Around the Tempo Toolset Userguide

Xavier Koegler, under the supervision of Nancy Lynch

March 20th - August 31st 2007

Abstract

From March 15th to August 31st 200, I attended a research internship within the Comuter Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology[1] under the supervision of Nancy Lynch.

Within her group, the Theory of Distributed Systems Group, I worked on the Tempo Toolset develloped by Veromod Inc.[2] and on six examples for the Tempo Toolset User Guide and Reference Manual.

Contents

[1]Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139
[2]Veromodo Inc. http://www.veromodo.com

1 Introduction

1.1 Introducing Tempo

Tempo is a simple formal language for modeling distributed systems as collections of interacting asynchronous state machines. It is based on the Timed Input/Output Automata Framework [5] which was developpped as an extenstion of the I/O Automata framework [9, 10, 7]. It gives a formal mathematical framework for the study of distributed algorithms and provides a rich set of capabilities for system modeling and analysis : invariant assertions, compositional reasoning and correspondences between levels of abstaction with support of timing features.

TIOA can be used to model practically any type of distributed system, from communication systems to automated process controls and even, to some extent, biological systems. TIOA allows for both discrete state changes, called **actions** or **transitions** , and continuous evolutions, called **trajectories**.

The Tempo language was designed to offer computer support to researchers to describe and analyze distributed systems. It provides a formal language to describe Timed I:O automata, based on pseudocode notations used in many research paper and allows the specification of properties such as invariants, relationships between automata at different level of abstaction.

The Tempo toolkit contains tools to support analysis of systems described using Tempo. These include lightweight tools, which check syntax and perform static semantic analysis; medium-weight tools, which simulate the action of an automaton and support model-checking using the Uppaal model-checker [6]; and heavyweight tools, which provide support for proving properties of automata using the PVS interactive theorem-prover [12].

1.2 Internship Organization

The purpose of my internship with Nancy A. Lynch was to help her in writing part of the Tempo Toolset User Guide and Reference Manual. During the first few weeks, this meant getting familiar with the Tempo Toolset and help write and correct the code for six toy examples she wanted to use in the User Guide to illustrate both the code syntax and the modeling possibilities of the Framework. Changes were made all along to the language to make it more intuitive and to keep it as close to the TIOA framework as possible and I helped keep the said examples' code up to date.

After that, I kept working around those examples, writing proofs of different properties of the algorithms, especially handproofs destined to be a reference and basis for the automated proofs run through the PVS theorem prover. This required me to discover the PVS theorem prover, although I did not actually run any of the proofs on it.

1.3 Acknowledgements

I would like to thank C.S.A.I.L, M.I.T and Veromodo Inc. for giving me the opportunity of this internship and the great environment provided.

I am most deeply thankful to Professor Nancy A. Lynch for all the time and effort she put into making this internship a very interesting experience.

I would like to thank every one in the T.D.S group and at Veromodo Inc. for welcoming me and helping me around. I would especially like to thank Shinya Umeno and Sayan Mitra – or should it be Dr. Sayan Mitra ? – for both their help and friendly chatter throughout my stay.

2 Fischer Timed Mutual Exclusion Algorithm

The Fischer Timed Mutual Exclusion Algorithm is a simple and standard test example for formal methods for modeling and analyzing distributed and timed systems. In this algorithm, a collection of processes are competing to enter a critical region by means of a shared variable called *turn*. This variable can take on a value that is either *nil* or the name of a process whose turn it is to enter the critical region.

The idea is that a process i that wants to enter the critical region should test if the *turn* variable is already owned by someone else. If such is the case then it waits and keeps retesting. If the variable is not currently owned by anyone, it proceeds to write its name in the variable to "reserve" the critical region. Testing and writing the variable are two different and separate actions meaning that between the times where i tested and wrote, another process j may have written the variable to reserve the critical region, leaving us with two processes believing that it is their turn to enter the critical region.

Thus a key step to avoid collisions in the critical region is to have a process that set the *turn* variable to its own name wait and check for any other process that might have written in the variable before entering the critical zone. Even though it may not be trivial, simply imposing a lower bound *first_check* on the time beetween setting the *turn* variable to a process name and checking to see if it has not been changed since before entering the critical zone and an upper bound *last_set* between testing for availability and writing the process name such that *last_set* is strictly less than *first_check* is enough to avoid collisions.

2.1 Fischer Automaton

There are two ways in TIOA and Tempo to model shared variables : you can either make the shared variables an automaton of their own, where the read-/write actions of the process automata are input/output actions of the shared variable automaton or you can model the entire system as a single automaton which has state variables for every single process and for the shared variables. The latter solution was adopted here. In the code found in figures 2.1 and 2.2, the actions are parametrized by the name of the process that takes them.

Most state variables are Arrays of variables indexed by process names modeling the corresponding variables of the different processes. This also creates an interesting configuration with the trajectories' stopping condtion which tests on the existence of a process for which the upper bound has been reached.

```
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(infty);
    first_check: Array[process, discrete Real] : = 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 Sample 2.1: Tempo description of the Fischer Timed Mutual Exclusion algorithm

```
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 Sample 2.2: Tempo description of the Fischer Timed Mutual Exclusion algorithm, continued

3 Two Task Race System

3.1 The Two Task Race Automaton

The Two Task Race System is a simple automaton used primarily to illustrate the concept of simulation relations and how they can be used to prove properties of certain algorithms. It this system we have two tasks *set*, which simply raises a boolean flag (once), and *main* which increments a counter while the flag is not set and then decrements it until it reaches 0 and reports when that is done. Both tasks come with both lower and upper bounds on their execution times and we wonder when the final *report* might happen.

The code for this Automaton can be found in Figure 3.3. Although it is not apparent in the code, the *increment*, *decrement* and *report* actions are all part of the main task, and thus refer to the same time bound as a precondition, while *set* constitutes the other task.

The parameters are simply the uper and lower time bounds on the different actions :

- *a1* and *a2* are respectively the lower and upper bound on the time passing between two consecutive actions of the *main* task.
- *b1* and *b2* are respectively the lower and upper bound on the time at which the *set* task will raise the flag.

Proving an upper and a lower bound on the algorithm, that an interval of time in which the *report* action will happen, is complicated if using an invariant based proof. This is why we will use another proof technique here : we will produce another automaton, called a *Specification* for which it will be easier to give upper and lower bounds as well as a *mapping* which will map any valid execution of the Two Task Race automaton to an execution of its Specification which has the same trace (that is, the same external actions occuring at the same time) thus proving that any execution of the Two Task Race automaton has the same timing properties as executions of the Specifications automaton. Such a mapping then defines a *Simulation Relation* between the two automata.

3.2 Specification and Simulation Relation

The specification automaton we use here is very simple and straightforward. The code for it can be found in Figure 3.4. The automaton has only one action, *report* which occurs once between times *c1* and *c2*. A clever choice of the parameters *c1* and *c2* is obviously needed for the existence of an apropriate mapping to be possible. Intuitively, the longest time for *report* to occur in *TTR* is when the flag is raised as late as possible, as many increases as possible have been made before that and the decreasing and reporting are done as slowly as possible. This suggests that $c2 = b2 + a2 * (b2/a1) + a2$. The same type of reasoning suggests that $c1 = b1 + a1 * (b1 - a2)/a2$.

