TIOA Simulator Manual

Panayiotis P. Mavrommatis

February 15, 2006

Chapter 1

Case Study: Failure Detection

We provide a simple example of a distributed system with timing guarantees that has been specified, simulated, checked and proved correct[?] using the TIOA tools. The example is the failure detection system from [1]. The system consists of three independent components:

- A sending process (P) that sends a message every $u1$ time units and has the potential of coming to a stopping failure,
- A channel (C) that delivers all its messages reliably within a time bound of b time units, and
- A timeout process (T) that detects the failure of the sending process by timing out. The timeout process indicates that a failure has occured in P if $u2 > u1 + b$ time has passed since it last received a message from P.

In Section 1.1 we specify the primitive automata for the components of the system, provide sample NDR schedule blocks for each component and show the output of the TIOA Simulator for these schedules. In Section 1.2 we specify the No Failure and Failure Detection systems using a composition of the individual components, and illustrate the two different options in simulation composite automata: schedules in the components or schedule in the composition. Simulator traces for both systems using both options are also shown. In Section 1.3 we show a paired simulation between two primitive automata, the hand-composed Failure Detection system's implementation and the system's specification. The NDR schedule in the implementation system and the provided step correspondence drive the paired simulation.

1.1 Simulating Primitive Automata

1.1.1 Periodic Send

In Figure 1.1.1, the `PeriodicSend` automaton uses the continuous variable `clock` as a timer to send a message every `u` time units. When a `send(m)` transition occurs, the timer is reset and another send cannot occur until `clock = u`. Its trajectory `traj` must stop when it is time to send a new message. The provided NDR schedule is a simple infinite loop that follows `traj` for `u` time units and fires the output transition `send` with a message (`m1`). Simulation of the `PeriodicSend` automaton with that schedule results in the trace shown in Figure 1.1.1. The trace repeats itself every two steps.

```
vocabulary Messages
  types M enumeration[nil, m1]

automaton PeriodicSend
  imports Messages
  signature
    output send(m: M)
  states
    u: Real := 5,
    clock: AugmentedReal := 0
    initially u ≥ 0
  transitions
    output send(m)
      pre clock = u
      eff clock := 0
  trajectories
    trajdef traj
      stop when clock = u
      evolve d(clock) = 1
  schedule do
    while (true) do
      follow traj duration u;
      fire output send(m1)
    od
  od
```

```
Automaton initialized
1:   trajectory traj for 5.0 units
2:   output transition send(m1)
3:   trajectory traj for 5.0 units
4:   output transition send(m1)
...
```

Figure 1.1: Periodic Send with no failure

1.1.2 Periodic Send with failure

In Figure 1.1.2 the `PeriodicSend2` automaton is specified, which is a modification of `PeriodicSend` that allows for a stopping failure to occur. The failure is modeled with an input transition (`fail`) which sets the `failed` flag. This disables the `send` transition and allows the `traj` trajectory to be followed for an infinite amount of time. In our sample NDR scheduler, we send two rounds of messages before failing. After failure we follow the trajectory for `\infity` time units. The trace for

this execution is also shown in Figure 1.1.2.

```
vocabulary Messages
  types M enumeration[nil, m1]

automaton PeriodicSend2 %(u: Real)
  imports Messages
  signature
     input fail
     output send(m: M)
  states
     u: Real := 5,
     failed: Bool := false,
     clock: AugmentedReal := 0
     initially u ≥ 0
  transitions
     output send(m)
       pre ¬failed ∧ clock = u
       eff clock := 0
     input fail
       eff failed:= true

trajectories
  trajdef traj
    stop when
      ¬failed ∧ u = clock
    evolve
      d(clock) = 1
schedule
  states
    count: Nat := 0,
    n: Nat := 2
  do
    % Send n rounds of messages
    while (count < n) do
      follow traj duration u;
      fire output send(m1);
      count := count + 1
    od;
    fire input fail;
    follow traj duration \infty
  od
```

```
Automaton initialized
1:   trajectory traj for 5.0 units
2:   output transition send(m1)
3:   trajectory traj for 5.0 units
4:   output transition send(m1)
5:   input transition fail
6:   trajectory traj for Infinity units
No more steps
No errors
```

