run InfiniteTrajectories;
od
```

The expected output is:

```
Instantiate automaton InfiniteTrajectories
State variables for automaton InfiniteTrajectories
  u : 5
  failed : 0
  clock : 0

1 : follow InfiniteTrajectories.traj duration 5.0
State variables for automaton InfiniteTrajectories
  u : 5
  failed : 0
  clock : 5.0

2 : fire output send(1)
State variables for automaton InfiniteTrajectories
  u : 5
  failed : 0
  clock : 0

3 : fire input fail
State variables for automaton InfiniteTrajectories
  u : 5
  failed : 1
  clock : 0

End of simulation reached.
```

```

ERROR C:\eclipse\workspace\TIOA_Distribution\common\examples\simulator\infinite-trajectory.tioa[25:7]
Runtime Error: follow statement evolved for an infinite amount of time
Finished checking specifications: infinite-trajectory.tioa
Messages Reported: 1

```

### 3.6 composite-simple.tioa

This sample specification includes a composite automaton with two component automata. It illustrates how the components are defined and how to schedule component actions.

```

% When simulating this example, a name of the automaton
% to be simulated must be provided in the command line.
% In this case it makes sense to simulate automaton 'C'.

```

```

automaton A
signature output out
states x:Bool := false;
transitions
  output out
  eff
    x := true;

automaton B
signature input inp
states x:Bool := true;
transitions
  input inp
  eff
    x := false;

automaton C
components A:A; comp2:B;

schedule do
  fire output A.out;
  fire input comp2.inp;
  print comp2.x;
od

simulate do
  run C;
od

```

The expected output is:

```

Instantiate automaton C
Instantiate automaton A
State variables for automaton C
  A : null
  comp2 : null

Instantiate automaton B
State variables for automaton C
  A.x : 0
  comp2 : null

State variables for automaton C
  A.x : 0
  comp2.x : 1

1 : fire output out
State variables for automaton C
  A.x : 1
  comp2.x : 1

```

```

2 : fire input inp
State variables for automaton C
  A.x : 1
  comp2.x : 0

0
End of simulation reached.
Finished checking specifications: composite-simple.tioa

```

Please note one difference in this trace from the previous traces. Just before the line “End of simulation reached.” there is a line with a single ‘0’ on it. This line was created by the *print* statement in the specification. Print statements can be used to debug the specification when using a command prompt.

### 3.7 composite-example.tioa

This is another specification of a composite automaton with two components. It illustrates three concepts commonly found in composite specifications.

- Automata can be parameterized. The `simulate` block is used to specify the automata parameters. Automaton `C` has two component automata `a` and `b`, these are based on templates `A` and `B` respectively. Both components have parameters and are instantiated based on the values of `C`’s parameters.
- Each component of `C` has a trajectory. The follow statement that appears in the `schedule` block must include each trajectory in it. When any trajectory stops, due to `stop-when` conditions becoming true, all trajectories are stopped.
- Action matching is performed on the output actions. When an output action occurs, it forces all like-named input actions to fire as well. In this example component `b` is the one that contains action `output foo`, which has a corresponding match in component `a`.

```

% A composed automaton example that demonstrates four aspects
% 1) composed follow, all trajectores will stop if any of the
%    stop-when conditions becomes true. At this point other
%    enabled actions can be fired.
% 2) automata instantiation, parameters may be used to instantiate
%    the composite automaton and component automata as needed.
% 3) paired action matching, all input actions are fired that
%    match the fired output action.
%
% NOTE THE FOLLOWING 'formals' FILE IS NEEDED
% Values are not important.
%( (n Nat 6)
%   (m Nat 7)
%   (i Int 8) )

automaton A(f: Nat)
  signature output fooA(g: Int) input foo(m: Int)
  states x: Bool := false; t:AugmentedReal := 0;
  transitions
    output fooA(g)
      eff x := true;
      g := f;
    input foo(m)
      eff x := false;
  trajectories
  trajdef ta
    stop when t = 10;
    evolve d(t) = 5;