The tempo code for the forward simulation can be found in Figure 3.5. It requires six formal parameters, one for every parameter of each automaton involved and includes a where clause defining conditions on said parameters. Note that some of those conditions on the parameters are the same as those imposed by the automata themselves. Then the mapping itself is defined by

```
automaton TTR(a1, a2, b1, b2: Real) where
  a1 > 0 ∧a2 > 0 ∧b1 ≥0 ∧b2 ≥0 ∧a2 ≥a1 ∧b2 ≥b1

  signature
    internal increment
    internal decrement
    output report
    internal set

  states
    count: Int := 0;
    flag: Bool := false;
    reported: Bool := false;
    now: Real := 0;
    first_main: discrete Real := a1;
    last_main: AugmentedReal := a2;
    first_set: discrete Real := b1;
    last_set: AugmentedReal := b2;

  transitions

    internal increment
      pre ¬flag ∧now ≥first_main;
      eff count := count + 1;
          first_main := now + a1;
          last_main := now + a2;

    internal set
      pre ¬flag ∧now ≥first_set;
      eff flag := true;
          first_set := 0;
          last_set := ∞;

    internal decrement
      pre flag ∧count > 0 ∧now ≥first_main;
      eff count := count − 1;
          first_main := now + a1;
          last_main := now + a2;

    output report
      pre flag ∧count =0 ∧¬reported ∧now ≥first_main;
      eff reported := true;
          first_main := 0;
          last_main := ∞;

  trajectories
    trajdef traj
      stop when now =last_main ∨now =last_set;
      evolve
        d(now) =1;
```

Code Sample 3.3: Tempo description of the Two-Task-Race algorithm

```
automaton TTRSpec(c1, c2: Real) where c2 ≥0 ∧c2 ≥c1

  signature
    output report

  states
    reported: Bool := false;
    now: Real := 0;
    first_report: discrete Real := c1;
    last_report: AugmentedReal := c2;

  transitions

    output report
      pre ¬reported ∧now ≥first_report;
      eff reported := true;
          first_report := 0;
          last_report := ∞;

  trajectories

    trajdef pre_report
      invariant ¬reported;
      stop when now =last_report;
      evolve d(now) =1;

    trajdef post_report
      invariant reported;
      evolve d(now) =1;
```

Code Sample 3.4: Tempo description of the Two-Task-Race behavior specification

```
forward simulation F(a1,a2,b1,b2,c1,c2: Real)
  from TTR(a1,a2,b1,b2)
  to TTRSpec(c1,c2)
  where a1 > 0 ∧a2 > 0 ∧b1 ≥0 ∧b2 ≥0 ∧c2 ≥0 ∧a2 ≥a1
  ∧b2 ≥b1 ∧c2 ≥c1
  ∧c1 =b1 + (b1 − a2) ∗ a1/a2
  ∧c2 =b2 + (b2 ∗ a2/a1) + a2

  mapping

    TTR.reported =TTRSpec.reported
    ∧TTR.now =TTRSpec.now

    ∧((¬TTR.flag ∧TTR.first_main ≤TTR.last_set) ⇒
        TTRSpec.last_report ≥
          (Real)TTR.last_set +
          (TTR.count + 2 + ((Real)TTR.last_set − (Real)TTR.first_main) / a1) ∗ a2)

    ∧((¬TTR.reported ∧(TTR.flag ∨TTR.first_main > TTR.last_set)) ⇒
        TTRSpec.last_report ≥(Real)TTR.last_main + TTR.count ∗ a2);

    ∧((¬TTR.flag ∧TTR.last_main < TTR.first_set) ⇒
        TTRSpec.first_report ≤
          (Real)TTR.first_set +
          (TTR.count + ((Real)TTR.first_set − (Real)(TTR.last_main)) / a2) ∗ a1)

    ∧((TTR.flag ∨TTR.last_main ≥TTR.first_set) ⇒
        TTRSpec.first_report ≤
          max((Real)TTR.first_main, (Real)TTR.first_set, TTR.now) + TTR.count ∗ a1)

end
```

Code Sample 3.5: Forward simulation from Two Task Race algorithm to its specification a predicate which link the variables of the two automata, each variable being preceded by the name of the automaton it belongs to for disambiguation.

The first two conjuncts of the predicate simply force that we consider identical traces of the two automata while the remaining four conjuncts express relationships between values of the earliest-time and deadline variables of the two automata.

To prove the correctness of the forward relation, one simply has to prove that it relates initial states of the two automata, and is preserved by every discrete transition and every trajectory of the lower-level automaton.

I wrote this handproof of the forward relation so that it could serve as a reference and basis for an automated proof using the PVS theorem prover and the Tempo2PVS translator as well as the predefined TIAO environment for PVS. As such it is especially detailed and a somewhat tiresome read. The proof can be found in Appendix A

4 Timeout-Based Failure Detector

In this problem, we have two processes, the sender and the receiver (or detector) and a one-way communication channel between the two. The sender process may be subject to crashes and our objective is for the detector to find out when the sender crashed. We want to be able to prove both that the detector has both *accuracy*, meaning that it does not signal the sender as failed when it has not in fact failed, and *completeness*, meaning that if the sender fails, then the receiver will actually soon time it out.

4.1 The Sender

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

automaton PeriodicSend(u: Real, M: Type) where u ≥0
    imports Message(M)
    signature
        output send(m:M)
        input fail
    states
        failed: Bool := false;
        clock: Real := 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;
```

Code Sample 4.6: Tempo description of a sending process

The sender process is a simple automaton that can do two things : it can *fail* and *Send* messages. The *fail* action is an input action since the failure is supposed to be due to some external factors. The automaton has one parameter u which is the period at which messages are to be sent: every time it sends a message, the sender resets its internal clock to 0 and it sends a new message whenever the clock reaches u again. Note that the clock keeps increasing if the sender fails thus making the exact time since the last message was sent available in any configuration.

4.2 The Timed Channel

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

automaton TimedChannel(b: Real, M: Type) where b ≥0
    imports Message(M)

    signature
        input send(m:M)
        output receive(m:M)

    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 ∃p: Packet (p  ∈ queue ∧now  =p.deadline);
            evolve d(now)  =1;
```

Code Sample 4.7: Tempo description of a sending process

The communication channel between sender and receiver is modeled by the *TimedChannel* automaton found in Figure 4.7. It consists of a time variable called *now* and a queue which is a FIFO queue which holds type *Packet* objects which are a message and its deadline. When a message is sent, its deadline is set to *now*+*b*, and a message has to be delivered (by the *receive* action) before its deadline is reached. Messages are thus delivered by the channel with a delay less than *b*. The channel is supposed to be failure-safe.

4.3 The Detector

On the receiving end of the communication channel is the detector (see Figure 4.8) which can do two things : *receive* a message in which case it resets its internal clock to 0 and *timeout* if its internal cock is greater than its parameter *u*

```
automaton Timeout(u2:Real, M: Type) where u ≥0
    imports Message(M)

    signature
        input receive(m:M)
        output timeout

    states
        suspected: Bool : = false;
        clock: Real : = 0;

    transitions

        input receive(m)
            eff clock : = 0;
                suspected : = false;

        output timeout
            pre clock =u ∧¬suspected;
            eff suspected : = true;

    trajectories
        trajdef traj
            stop when clock =u ∧¬suspected;
            evolve d(clock) =1;
```

Code Sample 4.8: Tempo description of a receiver process for the timeout system

```
automaton TimeoutSystem(u1,u2,b: Real, M: Type) where u1 ≥0 ∧u2 ≥0 ∧b ≥
0 ∧u2 > (u1 + b)
    components
        Sender: PeriodicSend(u1,M);
        Detector: Timeout(u2,M);
        Channel: TimedChannel(b,M);
```

Code Sample 4.9: Tempo description of a complete Timeout System

4.4 The Composed System and invariants

Here is illustrated how different automata can be composed to form a composed automaton. Note that the *send*(m) output action for the *PeriodicSend* component is the same action for the *TimedChannel* component but as an input.

Now, to prove both accuracy and completeness, we can prove a set of invvrariants. Those invariants can be implemented in Tempo, especially to be translated to PVS. The Tempo code for the invariants for the Timeout System can be found in Figure 4.10. A handproof of theses invariants I wrote as a base and reference for an automated PVS proof can be found in Appendix B

invariant of *TimeoutSystem*:
$Channel.now \geq 0$;

invariant of *TimeoutSystem*:
$\forall i$: *Nat* $\forall j$: *Nat* (
$(1 \leq i \wedge i < j \wedge j \leq len(Channel.queue)) \Rightarrow$
$Channel.queue[i].deadline \leq Channel.queue[j].deadline)$;

invariant of *TimeoutSystem*:
$\forall p$: *Packet* (
$p \in Channel.queue \Rightarrow$
$(Channel.now \leq p.deadline \wedge p.deadline \leq (Channel.now + b)))$;

invariant of *TimeoutSystem*:
$\neg Sender.failed \Rightarrow (Sender.clock \leq u1)$;

invariant of *TimeoutSystem*:
$\neg Detector.suspected \Rightarrow (Detector.clock \leq u2)$;

invariant of *TimeoutSystem*:
$\neg Sender.failed \Rightarrow$
$((Channel.queue \neq \emptyset \Rightarrow (head(Channel.queue).deadline <$
$Channel.now + u2 - Detector.clock))$
$\wedge$
$(Channel.queue = \emptyset \Rightarrow Channel.now + u1 - Sender.clock + b <$
$Channel.now + u2 - Detector.clock))$;

invariant of *TimeoutSystem*:
$Detector.suspected \Rightarrow Sender.failed$;

invariant of *TimeoutSystem*:
$Sender.clock \leq Detector.clock + b$

invariant of *TimeoutSystem*
$Sender.clock > u2 + b \Rightarrow Detector.suspect$