Figure 1.2: Periodic Send with failure

1.1.3 Reliable Channel with deadline guarantees

In Figure 1.1.3 we model a reliable channel that ensures delivery of its messages within b time units of their receival. We first specify the `TimedM` type that augments a message with deadline. The channel enqueues messages it receives through the `send` input action in a queue, and sets their deadline to `now + b`. Its trajectory may be followed for any amount of time when the queue is empty, otherwise it should stop before or exactly at the time of the first message's deadline. Since all messages have the same maximum delay, the deadlines in the queue are monotonically non-decreasing thus the first element in the queue always has the earliest deadline.

In our sample scheduler, in every phase of the execution (every loop), we randomly decide whether or not to send a message to the channel. Then, if the queue is empty we follow the

trajectory for some amount of time that is less than b (specifically, we chose $b/2$). Otherwise, we follow the trajectory up to the point where the the first message's deadline would be met and deliver the message. Another possible schedule could deliver the message earlier instead of waiting until its deadline. For the schedule we provide, we show one sample execution in Figure 1.1.3.

```
vocabulary Messages
  types M enumeration[nil, m1]
vocabulary Timestamp
  imports Messages
  types TimedM tuple [message: M, deadline: AugmentedReal]
vocabulary Random
  operators randomBool: → Bool

automaton TimedChannel%(b: Real)
  imports Timestamp, Random
  signature
    input send(m: M)
    output receive(m: M)
  states
    b: Real := 2,
    queue: Seq[TimedM] := {},
    now: AugmentedReal := 0
    initially b ≥ 0
  transitions
    input send(m)
      eff queue := queue ⊢
                   [m, now+b]
    output receive(m)
      pre head(queue).message = m
      eff queue := tail(queue)

trajectories
  trajdef traj
    stop when queue ≠ {} ∧
       now = head(queue).deadline
    evolve d(now) = 1
schedule do
    while (true) do
      if randomBool = true then
         fire input send(m1)
      fi;
      if queue = {} then
        follow traj duration b/2
      else
        follow traj duration
          head(queue).deadline - now;
        fire output
          receive(head(queue).message)
      fi
    od
  od

Automaton initialized
1:   trajectory traj for 1.0 unit
2:   trajectory traj for 1.0 unit
3:   trajectory traj for 1.0 unit
4:   input transition send(m1)
5:   trajectory traj for 2.0 units
6:   output transition receive(m1)
7:   trajectory traj for 1.0 unit
8:   input transition send(m1)
9:   trajectory traj for 2.0 units
10:  output transition receive(m1)
...
```

Figure 1.3: Reliable Channel with deadline guarantees

1.1.4 Failure Detector

The final component of our system is the process that receives the messages and detects any failures. The `Timeout` automaton of Figure 1.1.4 maintains a flag called `suspected` that indicates whether or not the sending process is suspected to have failed. This becomes true only when $u2$ time units have passed without receiving a message. Similar to `PeriodicSend`, the `clock` variable is used as a timer that is reset every time a message is received. The automaton's trajectory may be followed for any time duration when the process is suspected, but it should stop if the timer reaches $u2$, so that the `timeout` action can occur.