automaton B(e: Nat, g: Int)

```

```

signature internal fooB(h: Int) output foo(l: Int)
states x: Bool := true; p : Int := 0; t:AugmentedReal := 0;
transitions
  internal fooB(h)
    eff x := false;
    p := e;
    output foo(l)
    eff x := true;
trajectories
  trajdef tb
    stop when x = false \ / t = 5;
    evolve d(t) = 1;

automaton C(n: Nat, m: Nat, i: Int)
components a:A(m);
         b:B(n,i);
schedule
states
  p : Int := 0;
do
  follow a.ta, b.tb duration 10;
  fire output a.fooA(p);
  fire internal b.fooB(p);
  p := 8;
  fire internal b.fooB(m);
  fire output b.foo(p);
  follow a.ta, b.tb duration 20;
od

simulate do
  run C(6,7,8);
od

```

The expected output is:

```

Instantiate automaton C
Instantiate automaton A
State variables for automaton C
  n : 6
  m : 7
  i : 8
  a : null
  b : null

Instantiate automaton B
State variables for automaton C
  n : 6
  m : 7
  i : 8
  a.f : 7
  a.x : 0
  a.t : 0
  b : null

State variables for automaton C
  n : 6
  m : 7
  i : 8
  a.f : 7
  a.x : 0
  a.t : 0
  b.e : 6
  b.g : 8
  b.x : 1
  b.p : 0

```

```
b.t : 0

1 : follow a.ta, b.tb duration 10.0
State variables for automaton C
n : 6
m : 7
i : 8
a.f : 7
a.x : 0
a.t : 10.0
b.e : 6
b.g : 8
b.x : 1
b.p : 0
b.t : 2.0

2 : fire output fooA(0)
State variables for automaton C
n : 6
m : 7
i : 8
a.f : 7
a.x : 1
a.t : 10.0
b.e : 6
b.g : 8
b.x : 1
b.p : 0
b.t : 2.0

3 : fire internal fooB(7)
State variables for automaton C
n : 6
m : 7
i : 8
a.f : 7
a.x : 1
a.t : 10.0
b.e : 6
b.g : 8
b.x : 0
b.p : 6
b.t : 2.0

4 : fire internal fooB(7)
State variables for automaton C
n : 6
m : 7
i : 8
a.f : 7
a.x : 1
a.t : 10.0
b.e : 6
b.g : 8
b.x : 0
b.p : 6
b.t : 2.0

5 : fire output foo(8)
State variables for automaton C
n : 6
m : 7
i : 8
a.f : 7
a.x : 1
```

```

a.t : 10.0
b.e : 6
b.g : 8
b.x : 1
b.p : 6
b.t : 2.0

respond : fire input foo(8)
State variables for automaton C
n : 6
m : 7
i : 8
a.f : 7
a.x : 0
a.t : 10.0
b.e : 6
b.g : 8
b.x : 1
b.p : 6
b.t : 2.0

>>>> Follow statement located at [57:5] was not followed, since the stop when clause was satisfied at the
start of the trajectory
6 : follow a.ta, b.tb duration 20.0
State variables for automaton C
n : 6
m : 7
i : 8
a.f : 7
a.x : 0
a.t : 10.0
b.e : 6
b.g : 8
b.x : 1
b.p : 6
b.t : 2.0

End of simulation reached.
Finished checking specifications: composite-example.tioa

```

### 3.8 noSmartFire.tioa

This example demonstrates use of functionality currently unsupported by the Simulator. The Simulator handles the unsupported functionality gracefully, rather than failing, and outputs the reason for which the specification cannot be simulated.

A smart fire is a `fire` statement without any specification of the action to fire. The Simulator currently will not choose an arbitrary action to fire. The key difficulty is in the computation of the set of enabled actions, especially since some of the actions may be parameterized.