Code Sample 4.10: Tempo description of the invariants of the Timeout System

5 Leader Election Algorithm

The Leader Election Algorithm is a distributed algorithm in which a collection of processes coordinates in order to distinguish one of them as a "leader" and avoid that two different processes ever claim being the "leader". As a complication, the different processes can fail and recover. A failure detection system is included but as a separate automaton, like a black box, that keeps track of all live processes at any given time and periodically informs other processes. Such a system could be designed thanks to a Timeout based failure detector such as found in section **??** but we have left that out for the moment and use the code found in Figure 5.11 to model the failure detecting system.

```
automaton FailureDetector(delta: Real) where delta > 0
imports Process

 signature
        input fail(j: J), recover(j: J)
        output status(L: Set[J], j: J)

 states
        live: Set[J] := {j:J where true};
        clock: Real := 0;
        next_status: Array[J,Real] := constant(0);

 transitions

        input fail(j)
                eff live := live − {j};

        input recover(j)
                eff live := live ∪{j};

        output status(L, j)
                pre L =live;
                eff next_status[j] := clock + delta;

        trajectories
                trajdef normal
                        stop when ∃j:J (next_status[j] =clock);
                        evolve d(clock) =1;
```

Code Sample 5.11: Tempo description of the failure-detection service

Whenever a process fails through the *fail(j)* action, which is an input action for both the failure detecting system and the actual process, the *FailureDetector* removes it from its list of living process and whenever a process recovers, it is added back to that list. For every process, at least every period of time *delta* the system sends an update to said process with the list of currently living processes. Note that these updates are asyncronous between the different processes, but that all share the same upper bound on the period of those updates.

5.1 The Leader Election process

The actual processes are modeled through the Tempo code found in Figure 5.12. Each process keeps a list of processes it believes to be alive. To elect itself as leader, it must be the process with minimal index in that list. But that simple condition is not enough : indeed initially and after every recovery the process has no knowledge of the other processes and only knows itself as alive. To avoid newly recovered processes wrongly declaring themselves "leader" all the time, processes have to wait at least a period of time d after a recovery to elect themselves. Choosing d high enough to make sure that processes have heard of each-other when they elect themselves. In fact having d be greater than the *delta* parameter of the failure-detection system is enough to guarantee that no two processes ever claim to be the leader at the same time.

If a process is the leader, it broadcasts a message identifying itself through the output *leader*(j) action which it will then repeat periodically as long as it is the leader. Note that all actions for a process are parametrized by the index of the process itself (which is marked by the **const** j in the action declaration, signifying that not any process index may parametrize an action but only this very one).

Whenever a process receives an update on which processes are alive, it unelects itself if it was leader and goes through the whole process again.

5.2 The Complete Leader-Election System

Figure 5.13 presents us with the code for the entire system. Note that the formal parameter d of the *Elect* processes is the same as the *delta* parameter of the Failue detector which means that a process will have to wait at least until it received an update on who is alive after a recovery before electing itself. Another interesting point is the **hidden** *Status*(L,j) actions. Indeed those actions are external actions for the components of the composed system, but are **hidden** which means they become internal actions of the global system.

```
vocabulary Processes
    types J
    operators min: Set[J] → J
end

automaton Elect(j:J, d, e, u: Real) where d > 0 ∧ e > 0 ∧ u > 0
imports Processes

 signature
    input status(L: Set[J], const j), fail(const j), recover(const j)
    internal elect(const j)
    output leader(const j)

 states
    live:Set[J] := {j};
    elected: Bool := false;
    clock: Real := 0;
    last_rectime: Real := 0;
    next_announce: AugmentedReal := ∞;

 transitions
    input status(L,j)
        eff if j ∈ live then
              live := L;
              if (j ≠ min(live)) then
                elected := false;
              fi;
            fi;

    input fail(j)
        eff live := ∅;
            elected := false;
            last_rectime := 0;
            next_announce := ∞;

    input recover(j)
        eff live := {j};
            last_rectime := clock;

    internal elect(j)
         pre j ∈ live ∧ j = min(live) ∧ clock > last_rectime + d ∧ ¬elected;
         eff elected := true;
             next_announce := clock;

    output leader(j)
        pre elected = true;
        eff next_announce := clock + u;


trajectories
    trajdef normal
      stop when j ∈ live ∧
              ((j = min(live) ∧ clock ≥ last_rectime + d + e ∧ ¬elected) ∨
               clock = next_announce);
      evolve d(clock) = 1;
```

Code Sample 5.12: Tempo description of a leader-election process

```
automaton LeaderSystem(delta,e,u: Real) where delta > 0 ∧e > 0 ∧u > 0
  components
    E[j:J]: Elect(j, delta, e, u);
    FD: FailureDetector(delta);
  hidden
    status(L,j);
```

Code Sample 5.13: Tempo description of the leader-election system

6 A Dynamic Bellman-Ford Routing Algorithm.

The next example is a dynamic version of the distributed Bellman-Ford routing algorithm. The tradionnal distributed Bellmand-Ford algorithm is a routing algorithm computing a minimum-weight path from a source to other vertices in a weighted directed graph. Here, we add failure and restart mecanisms to the processes involved and try to evaluate how long it takes the automaton to correct its behavior to process failures and restarts.

6.1 Informal presentation

The principle of the algorithm is as follows.

- Unless it has failed, the source s sends periodic messages holding a distance of 0 with a period u starting immediatly. That is, unless it fails, the source will send messages at times 0, u, $2u$,...
- If it has failed and not been restarted, the source does not do anything.
- When it is restarted the source immediately sends a message to its neighbours and resumes its periodical sending every u from that time on.
- a message sent from a node i to a node j is delivered with delay no greater than b. That is, if i sends a message at time t, j will have received the message by time $t + b$.
- A non-source node keeps track of a parent information and a distance information both of which can be empty and are so initially.
- A (non-source) node that has non-empty distance and parent information periodically sends out its distance information to its outgoing neighbours with period u.
- When a (non-source) node i receives a message containing a distance c from another node j, it compares its own distance information with $c+w_{i,j}$ (which is the total weight of a path from the source through j).
  - If the new weight is stricly less than the old weight, it updates its parent and distance information to the new path (that is the parent becomes j and the distance $c + w_{i,j}$) and immediately sends out its new weight information to its outgoing neighbours. Any distance is considered better than empty distance information.
  - If the new weight is not better than the old one, it discards it.
- A non-source node with non-empty parent (and distance) information expects to hear from its parent at least every $u+b$. If it does not, it will time its parent out and discard its distant and parent information. If a node last heard from its parent at time t, it will time its parent out between times $u + b$ and $u + b + e$ if it doesn't hear from it by then.
- A failing non-source node loses all distance and parent information and any message received by such a node is lost : it is neither processed nor stored. Thus if it restarts, its parent and distance information will be empty as initially.