The provided schedule block randomly decides whether to receive a message in every round. It then checks whether it has not received a message for the past $u2$ units, in which case it fires a `timeout` action, otherwise it allows $u2/2$ time to pass before checking again. If the sending process is already suspected of having failed, it allows an infinite amount of time to pass. Every execution of this scheduler should result in different traces because of the `randomBool` operator. We show one where a message was not received in rounds 1,2,5,6 and 7, and thus a timeout occured in the seventh round.

```
vocabulary Messages
  types M enumeration[nil, m1]
vocabulary Random
  operators randomBool: → Bool

automaton Timeout
  imports Messages, Random
  signature
    input receive(m: M)
    output timeout
  states
    u2: Real := 8,
    suspected: Bool := false,
    clock: AugmentedReal := 0
  transitions
    input receive(m)
      eff clock:=0;
          suspected:= false
    output timeout
      pre ¬suspected ∧ clock = u2
      eff suspected := true
```

```
trajectories
  trajdef traj
    stop when
      ¬suspected ∧ clock = u2
    evolve
      d(clock) = 1
schedule
  states done : Bool := false
do while (¬done) do
  if (¬suspected) then
    if randomBool then
      fire input receive(m1)
    fi;
    if clock = u2 then
      fire output timeout
    else
      follow traj duration u2/2
    fi
  else
    follow traj duration \infty;
    done := true
  fi
od
od
```

```
Automaton initialized
1:   trajectory traj for 4.0 units
2:   trajectory traj for 4.0 units
3:   input transition receive(m1)
4:   trajectory traj for 4.0 units
5:   input transition receive(m1)
6:   trajectory traj for 4.0 units
7:   trajectory traj for 4.0 units
8:   output transition timeout
9:   trajectory traj for Infinity units
No more steps
No errors
```

Figure 1.4: Failure Detector

1.2 Simulating Composite Automata

In the previous section we specified and tested all the components of the system independenty. Testing the system as a whole and the interactions of the components is not possible unless we perform a composite simulation. In Section 1.2.1 we show the first option in simulating a composite system, which is to include the schedules for the individual components and not for the composition. Alternatively, we can test the system by providing a schedule in the composition and not in the components, as we do in Section 1.2.2. For each option, we test two systems: The `No Failure` system where the `PeriodicSend` process does not fail, and the `Failure Detection` system in which the sending process fails.

1.2.1 Schedules in the Components

No Failure In Figure 1.2.1 we provide a composition of one instance of `PeriodicSend`, `TimedChannel` and `Timeout` automata. The file in which the system is specified also includes the specifications and NDR schedule blocks of `PeriodicSend`, `TimedChannel` and `Timeout` shown in Figures 1.1.1,1.1.3 and 1.1.4. The `Composition` automaton simply specifies one instance of each component and provides values for their formal parameters.

Simulation of the system results in the trace shown in Figure 1.2.1. After 5 time units, the component `P` sends a message. The input action `send` of the `C` component is also fired at the same time. After 2 time units the channel delivers the message to `T`, and 3 units later the process starts over again. The trajectories are some times broken into 1-unit steps since the `TimedChannel` process follows its trajectory every $b/2$ units when its queue is empty.

Failure Detection The composite automaton of Figure 1.2.1 is identical to that of Figure 1.2.1 except from the fact that it uses `PeriodicSend2` which can fail. The file in which the system is specified also includes the specifications and NDR schedule blocks of `PeriodicSend2`, `TimedChannel` and `Timeout` shown in Figures 1.1.2, 1.1.3 and 1.1.4. The `Composition` automaton simply specifies one instance of each component and provides values for their formal parameters.

Simulation of the system results in the trace shown in Figure 1.2.1. After 5 time units, the component `P` sends a message through the channel. After 2 time units the channel delivers the message to `T`. A message is sent once more, as the schedule of `PeriodicSend2` specifies, and at that

```
% specifications and schedules of PeriodicSend, TimedChannel and Timeout
% ...

automaton Composition
  components
    P: PeriodicSend(5: Real);
    C: TimedChannel(2: Real);
    T: Timeout(8: Real);
Automaton initialized
1:   trajectory T.traj, C.traj, P.traj for 1.0 unit
2:   trajectory T.traj, C.traj, P.traj for 1.0 unit
3:   trajectory T.traj, C.traj, P.traj for 1.0 unit
4:   trajectory T.traj, C.traj, P.traj for 1.0 unit
5:   trajectory T.traj, C.traj, P.traj for 1.0 unit
6:   output transition P.send(m1), connected to:
     input transition C.send(m1)
7:   trajectory T.traj, C.traj, P.traj for 1.0 unit
8:   trajectory T.traj, C.traj, P.traj for 1.0 unit
9:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
10:  trajectory T.traj, C.traj, P.traj for 1.0 unit
11:  trajectory T.traj, C.traj, P.traj for 1.0 unit
12:  trajectory T.traj, C.traj, P.traj for 1.0 unit
13:  output transition P.send(m1), connected to:
     input transition C.send(m1)
14:  trajectory T.traj, C.traj, P.traj for 2.0 units
15:  output transition C.receive(m1), connected to:
     input transition T.receive(m1)
...
```