```

% The fire statement must be followed by a specific
% action, in this case it would be "fire output out"

```

```

automaton A
signature output out
states x:Bool := true;
transitions output out

schedule do
  fire; %smart fire not implemented
  fire output out;
od

```

The expected output is:

```
Instantiate automaton A
State variables for automaton A
  x : 1

1 : fire output out
State variables for automaton A
  x : 1

End of simulation reached.
WARNING SIM C:\eclipse\workspace\TIOA_Distribution\common\examples\simulator\noSmartFire.tioa[10:5]
smart fire statements are ignored by the simulator
Finished checking specifications: noSmartFire.tioa
Messages Reported: 1
```

## 4 Running Simulator with the UI

The steps needed to run the Simulator in the UI are depicted in Figures 1 to 10. The pictures were obtained using a Windows Vista machine; other operating systems should have very similar looking windows. Let us go through each step.

1. To begin, a project must be created. Multiple projects may coexist at the same time. Use *File, New* to open a new project wizard, as in Figure 1. Then simply click *Next*.

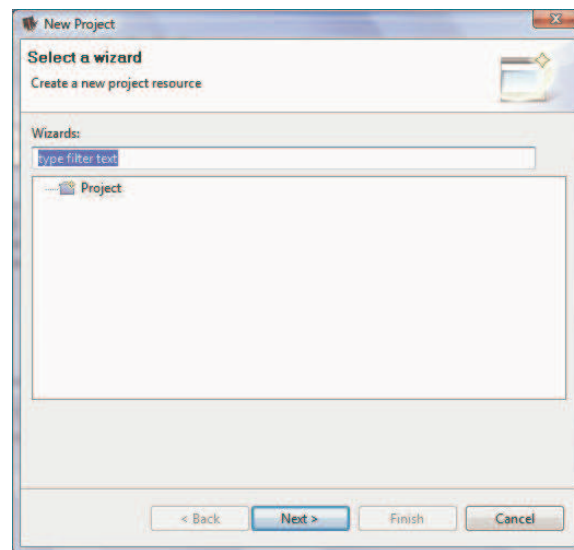


Figure 1:

2. Your project name is assigned in the next window, as in Figure 2. You may choose the default location or pick your own favorite location. Then click *Finish*. The result of your actions will look something like Figure 3.

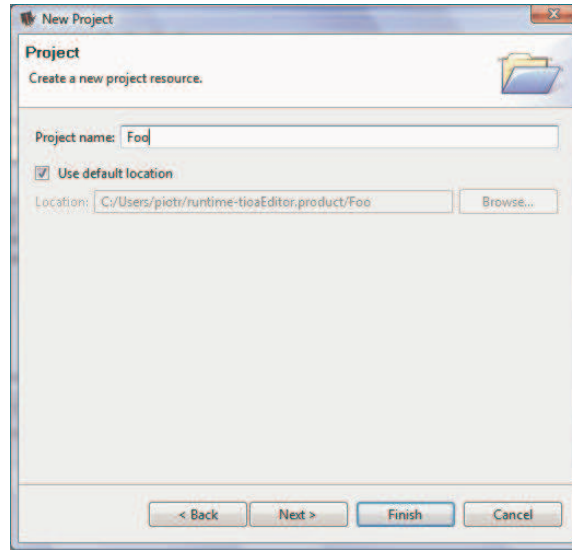


Figure 2:

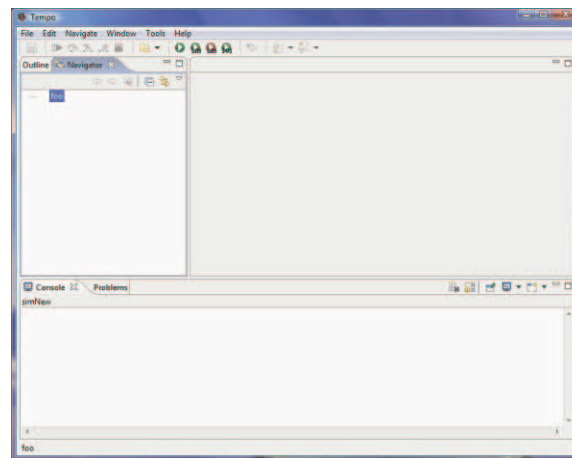


Figure 3:

3. Let's add some files into your project. First, you may want to create folders in your project (this is optional). To create a folder, choose *File, New, Folder* from the menu. You may create one or link to existing local repository of TEMPO models. The new folder dialog box is depicted in Figure 4. Click *Finish*, and the result will look something like Figure 5 (assuming that the directory is full of nice TEMPO models). Adding a new file is performed in a similar way as adding a new folder. Make sure that all TEMPO files are saved with the `.tioa` extension.

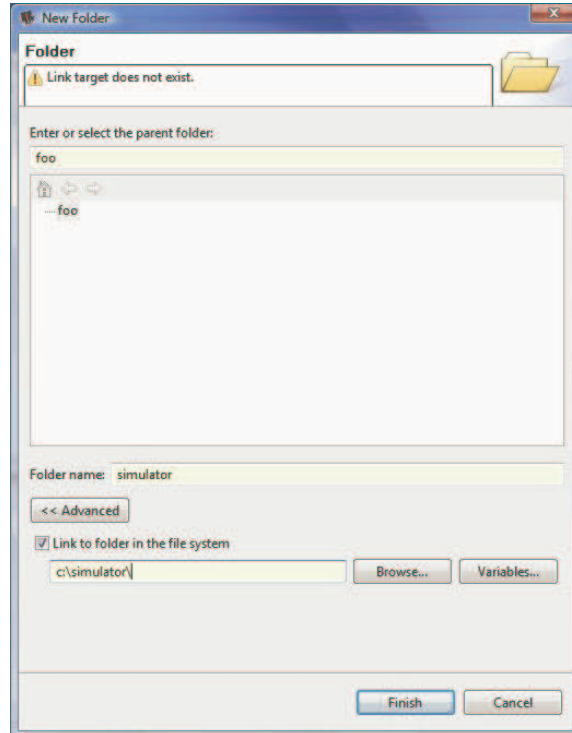


Figure 4:

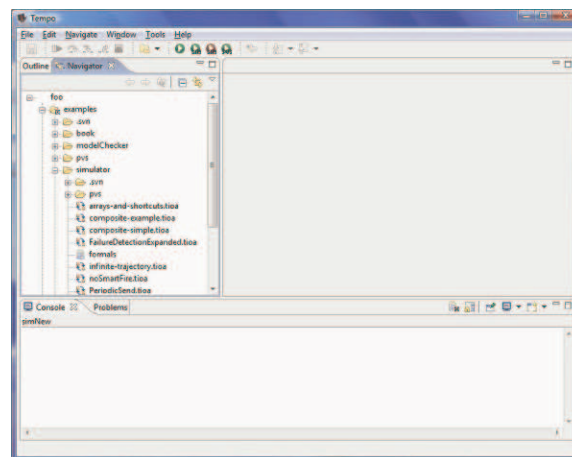


Figure 5:

4. Double click on any file, which will result in the file being displayed in the panel to the right. Please note the colored buttons, as shown in Figure 6. These buttons are associated with the different plugins. Find one that is dedicated to the Simulator. When clicked, the Simulator should run and produce a simple non-interactive trace in the *Console* tab, an example of which is shown in Figure 7.