This algorithm is formalized into a proper Tempo automaton using the non-predefined data types presented in Figure 6.14. *Index* is used to represent the Vertices in the automaton. In addition to self-explanatory type definitions, we need a few user-defined operators : *createedge* which converts a pair of indices to the appropriate edge ; *root* which maps a weighted-graph to one of its vertices (which shall be the source, or root, for the automaton) ; *innbrs* and *outnbs* mapping a vertice to the set its incoming or outcoming neighbours.

```
vocabulary Nodes
   types Index,
            Edge : Tuple[source: Index, target: Index],
            Graph : Set[Edge],
            WeightedGraph: Tuple[G: Graph, weight: Map[Edge, Nat]],
            Message : Tuple[weight : Nat, destination: Index]

    operators
            createedge: Index, Index →Edge,
            root: Graph →Index,
            innbrs, outnbrs: Graph, Index →Set[Index]

end
```

Code Sample 6.14: Vocabulary for dynamic Bellman-Ford

6.2 The Automaton

The automaton uses combines automata of three different types to model the root process, the other processes and the communication channels (the edges of the the graph). The composed automaton appears in Figure 6.15.

```
automaton BFSystem(W: WeightedGraph, u, b, e: Real)
  where u > 0 ∧b ≥0 ∧e > 0
  components
    BRoot: BellmanFordRoot(W, u, root(W.G));
    BNR[j: Index]: BellmanFordNonRoot(W, u, b, e, j) where j ≠root(W.G);
    TC[i,j: Index]: TimedChannel2(b,i,j,Nat) where createedge(i,j) ∈ W.G;

vocabulary TimedChannel2Types(M: Type)
   types
           Index,
           Packet : Tuple[message: M, deadline: Real]
end
```

Code Sample 6.15: The Bellman-Ford system

The *TimedChannel2* automaton , found in figure 6.16 along with its specific vocabulary, is simmilar to the *TimedChannel* from example 3 with the simple addition of the indexes of the sender and receiver (i and j).

Figure 6.17 contains the automaton for the root process. The parameter u represents the interval between successive times when the automaton sends information to all of its neighbors.

```
vocabulary TimedChannel2Types(M: Type)
  types
        Index,
        Packet : Tuple[message: M, deadline: Real]
end

automaton TimedChannel2(b: Real, i,j: Index, M: Type) where b ≥0
imports TimedChannel2Types(M)

  signature
    input send(m:M, i,j: Index)
    output receive(m:M, i,j: Index)

  states
    queue: Seq[Packet] : = ∅;
    now: Real : = 0;

  transitions
    input send(m,i,j)
      eff queue : = queue ⊢[m,now+b];
    output receive(m,i,j)
      pre queue ≠∅∧head(queue).message  =m;
      eff queue : = tail(queue);

  trajectories
    trajdef traj
      stop when ∃p: Packet (p  ∈ queue ∧now  =p.deadline);
      evolve d(now)  =1;
```

Code Sample 6.16: Communication channels for dynamic Bellman-Ford

```
automaton BellmanFordRoot(W: WeightedGraph, u: Real, i: Index) where u > 0 ∧
i =root(W.G)
imports Nodes

  signature
    input fail, restart
    output send(m: Nat, const i, j: Index) where m =0 ∧j ∈ outnbrs(W.G,i)
    input receive(m: Nat, j: Index, const i) where j ∈ innbrs(W.G,i)
    internal sendupdate

  states
    failed: Bool : = false;
    sendbuffer: Set[Message] : = ∅;
    clock: Real : = 0;
    next_send: AugmentedReal : = 0;

  transitions

    internal sendupdate
      pre clock =next_send;
      eff sendbuffer : = sendbuffer ∪{m:Message where m.weight =0 ∧
m.destination ∈ outnbrs(W.G,i)};
          next_send : = clock + u;

    output send(m, i, j)
      pre [m,j] ∈ sendbuffer;
      eff sendbuffer : = sendbuffer − {[m,j]};

    input receive(m, i, j)
      eff

    input fail
      eff failed : = true;
          sendbuffer : = ∅;
          next_send : = ∞;

    input restart
      eff failed : = false;
          next_send : = clock;

  trajectories

    trajdef traj
      stop when
        clock =next_send ∨sendbuffer ≠∅;
      evolve
        d(clock) =1;
```

Code Sample 6.17: Root process for dynamic Bellman-Ford

The *sendbuffer* state variable models a messages buffer which will be emptied by *send* actions whenever it is not empty. *clock* mesures the time passed since the execution began while *nextsend* is the deadline at which a new set of messages has to be filed into the buffer by the *sendupdate* action. Incoming messages are possible, but they will be ignored. Hence the effectless *receive* action.

Finally, the *failed* variable indicates the status of the automaton and is affected by the *fail* and *restart* actions. Note that a failed root (this will also be true for other automatons) still keeps track of passing time and that an update is send right away after a *restart* action.

Trajectory stopping condtions induce that when *sendbuffer* contains any messages those are send immediately and that time cannot pass beyond a scheduled update-sending.

The other processes are all modelled by NonRoot automata found in Figures 6.18 and 6.19. These automata work the same way as the root automaton, but have some additional actions and state variables.

The *parent* and *dist* variable store the node's current path information and *timeout_deadline* stores the time by which the node should hear from its parent.

The *receive* action actually processes incoming messages, updating *parent*, *dist* and *timeout_deadline* if appropriate. Note that if *dist* is improved, an update is immediately filed into *sendbuffer* to be sent to outgoing neighbors.

Thanks to the trajectories' stopping condition and the action's precondtion, a *timeout* occurs between *timeout_deadline* and *timeout_deadline*+*e*. That means that the parent has not sent updates it was supposed to and all information about the supposed path is no longer accurate and thus discarded. No more updates are to be sent unless new information is received.

The *fail* action not only raises the flag but also loses all stored path informations and cancels all scheduled updates and timeouts.

6.3 Proving self-stabilization for the Bellman-Ford algorithm

This formal model of the Bellman-Ford algorithm allows us to prove in a fairly clear way a self-stability property. We will now prove in Theorem 1 that if at some point we can guarantee that no failures or recovery will happen for any process then after a certain time all nodes that are still connected to the source will have a correct minimal-weight path from the source to the node.

Let us consider an execution α with $l(\alpha) = \infty$ and a finite prefix α' of α such that no *fail* or *restart* actions occur in α after α' is complete. Let us define $t_0 = l(\alpha')$. In the following, we will only consider configurations that occur in the suffixe of α after α' is done.

Let $V' = \{i \in V, i.fail = false\ in\ \alpha after\ \alpha'\ is\ done\}$ and G' be the restriction of graph G to V'. Let V'' be the connex component of G' containing s and $N = |V''|$.

Let us define for all $i \in V', d(i)$ the weight of a minimal-weight path if it exists and $d(i) = \infty$ else. $V'' = \{i \in V', d(i) \in \mathbb{N}\}$.

Our objective is to prove that in α all nodes for which $d(i)$ is finite will, within a given time after the α', have the correct

Definition 1. Unrealistic Information

```
automaton BellmanFordNonRoot(W: WeightedGraph, u:Real, b:Real, e: Real, i: Index)
  where u > 0 ∧b ≥0 ∧e > 0 ∧i ≠root(W.G)
imports Nodes

  signature
    input fail, restart
    output send(m: Nat, const i, j: Index) where j ∈ outnbrs(W.G,i)
    input receive(m: Nat, j: Index, const i) where j ∈ innbrs(W.G,i)
    internal sendupdate, timeout

  states
    failed: Bool := false;
    sendbuffer: Set[Message] := ∅;
    dist: Null[Nat] := nil;
    parent: Null[Index] := nil;
    clock: Real := 0;
    next_send: AugmentedReal := ∞;
    timeout_deadline: AugmentedReal := ∞;

  transitions

    input receive(m, j, i)
      locals
        w:Nat := W.weight[createedge(j,i)];
      eff
        if ¬failed then
          if dist =nil ∨(dist ≠nil ∧(m + w < (Nat)dist)) then
            dist := embed(m + w);
            parent := embed(j);
            timeout_deadline := clock + u + b;
            sendbuffer := sendbuffer ∪{m:Message where m.weight = (Nat)dist ∧
m.destination ∈ outnbrs(W.G,i)};
            next_send := clock + u;
          else
            if (parent =embed(j) ∧dist =embed(m+w)) then
              timeout_deadline := clock + u + b;
            fi;
          fi;
        fi;

    internal timeout
      pre timeout_deadline ≠∞∧clock > timeout_deadline;
      eff dist := nil;
          parent := nil;
          next_send := ∞;
          timeout_deadline := ∞;
```

Code Sample 6.18: Non-root process for dynamic Bellman-Ford

```
internal sendupdate
  pre clock = next_send ∧ dist ≠ nil;
  eff sendbuffer := sendbuffer ∪ {m:Message where m.weight = (Nat)dist ∧
m.destination ∈ outnbrs(W.G,i)};
      next_send := clock + u;

output send(m, i, j)
  pre [m,j] ∈ sendbuffer;
  eff sendbuffer := sendbuffer − {[m,j]};

input fail
  eff failed := true;
      sendbuffer := ∅;
      dist := nil;
      parent := nil;
      next_send := ∞;
      timeout_deadline := ∞;

input restart
  eff failed := false;

trajectories

 trajdef traj
   stop when
     clock = next_send ∨ clock = timeout_deadline + e ∨ sendbuffer ≠ ∅;
   evolve
     d(clock) = 1;
```

Code Sample 6.19: Non-root process for dynamic Bellman-Ford, continued

A node $i \in V'$ is said to have *unrealistic distance* d if and only if

$$d \neq nil \wedge (BF[i].dist = d) \wedge (d < d(i)).$$

A message containing distance information d sent from node i to node j is said to be an *unrealistic message* if and only if $j \in V' \wedge (d + w_{i,j} < d(j))$.