Figure 1.5: No Failure System

point P fails (step 14). The message is delivered and 8 ($u2$) time units after the delivery T times out. From then on, no actions are enabled and the trajectories of the components are followed for an infinite amount of time, broken into 1-unit steps. This break happens since the TimedChannel process follows its trajectory every $b/2$ units when its queue is empty.

```
% specifications and schedules of PeriodicSend, TimedChannel and Timeout
% ...

automaton Composition
  components
    P: PeriodicSend2(5: Real);
    C: TimedChannel(2: Real);
    T: Timeout(8: Real);
Automaton initialized
1:   trajectory P.traj, T.traj, C.traj for 1.0 unit
2:   trajectory P.traj, T.traj, C.traj for 1.0 unit
3:   trajectory P.traj, T.traj, C.traj for 1.0 unit
4:   trajectory P.traj, T.traj, C.traj for 1.0 unit
5:   trajectory P.traj, T.traj, C.traj for 1.0 unit
6:   output transition P.send(m1), connected to:
     input transition C.send(m1)
7:   trajectory P.traj, T.traj, C.traj for 1.0 unit
8:   trajectory P.traj, T.traj, C.traj for 1.0 unit
9:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
10:  trajectory P.traj, T.traj, C.traj for 1.0 unit
11:  trajectory P.traj, T.traj, C.traj for 1.0 unit
12:  trajectory P.traj, T.traj, C.traj for 1.0 unit
13:  output transition P.send(m1), connected to:
     input transition C.send(m1)
14:  input transition P.fail
15:  trajectory P.traj, T.traj, C.traj for 1.0 unit
16:  trajectory P.traj, T.traj, C.traj for 1.0 unit
17:  output transition C.receive(m1), connected to:
     input transition T.receive(m1)
18:  trajectory P.traj, T.traj, C.traj for 1.0 unit
19:  trajectory P.traj, T.traj, C.traj for 1.0 unit
20:  trajectory P.traj, T.traj, C.traj for 1.0 unit
21:  trajectory P.traj, T.traj, C.traj for 1.0 unit
22:  trajectory P.traj, T.traj, C.traj for 1.0 unit
23:  trajectory P.traj, T.traj, C.traj for 1.0 unit
24:  trajectory P.traj, T.traj, C.traj for 1.0 unit
25:  trajectory P.traj, T.traj, C.traj for 1.0 unit
26:  output transition T.timeout
27:  trajectory P.traj, T.traj, C.traj for 1.0 unit
...
```

Figure 1.6: Failure Detection System

1.2.2 Schedule in the Composition

An alternative to providing shedules in the individual components is to write a schedule for the composite automaton itself. We discuss the same examples (`NoFailure` and `FailureDetection`) scheduled in this way.

No Failure The `Composition` automaton of Figure 1.2.2 is identical to that of Figure 1.2.1, but includes an NDR schedule. The file also includes the specifications of `PeriodicSend`, `TimedChannel` and `Timeout`, but without their schedule blocks.

The schedule we provide enters an infinite loop in which every $u1$ units `P` sends a message that is delivered b units later, as the trace verifies.

Failure Detection The system of Figure 1.2.2 composes `PeriodicSend2` with the channel and timeout proccesses, and includes an NDR schedule in the composition. The file must also includes the specifications (without the schedule blocks) of `PeriodicSend2`, `TimedChannel` and `Timeout`.