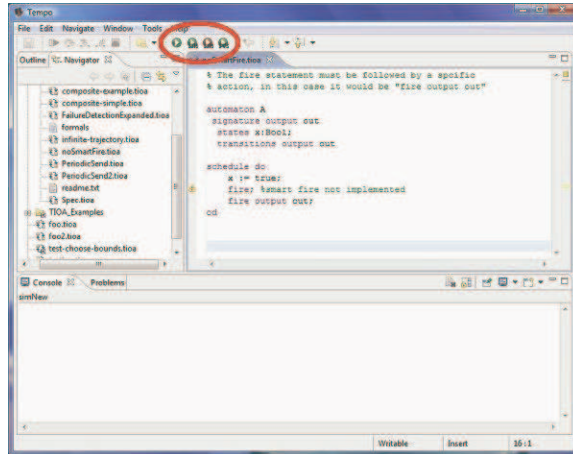


Figure 6:

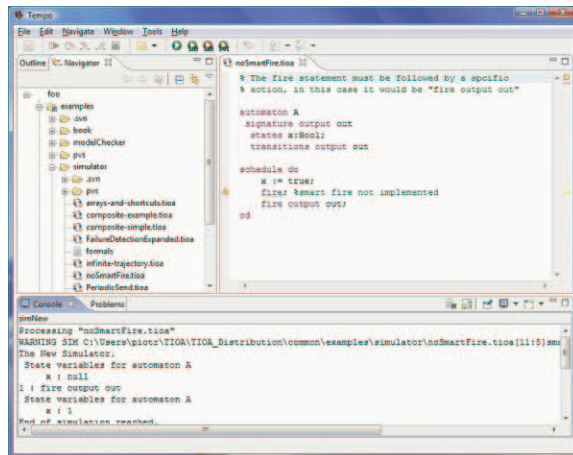


Figure 7:

5. The Simulator can be run in an interactive mode, which is enabled by setting the *Debug* option in the Simulator Options properties dialog box, as shown in Figure 8. This is accessed via *Window, Preferences*. Two other Simulator modes also are available. In the *Run* mode the Simulator will run until the simulation terminates or until a breakpoint is reached. In the *Step* mode the Simulator will pause after each assignment and comparison statement.

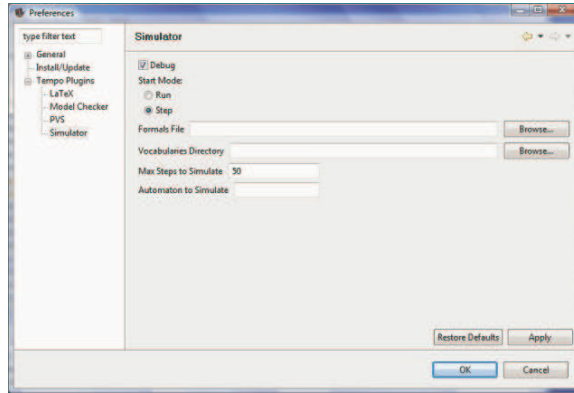


Figure 8:

6. In the *Step* mode, click on the simulate button. Additional buttons will become un-grayed. These allow single-stepping, step-into, step-out, run, and stop controls. Figure 9 depicts a simulation mid-step.

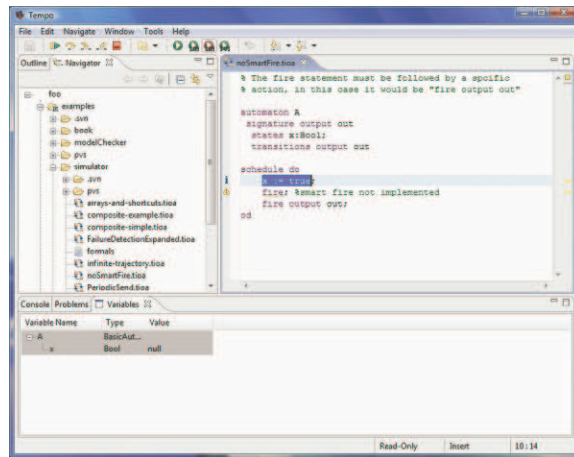


Figure 9:

7. In the *Run* mode, add some breakpoints by double-clicking the gray area next to the edit area of the file, or by right clicking on the desired line and selecting *Add breakpoint*. A simulation that has reached a user-inserted breakpoint is depicted in Figure 10.