An *unrealistic information* is either an unrealistic message (with the weight of the edge added) or an unrealistic distance.

Lemma 1. *In the suffix of α, once α' is done, if an unrealistic message with contend d is received by node j from node i then either $i \in V \setminus V'$ or $i \in V' \wedge d < d(i)$.*

Proof. Let us suppose that i is still alive and that $d \geq d(i)$. This means that there exists a path from the source s to i with total weight w less or equal to d. Such a path naturally gives us a path from the source to j with cost $w + w_{i,j} \leq d + w_{i,j}$ and thus $d(j) \leq w_{i,j}$.

By contraposition, if d is an unrealistic message, then either i has failed (and thus $i \in V \setminus V'$) or $d < d(i)$.

□

This means that after α is complete, any unrealistic message has either been sent by a now failed node or by a node which had an unrealistic distance.

If C is a configuration of the automaton occuring in α after α' is complete, we shall use the following notations, where all state variables are considered as their value in C :

$$\begin{aligned}
m(C) = min(&\{n \in \mathbb{N}, \exists i \in V', ((BF[i].dist = n) \wedge (n < c(i)))\} \cup \\
&\{n \in \mathbb{N}, \exists (i,j) \in E, (j \in V' \wedge (n - w_{i,j}) \in TC[i,j].queue)\}) \\
I(C) = &\{i \in E', ((BF[i].dist = m(C)) \wedge (BF[i].dist < c.(i)))\} \\
P(C) = &\{(d,i,j) \in \mathbb{N} \times V \times V', \\
&((i,j) \in E) \wedge (d + w_{i,j} = m(C)) \wedge (d \in TC[i,j].queue)\}
\end{aligned}$$

m is the minimal unrealistic information in the configuration, I is the set of nodes having unrealistic distance m and P is the set of messages carrying unrealistic information m.

Lemma 2. *For any configurations C and C' reached in α after time t_0, such that C' is reached in α after C we have $m(C') \geq m(C)$.*

That is to say, the minimal unrealistic information cannot decrease after α' is complete.

Proof. Such a decrease could only happen if unrealistic information is added to the system.

Transitions do not affect unrealistic informations.

Fail and Restart actions cannot occur since we are after α'.

Sendbuffer actions do not affect the unrealistic information in the system.

A timeout sets the *dist* variable of a node to *nil* which is always a realistic information so that can only increase the minimal unrealistic information.

Sending a realistic message has no effect on the unrealistic information in the system and if an unrealistic message is sent, it is sent by a node with unrealistic information which is already accounted for in the minimal unrealistic information.

Receiving a realistic message cannot add unrealistic information to the system. If a node j receives an unrealistic message with content d from node i that is better that its current information, then it sets its distance information to $d + w_{i,j} \geq m(C)$ (where C is the configuration where the action is taken) since the message was accounted for in the computation of $m(C)$.

Thus, m can only increase in α after α'.

□

Let us define for all $k \in \mathbb{N}$, C_k as the configuration reached by the automaton in α at time $t_k = t_0 + k(2b + u + e)$ after all actions occuring at that time are executed.

Lemma 3. $\forall k \in \mathbb{N}, m(C_k) \geq m(C_0) + k$.

Proof. This is proven by induction on k.

- It is trivial for $k = 0$ that $m(C_k) \geq m(C_0) + k$.
- Let us assume that $k \geq 0$ and $m(C_k) \geq m(C_0) + k$.

  Let C be the configuration reached once the last message in $P(C_k)$ has been received (which happens between times t_k and $t_k + b$). Now thanks to lemma 2 and the induction hypothesis, we are sure that $m(C) \geq m(C_0) + k$. Furthermore, we can assure that any unrealistic message $d \in TC[i,j].queue$ in transit in C between two nodes i and j is such that $d + w_{i,j} > m(C_0) + k$. Indeed, either one such unrealistic message has been sent after C_k was reached in which case $d \geq m(C_k)$ and $w_{i,j} \geq 1$ or it was already in $TC[i,j].queue$ in C_k and since it was not in $P(C_k)$, $d + w_{i,j} > m(C_0) + k$.

  Necessarily, in any configuration C' reached after C in α, for all nodes $i \in V'$ having unrealistic distance information, $BF[i].dist \geq m(C)$. This means that, because any unrealistic message sent after C has to be sent by a node having unrealistic information (since there are no more failiures), any unrealistic message received after C with content d is such that $d \geq m(C_0) + k$.

  Now let us consider the configuration C_{k+1}. Let j be a node with unrealistic distance $BF[j].dist < d(j)$. Necessarily, it has received a message from $i = BF[j].parent$ after time $t_k + b$ (or it would have timed out by time $t_{k+1} = t_k + b + (b + u + e)$) and such a message is by definition an unrealistic message which has been sent after C_k. Let d be the content of said message. Necessarily, $d \geq m(C_k)$ and thus, $BF[j] \geq m(C_k) + 1$.

  We are now sure that in C_{k+1}, any unrealistic message $d \in TC[i,j].queue$ is such that $d + w_{i,j} \geq m(C_k) + 1$ because it was sent after C_k and any node $i \in V'$ with unrealistic distance is such that $BF[i].dist \geq m(C_k) + 1$ so, according to the induction hypothesis, $m(C_{k+1}) \geq m(C_0) + k + 1$.

Finally, by induction, we have proven that $\forall k \in \mathbb{N}, m(C_k) \geq m(C_0) + k$.

□

Let $M = max\{n \in \mathbb{N}, \exists i \in V', d(i) = n\}$ be the maximal finite minimal distance of a node from the source in the restricted graph. Thanks to lemma 3, we can be sure that when C_M is reached (and any configuration thereafter reached in α), the only nodes with unrealistic distances are those that are cut off from the source.

Definition 2. If C is a configuration of the automaton let $Head_C$ be the function defined on V as follows :

$$Head_C(i) = \begin{cases} s \; when \; i = s \\ i \; when \; BF[i].parent = nil \\ i \; when \; BF[i].dist \neq d(i) \\ i \; when \; (BF[i].parent = j) \wedge (BF[j].dist + w_{j,i} \neq BF[i]) \\ Head_C(BF[i].parent) \; else \end{cases}$$

We also define $S(C) = Head_C^{-1}(\{s\})$ and $C'_k, k \in \mathbb{N}$ as the configuration reached at time $t'_k = t_0 + M(2b + u + e) + k(3b + 2u + e)$ after all actions occuring at that time have been taken.

Lemma 4. *Let C be a configuration reached after C_M in α.*

$$\begin{aligned} \forall i \in V', (Head_C(i) = s) \Rightarrow & \\ & (\exists n \in \mathbb{N}, \exists (i_k)_{0 \leq k \leq n}, (i_0 = i) \wedge (i_n = s) \wedge \\ & (\forall k \in \{0, ..., n-1\}, (BF[i_k].parent = i_k + 1) \wedge \\ & (Head_C(i_k) = s) \wedge (BF[i_k].dist = d(i_k)))) \end{aligned}$$

Proof. Let C be such a configuration and i a non-source node in V' such that $Head_C(i) = s$. Unfolding the pile of recursive calls in the computation of $Head_C(i)$ trivially creates such a finite set $(i_k)_{0 \leq k \leq n}$ such that $BF[i_k].parent = i_k + 1$ and $i_0 = i$ and $i_n = s$ and furthermore, for all $k < n$, $Head_C(i_k) = s$ and thus $BF[i_k].dist = d(i)$. □

This means that in the *Parent Graph*, that is the subgraph of G where there is an edgre from i to j if and only if $BF[j].parent = i$, $S(C)$ is a tree rooted on s where all nodes have their best possible estimate on the distance to the source. Furthermore, $S(C)$ is included in V''.