The schedule we provide specifies that `P` sends $n = 2$ messages before failing. After its failure and the delivery of its last message, it is detected and a `timeout` action occurs. The trace verifies this behaviour.

```
% specifications (without schedules) of PeriodicSend, TimedChannel and Timeout
% ...

automaton Composition
  components
    P: PeriodicSend(5: Real);
    C: TimedChannel(2: Real);
    T: Timeout(8: Real);
schedule
  states
      u1: Real := 5,
      u2: Real := 8,
      b: Real := 2
  do
    follow P.traj, C.traj, T.traj duration u1;
    while (true) do
      fire output P.send(m1);
      follow P.traj, C.traj, T.traj duration b;
      fire output C.receive(m1);
      follow P.traj, C.traj, T.traj duration (u1-b);
    od
  od
Automaton initialized
1:   trajectory P.traj, C.traj, T.traj for 5.0 units
2:   output transition P.send(m1), connected to:
     input transition C.send(m1)
3:   trajectory P.traj, C.traj, T.traj for 2.0 units
4:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
5:   trajectory P.traj, C.traj, T.traj for 3.0 units
6:   output transition P.send(m1), connected to:
     input transition C.send(m1)
7:   trajectory P.traj, C.traj, T.traj for 2.0 units
8:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
...
```

Figure 1.7: No Failure System with schedule in the composition

```
% specifications (without schedules) of PeriodicSend2, TimedChannel and Timeout
% ...

automaton Composition
  components
    P: PeriodicSend2(5: Real);
    C: TimedChannel(2: Real);
    T: Timeout(8: Real);
schedule
  states
    u1: Real := 5,
    u2: Real := 8,
    b: Real := 2,
    count: Nat := 0,
    n: Nat :=2
  do
    follow P.traj, C.traj,
      T.traj duration u1;
    % Send n messages before failing
    while (count < n) do
      fire output P.send(m1);
      follow P.traj, C.traj, T.traj duration b;
      fire output C.receive(m1);
      follow P.traj, C.traj, T.traj duration (u1-b);
      count := count + 1
    od;
    % failure
    fire input P.fail;
    follow P.traj, C.traj, T.traj duration u2 - (u1-b);
    % detection
    fire output T.timeout;
    follow P.traj, C.traj, T.traj duration \infty;
  od
Automaton initialized
1:   trajectory P.traj, C.traj, T.traj for 5.0 units
2:   output transition P.send(m1), connected to:
     input transition C.send(m1)
3:   trajectory P.traj, C.traj, T.traj for 2.0 units
4:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
5:   trajectory P.traj, C.traj, T.traj for 3.0 units
6:   output transition P.send(m1), connected to:
     input transition C.send(m1)
7:   trajectory P.traj, C.traj, T.traj for 2.0 units
8:   output transition C.receive(m1), connected to:
     input transition T.receive(m1)
9:   trajectory P.traj, C.traj, T.traj for 3.0 units
10:  input transition P.fail
11:  trajectory P.traj, C.traj, T.traj for 5.0 units
12:  output transition T.timeout
13:  trajectory P.traj, C.traj, T.traj for Infinity units
No more steps
No errors
```

Figure 1.8: Failure Detection System with schedule in the composition

1.3 Paired Simulation

Paired Simulations enable testing of simulations relation which indicate the relationship between the states of an implementation and a specification. If a simulation relation is proved, the implementation system is then shown to satisfy the specifications and its properties. Proving a simulation relation usually requires showing for each step of the implementation starting from an implementation state that is related to a specification state, which sequence of steps should be taken by the spec. system to result in a new state that is also related to the implementations's new state.

Both the simulation relation and its proof steps are not always easy to come up with, and are certainly very hard for a program to discover them automatically. They must therefore be provided to the Paired Simulator. In the following subsections we show an example of a system's specification, implementation and simulation relation. The system is the Failure Detection system which has already been implemented and simulated in the previous sections.

1.3.1 Failure Detection Specification

In Figure 1.3.1 we provide an abstract specification of the failure detection system. The system is specified as a single process that might `fail` and `timeout`. It keeps track of two flags, `suspected` and `failed` that carry the same meaning as in the implementation system. The `timeout_deadline` variable indicates the latest time a `timeout` transaction should occur, and `now` grows at the same rate as real time. When a failure occurs, we set `fail` to true and `timeout_deadline` to `now + u2 + b` and when a timeout occurs we set `timeout_deadline` to `Infinity` and `suspected` to true. The trajectory must stop if a failure has occurred, a timeout has not occurred and `now` has reached the `timeout_deadline`.