Lemma 5. *If C is a configuration of the automaton reached in α after C_M then for all configuration C' reached in α after C, $S(C) \subseteq S(C')$.*

Proof. Thanks to Lemma 3, we know that for any configuration C reached after C_M, $m(C) \geq M + 1$. This means that no node $i \in V'$ that is connected to s in G' can ever receive any unrealistic distance information which implies that as long as a node i has distance information equal to $d(i)$, the only change that may happen is a timeout (since receiving a "better" message would be receiving unrealistic information).

Now, let C be a configuraion reached after C_M in α and C' a configuration reached after C. Let i be a node in $S(C)$. According to lemma 4, there exists $n \in \mathbb{N}$ and a chain $(i_k)_{0 \leq k \leq n}$ of nodes such that for all $k \in \{0...n\}$, $i_k \in S(C)$.

i_n being the source will keep sending out update to the other nodes every b so at least i_{n-1} will not timeout (since it will always receive the message from the source). Thus i_{n-1} will also keep sending updates to its outgoing neighbours, among which is i_{n-2}. And from neighbour to neighbour, no node in $(i_k)_{0\leq k\leq n}$ will time out.

This means that all those nodes will keep the exact same distance information in every state C' reached after C and thus i will be in $S(C')$ for any such state C'. □

Lemma 6. $\forall k \in \mathbb{N}, (S(C'_k) \neq V'') \Rightarrow (S(C'_{k+1}) \setminus S(C'_k) \neq \emptyset)$.

Proof. Let $k \in \mathbb{N}$ be such that $S(C'_k)$ is strictly smaller than V''. We already know that $S(C'_k) \subseteq S(C'_k + 1)$ thanks to Lemma 5.

Let j_0 be a node in $V'' \setminus S(C'_k)$ such that $d(j_0) = min\{d(j), j \in V'' \setminus S(C'_k)$. for any node $i \in V'$ such that there is a minimal-weight path from the source to j_0 in which i is the parent of j_0. Then $d(j_0) = d(i) - w_{i,j_0}$ and by definition of j_0, necessarily, $i \in S(C_k)$.

Let us prove that $j_0 \in S(C'_{k+1}$. There are two cases to consider :

1. if $BF[j_0].dist > d(j_0)$. Then by time $t_k + u + b$, j_0 will have received a message with global cost $d(j_0)$ from a node i (and no smaller-cost messages since those would be unrealistic). Let i_0 be the sender of the first such message sent. Then $d(i_0) = d(j_0) - w_{i_0,j_0}$, $i_0 \in S(C_k)$ and $d(i_0) + w_{i_0,j_0} < BF[j_0].dist$ when j_0 receives the message, thus, by $t_k + u + b$, $BF[j_0].dist = d(i_0) + w_{i_0,j_0} = d(j_0)$ and $BF[j_0].parent = i_0$ and thus $j_0 \in S(C)$ and consequently, $j_0 \in S(C'_{k+1})$.

2. if $BF[j_0].dist = d(j_0)$ then necessarily, $BF[j_0].parent \neq i_0$ (or else, we would have $i_0 \in S(C'_k)$). Let $i = BF[j_0].parent$. $BF[i].dist + w_{i,j_0} \geq d(j_0)$ or i would necessarily have some unrealistic distance that is less than M (which would be impossible according to lemma 3, since we are past C_M). Now $BF[i].dist + w_{i,j_0} \neq d(j_0)$ or we would have $Head_{C_k}(j_0) = Head_{C_k}(i)$, thus $i \notin S(C_k)$ and $BF[i].dist < BF[j_0].dist$ which contradicts the definition of i_0. So necessarily, i_0 cannot have its information refreshed after $t_k + b$ and it cannot receive better (thus unrealistic) information, so it is bound to time i out before time $t_k + u + 2b + e$. Then as above, by time $t_k + u + 2b + e + u + b$ it will have received a message with global cost $d(j_0)$ guaranteeing that $j_0 \in S(C'_{k+1})$.

□

Theorem 1. *In any configuration C reached in α after time $t_0 + M(2b + u + e) + N(3b + 2u + e)$ we have :*

$$\forall i \in V'', (i \neq s) \Rightarrow (BF[i].dist = d(i) \wedge Head_C(i) = s)$$

Proof. According to lemma 6, for $k \in \mathbb{N}$ as long as $S(C'_k)$ is smaller than V'', $S(C'_{k+1})$ is strictly larger. V'' being finite, there exists $k_0 \in \mathbb{N}$ such that $S(C'_{k_0}) = V''$ and $k_0 \leq N$. Thus in any configuration C reached in α after C'_N, we have $S(C) = V''$ which proves our theorem. □

7 One-Shot Vehicle Controller

The final automaton is used to illustrate the modelling of hybrid (i.e. both discrete and contiunous) automata like robots of vehicles. This example models a Train-Controler system. The system (Figure 7.20) has six parameters : the train's initial speed *v0*, a target position *xt*, a minimal and maximal speed desired when the train reaches its target position *vtmin* and *vtmax* and finaly a minimal and maximal braking acceleration for the train.

automaton *OneshotSys*(*v0*, *xt*, *vtmin*, *vtmax*, *amin*, *amax*: *Real*)
  where $v0 > 0 \wedge amin \leq amax \wedge amax < 0 \wedge 0 \leq xt \wedge$
    $0 \leq vtmin \wedge vtmin \leq vtmax \wedge vtmax \leq v0 \wedge$
    $xt \geq (vtmax * vtmax - v0 * v0) \; / \; (2 * amax)$

components
  Train: *Train*(*v0*, *amin*, *amax*);
  Controller: *Oneshot*($1/v0 * (xt - (vtmax * vtmax - v0 * v0) \; / \; (2 * amax))$,
    $(vtmax - v0)/amax$,
    $(vtmin - v0)/amin$);

Code Sample 7.20: The controlled system

The Train (Figure 7.21 is a very simple vehicle which moves in a straight line. It has two possible moving modes which are “normal” and “braking”. In normal mode, the train's acceleration is 0. When it switches into braking mode, through the *brakeOn* **input** action, it randomly chooses an acceleration value within an interval given as a parameter. This interval is an interval of negative numbers to insure that the train does actually brake.

The interesting behavior of the *Train* automaton comes from its trajectories : In previous automata, the only evolution happening in trajectories was for time or clock variables that evolved at a constant rate. Here, we have three parameters evolving : velocity *v*, position *x* and time *now*. The evolution of *now* is as usual and *v* also has a fixed, though parametrized, evolution. But for position, the evolution of *x* is linked to the velocity *v*.

The *Oneshot* controler automaton (Figure 7.22 is a more classical discrete Timed automaton which will give the *Train* *brakeOn* and *brakeOff* orders once each. There are three possible *phases* : *idle*, *braking* or *done*.

The controller is *idle* until it gives the *brakeOn* order at which point it switches to the *braking* phase. One in *braking*, the controller can give the *brakeOff* order which will switch it to *done* where it does not do anything anymore. The three parameters *A*,*B* and *C* respectively determin a deadline for the *brakeOn* action, a lower and an upper bound on the time between *brakeOn* and *brakeOff*.

Looking back at Figure 7.20, we see the instatiations of those three parameters A, B and C which are calculated specifically to ensure that the train's speed is within the desired interval when the train reaches its desired position which can be written as the following invariant :

$$x = xt \Rightarrow vtmin \leq v \wedge v \leq vtmax$$

```
automaton Train(v0, amin, amax: Real) where v0 > 0 ∧amin ≤amax ∧amax < 0

    signature
        input brakeOn, brakeOff

    states
        x: Real := 0;
        v: Real := v0;
        a: Real := 0;
        b: Bool := false;
        now: Real := 0;

    transitions
        input brakeOn
            eff b := true;
                a := choose k where amin ≤k ∧k ≤amax;

        input brakeOff
            eff b := false;
                a := 0;


    trajectories
        trajdef On
            invariant
                b ∧amin ≤a ∧a ≤amax;
            evolve d(v) =a;
                d(x) =v;
                d(now) =1;


        trajdef Off
        invariant ¬b ∧a =0;
        evolve d(v) =a;
            d(x) =v;
            d(now) =1;
```

Code Sample 7.21: The train

```
vocabulary Oneshot_types
    types Phase: Enumeration [idle, braking, done]
end

automaton Oneshot(A, B, C: Real)
imports Oneshot_types

    signature
        output brakeOn, brakeOff

    states
        phase: Phase := idle;
        now: Real := 0;
        last_on: Real := A;
        first_off: discrete Real := 0;
        last_off: AugmentedReal := ∞;

    transitions
        output brakeOn
            pre phase =idle;
            eff phase := braking;
                first_off := now + B;
                last_off := now + C;

        output brakeOff
            pre phase =braking ∧first_off ≤now;
            eff phase := done;

    trajectories
        trajdef idle
            invariant phase =idle;
            stop when now =last_on;
            evolve d(now) =1;

        trajdef braking
            invariant phase =braking;
            stop when now =last_off;
            evolve d(now) =1;

        trajdef done
            invariant phase =done;
            evolve d(now) =1;
```

Code Sample 7.22: The controller

A Proving the Forward Relation for the Two Task Race Automaton

We want to prove that the following predicate defines a forward simulation relation between TTR and TTRSpec.

$$TTR.reported = TTRSpec.reported \wedge TTR.now = TTRSpec.now$$

$$\wedge$$

$$((\sim TTR.flag \wedge TTR.first\_main \leq TTR.last\_set) \Rightarrow TTRSpec.last\_report \geq TTR.last\_set + (TTR.count + 2 + (TTR.last\_set - TTR.first\_mains)/a1) * a2) \tag{A.1}$$

$$\wedge$$

$$((\sim TTR.reported \wedge (TTR.flag \cup TTR.first\_main > TTR.last\_set)) \Rightarrow TTRSpec.last\_report \geq TTR.last\_main + TTR.count * a2 \tag{A.2}$$

$$\wedge$$

$$((\sim TTR.flag \wedge TTR.last\_main < TTR.fist\_set) \Rightarrow TTRSpec.first\_report \leq TTR.first\_set + (TTR.Count + (TTR.first\_set - TTR.last\_main)/a2) * a1) \tag{A.3}$$

$$\wedge$$

$$((TTR.flag \cup TTR.last\_main \geq TTR.first\_set \Rightarrow (TTRSpec.first\_report \leq max(TTR.first\_main, TTR.first\_set, TTR.now) + TTR.count * a1) \tag{A.4}$$

In the proof, if x is a state variable and α is an execution fragment consisting of a discrete action surrounded by twopoint trajectories, then x^+ will be used for the value of x after α is over, while x refers to the first value of x in α. If an action leaves x unchanged, x may be used instead of x^+.

Initialization

Initially, $TTR.reported = TTRSpec.reported = false$ and $TTR.now = TTRSpec.now = 0$. Furthermore, $TTR.flag = false$.

For (1) and (2), there are two cases to consider :

- Case 1: if $a1 \leq b2$. then initially, $TTR.first\_main \leq TTR.last\_set$ and thus (2) is trivially true.

  Furthermore, we then have $TTRSpec.last\_report = b2 + b2 * \frac{a2}{a1} + a2$ which is exactly the value of the right hand side of the inequality in (1) and thus (1) is true.

- Case 2: if $a1 > b2$ then initially, $TTR.first\_main > TTR.last\_set$ and thus (1) is trivially true.

  Furthermore, we have $TTR.last\_main = a2 < a2 + b2 * \frac{a2}{a1} + b2$ and (2) is true.

For (3) and (4) there are also two cases to consider :

- Case 1: if $a2 < b1$, then initially $TTR.last\_main < TTR.first\_set$ and (4) is trivially true.

  Furthermore, we then have $TTRSpec.first\_report = b1 + (b1 - a2)/a2 * a1$ which is exactly the value of the right side of the inequality in (3). And (3) is true.

- Case 2: $a2 > b1$, then (3) is trivially true and $TTR.now = 0, TTR.count = 0$. Thus the value of the right side of the inequality in (4) is $max(a1, b1)$. But $TTRSpec.first\_report = b1 + (b1 - a2)/a2 * a1 < b1$ since $b1 - a2 < 0$. Thus (4) is true.

Induction

- increment As a precondition to increment, $TTR.flag = false$ and $TTR.first\_main \leq TTR.now$.

  $TTR.reported$, $TTRSpec.reported$, $TTRSpeclast\_report$, $TTR.last\_set$, $TTRSpec.first\_report$, $TTR.now$ and $TTR.first\_set$ are left unchanged.

  For (1) and (2), there are two cases to be considered :

  – Case 1 : When $TTR.now + a1 \leq TTR.last\_set$, (2) is trivially true since $TTR.first\_main^+ \leq TTR.last\_set$ and $TTR.flag = false$ as a precondition.

    Let us prove that (1) is also true.
    By induction, we have necessarily

    $$TTRSpec.last\_report \geq$$
    $$TTR.last\_set + (TTR.count + 2 + (TTR.last\_set - TTR.first\_main)/a1) * a2$$

    $TTR.count^+ = TTR.count + 1$ and $TTR.first\_main^+ = TTR.now + a1$, so that $TTR.first\_main^+ \geq TTR.first\_main - a1$.

    $$TTR.last\_set^- TTR.last\_main^+ \leq TTR.last\_set - TTR.first\_main - a1$$

    Thus

    $$count^+ \frac{last\_set^+ - first\_main^+}{a1} \leq count + \frac{\_last\_set - first\_main}{a1}$$

    And since last_set is unchanged,

    $$TTRSpec.last\_report^+ \geq$$
    $$TTR.last\_set^+ + (TTR.count^+ + 2 + (TTR.last\_set^+ - TTR.first\_main^+)/a1) * a2.$$

    Thus (1) is true.

– Case 2 : When $TTR.last\_set < TTR.now + a1$
  Since $TTR.fist\_main^+ = TTR.now + a1 > TTR.last\_set^+$, (1) is trivially true after increment.
  Let us prove that (2) is true too.
  We know that $TTR.now \leq TTR.last\_set$ for it is an invariant of the automaton and since $TTR.first\_main \leq TTR.now$ (which implies that $TTR.frist\_main < TTR.last\_set$), by induction hypothesis (using (1)) we have

  $$TTRSpec.last\_report \geq TTR.last\_set + (TTR.count + 2 + (TTR.last\_set - TTR.first\_main)/a1) * a2.$$

  but $TTR.last\_set - TTR.first\_main > 0$, thus

  $$\begin{aligned} TTR.last\_set + a2 + (TTR.last\_set - TTR.first\_main)/a1 * a2 &> TTR.last\_set + a2 \\ &> TTR.now + a2 \\ &> TTR.last\_main^+ \end{aligned}$$

  and $count^+ * a2 = count * a2 + a2$
  Thus $TTRSpec.last\_report \geq TTR.last\_main^+ + TTR.count^+ * a2$ and (2) is true.

For (3) and (4) : as an invariant of the system, $TTR.last\_main \leq TTR.now + a2$, thus, $TTR.last\_main \leq TTR.last\_main^+$

So there are are three cases to be considered here, depending on how $first\_set$ compares to those two :

– Case 1: When $TTR.last\_main^+ \leq TTR.first\_set$.
  (4) is trivially true.
  Since $TTR.last\_main^+ = TTR.now + a2$ and $TTR.last\_main \leq now + a2$, necessarily, $TTR.last\_main < TTR.first\_set$.
  The induction hypotheses, thus guaranties from (3) that

  $$TTRSpec.first\_report \leq TTR.first\_set + (TTR.Count + (TTR.first\_set - TTR.last\_main)/a2) * a1).$$

  Now, $TTR.Count^+ = TTR.Count + 1$ and $(TTR.first\_set^+ - TTR.last\_main^+)/a2 = (TTR.first\_set^+ - now)/a2 - 1$.
  Since $TTR.now < TTR.last\_main$, we have $TTR.first\_set^+ - TTR.now > TTR.first\_set - TTR.last\_main$.
  Thus, $TTR.first\_set^+ + (TTR.Count^+ + (TTR.first\_set^+ - TTR.last\_main^+)/a2) * a1)$ is greater than $TTR.first\_set + (TTR.Count + (TTR.first\_set - TTR.last\_main)/a2) * a1)$ and (3) is true.