TODO: describe invariant

1.3.2 Failure Detection Implementation

Figure 1.3.2 provides the implementation of the Failure Detection system, in an "expanded" composition form. This means that we have transformed our composition automation into a primitive one by: (a) encapsulating the state of each component in the state of the composition (b) merging transitions by conjunctions of the preconditions and composition of the effect programs and by (c) merging the trajectory definitions by disjunctions of the stopping conditions and compositions of the evolve classes. This step was necessary because the current version of the TIOA simulator does

```
automaton FDSpec
  signature
    internal fail
    output timeout
  states
    u1: Real := 5,
    u2: Real := 8,
    b: Real := 2,
    last_timeout:
      AugmentedReal := \infty,
    now : AugmentedReal := 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

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)
```

Figure 1.9: Failure Detection System Specification

not support paired simulations among composite automata.

The implementation system is also accompanied by a schedule that will drive the execution of both systems during the paired simulation. This is identical to the schedule in ?. The invariant of the specific implementation is also provided.

1.3.3 Forward Simulation

The relation among the states and the step correspondence can now be specified and tested. The relation itself is a set of predicates relating the states of the implementation and the specification. The step correspondence is provided in a **proof** block (the name implies the fact that an actual proof would specify these step correspondences as well). Providing the implementation automation and schedule, specification automation and forward simulation with the step correspondence as those of Figure 1.3.3 in a file allows us to perform a paired simulation. A trace from the paired simulation is shown in Fig 1.3.3.

```
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]

automaton FDImpl
  imports Composition
  signature
    internal fail
    internal send(m: M)
    internal receive(m: M)
    output timeout
  states
    u1: Real := 5,
    u2: Real := 8,
    b: Real := 2,
    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
    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 FDImpl:
  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;
  ∀ 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
```

Figure 1.10: Failure Detection System Implementation

```
  forward simulation from FDImpl to FDSpec:

    % Simulation Relation
    FDImpl.P.failed = FDSpec.failed;
    FDImpl.T.suspected = FDSpec.suspected;
    FDImpl.C.now = FDSpec.now;
    (¬FDSpec.failed ⇒ FDSpec.last_timeout = \infty);
    ((FDSpec.failed ∧ FDImpl.C.queue ≠ {}) ⇒
      ∀ k: Nat (k < len(FDImpl.C.queue) ⇒
        FDSpec.last_timeout ≥ FDImpl.C.queue[k].timestamp));
    ((FDSpec.failed ∧ FDImpl.C.queue = {}) ⇒
      FDSpec.last_timeout ≥ FDImpl.T.clock)

    % Step Correspondence
    proof
      for internal send(m: M) ignore
      for internal receive(m: M) ignore
      for internal fail do fire internal fail od
      for output timeout do fire output timeout od
      for trajectory traj duration x do follow traj duration x od
Automaton initialized
1:   trajectory FDImpl.traj for 5.0 units
     trajectory FDSpec.traj for 5.0 units
2:   internal transition FDImpl.send(m1)
3:   trajectory FDImpl.traj for 2.0 units
     trajectory FDSpec.traj for 2.0 units
4:   internal transition FDImpl.receive(m1)
5:   trajectory FDImpl.traj for 3.0 units
     trajectory FDSpec.traj for 3.0 units
6:   internal transition FDImpl.send(m1)
7:   trajectory FDImpl.traj for 2.0 units
     trajectory FDSpec.traj for 2.0 units
8:   internal transition FDImpl.receive(m1)
9:   trajectory FDImpl.traj for 3.0 units
     trajectory FDSpec.traj for 3.0 units
10:  internal transition FDImpl.fail
     internal transition FDSpec.fail
11:  trajectory FDImpl.traj for 5.0 units
     trajectory FDSpec.traj for 5.0 units
12:  output transition FDImpl.timeout
     output transition FDSpec.timeout
13:  trajectory FDImpl.traj for Infinity units
     trajectory FDSpec.traj for Infinity units
No more steps
No errors
```

Figure 1.11: Failure Detection System Forward Simulation

Bibliography

[1] Dilsun Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The theory of timed I/O automata. To appear in Synthesis Lectures on Computer Science, Morgan Claypool Publishers, 2005.