- When $TTR.last\_main > TTR.first\_set$
  Then, necessarily, $TTR.last\_main^{+} > TTR.first\_set$, and (3) is trivially true.
  The induction hypotheses, guaranties from (4) that

  $$(TTRSpec.first\_report \leq max(TTR.first\_main, TTR.first\_set, TTR.now) + TTR.count * a1)$$

  Since $TTR.first\_main$ and $TTR.count$ increase and the rest is left unchanged, this inequality trivially remains true and so does (4).

- When $TTR.last\_main \leq TTR.fist\_set < TTR.last\_main^{+}$.
  (3) is trivially true in the post state.

  $max(TTR.first\_main, TTR.first\_set, TTR.now) \geq TTR.first\_set$.
  Furhtermore, $TTR.last\_main^{+} = TTR.now = a2 \leq TTR.last\_main + a2$ and $TTR.first\_set - TTR.last\_main^{+} \leq 0$
  Thus, $TTR.first\_set - TTR.last\_main \leq a2$ and, consequently, $TTR.count^{+} > TTR.count + (TTR.first\_set - tTR.last\_main)/a2$
  All of this insures that

  $$max(TTR.first\_main, TTR.first\_set, TTR.now) + TTR.count * a1 \geq TTR.first\_set + (TTR.Count + (TTR.first\_set - TTR.last\_main)/a2) * a1)$$

  And from that, (4) is true.

- decrement

  As a precondition to this action, $TTR.flag = true$ and $TTR.reported = false$, thus (1) and (3) are trivially true.

  Proving (2):

  $TTR.flag = true$, $first\_set = 0$ (because of an invariant that states that $flag = true \Rightarrow first\_set = 0$ and $TTR.now \geq TTR.first\_main$. $TTR.now$, $TTR.first\_set$ and $TTRSpec.first\_report$ are left unchanged.

  $TTR.last\_main^{+}1 = TTR.now + a2 \leq TTR.last\_main + a2$ and $TTR.count^{+} = TTR.count - 1$, thus $TTR.last\_main^{+} + TTR.count^{+} * a2 \leq TTR.last\_main + TTR.count * a2$. and $TTR.Spec.last\_report^{+} \geq TTR.last\_main^{+} + TTR.count^{+} * a2$.

  (2) is true.

  Proving (4) :

  $max(TTR.now, TTR.first\_main, TTR.first\_set) = TTR.now$.

  Since $TTR.first\_main^{+} = TTR.now + a1$ and $TTR.count^{+} = TTR.count - 1$, we have

  $max(TTR.now^{+}, TTR.first\_main^{+}, TTR.first\_set^{+}) + a1 * TTR.count^{+} \geq TTR.now + a1 * TTR.count$

  And thus (4) is true in the poststate.

- set

  As a precondition $TTR.now \geq TTR.first\ _set$ and since $TTR.now < TTR.last\_main$, $TTR.last\_main > TTR.first\_set$.

  $TTR.flag^+ = true$, thus (1) and (3) are trivially true.

  Proving (4)

  $TTR.now, TTR.count, TTR.last\_main, TTR.first\_main$ and $TTRSpec.first_report$ are left unchanged. Thus, we know that $max(TTR.now, TTR.first\_set, TTR.first\_main) = max(TTR.now^+, TTR.first\_set^+, TTR.first\_main^+)$.

  And since $TTR.last\_main \geq TTR.first\_set$, the induction hypothesis, assures us that

  $$(TTRSpec.first\_report \leq max(TTR.first\_main, TTR.first\_set, TTR.now) + TTR.count * a1)$$

  Since both sides of this inequation are left unchanged, it remains true, and so does (4).

  Proving (2) :

  There are 2 cases :

  First case : $TTR.first\_main \leq TTR.last\_set$

  $TTR.last\_main$ is left unchanged and $TTR.last\_main \leq TTR.now + a2 \leq TTR.last\_set + a2$ and $TTR.last\_set - TR.first\_main \geq^0$.

  Thus

  $$\begin{aligned} TTR.last\_main^+ + TTR.count * a2 &\leq TTR.now + a2 + TTR.count * a2 \\ &\leq TTR.last\_set + TTR.count * a2 \\ \leq TTR.last\_set + (TTR.cout + 2 + (TTR.last\_set - TTR.first\_main)/a1) * a2 \\ &\leq TTRSpec.last\_report \end{aligned}$$

  and (2) is true.

  second case : $TTR.last\_set \leq TTR.first\_main$

  Then we can directly use (2) since none of the appropriate state variables have changed.

- report

  First we prove that report is enabled in Spec whenever it is enabled in TTR :

  As a precondition to report in TTR, $TTR.count = 0$ and $TTR.flag = true$. Furthermore, $TTR.now \geq TTR.first\_main$ and $TTR.first\_set =$

0. Thus, by induction hypothesis, since (4) is true, $TTRSpec.first_report \leq TTR.now = TTRSpec.now$, thus report is also enabled in $TTRSpec$.

Now we prove that all conditions are preserved :

Both $TTR.count$ and $TTR.flag$ are left unchanged by the action, thus (1) and (3) are trivially true.

$TTR.reported^{+} = true$, thus (2) is also trivially true.

$TTRSpec.first\_report^{+} = 0$ and $TTR.now \geq 0$ thus, $TTRSpec.first\_report^{+} < TTR.now + TTR.count$ which guarantees that (4) is true.

- Trajectories :

  All state variables appearing in (1), (2) and(3) are constant with the trajectories, so (1), (2) and (3) is preserved by trajectories.

  The only state variable in (4) that evolves non-trivially with trajectories is $TTR.now$ which is growing. This assures that (4) is also preserved by trajectories.

  In trajectories, $TTR.now$ and $TTRSpec.now$ have the same differential equation, thus the equality is preserved.

  Furthermore, if a trajectory is valid for TTR, necessary, during the whole trajectory, $TTR.now \leq TTR.last\_set$ and $TTR.now \leq TTR.last\_main$ and, thanks to (1) and (2), the two imply that $TTR.now < TTRSpec.last\_report$, thus the trajectory is valid for $TTRSpec$ too.

B Proof of the Invariants for the Timeout Based Failure Detector

1. $Channel.now \geq 0$
2. $\forall p : Paquet(p \in Channel.queue \Rightarrow$

   $Channel.now \leq p.deadline \wedge p.deadline + Sender.clock \leq Channel.now + b)$

   I changed this one a little so I can use it to prove a time bound on failure detection. It basically states that the difference between the deadline of a waiting message and the current time, is less than b minus the time since the last message was sent.
3. $\forall i, j : Nat, (1 \leq i \wedge i < j \wedge j \leq len(Channel.queue)) \Rightarrow$

   $Channel.queue[i].deadline \leq Channel.queue[j].deadline$
4. $\tilde{~}Detector.suspected \Rightarrow Detector.clock <= u2$
5. $\tilde{~}Sender.failed \Rightarrow Sender.clock \leq u1$
6. $\tilde{~}Sender.failed \Rightarrow$

   $(Channel.queue = \{\} \Rightarrow (head(Channel.queue).deadline < Channel.now + u2 - Detector.clock)$

   $\wedge$

   $(Channel.queue \neq \{\} \Rightarrow Channel.now + u1 - Sender.clock + b < Channel.now + u2 - Detector.clock)$
7. $Detector.suspected \Rightarrow Sender.failed.$
8. $Sender.clock \leq Detector.clock + b$
9. $Sender.clock > u2 + b \Rightarrow Detector.suspect$

Proofs : (The invariants will be enumerated as $I_1, ..., I_9$) I didn't do any initialisations since they are trivial.

Invariant I_7 proves the accuracy of the system, while I_9 proves the completeness.

I call x^+ the value of state variable x after a given action (while simply x will be used to refer to the value before the action).

$I_1 : Channel.now \geq 0$

- actions have no impact on channel.now
- trajectories : $d(Channel.now) > 0$

$I_2 : \forall p : Paquet(p \in Channel.queue \Rightarrow$
$Channel.now \leq p.deadline \wedge p.deadline + Sender.clock \leq Channel.now + b)$

- trajectories : $d(Sender.clock) = d(Channel.now)$. $Channel.now \leq p.deadline$ because of the stopping condition.

- fail, timeout : no changes
- receive(m) : $Sender.clock, Channel.now$ are not changed.

  $\forall p : Paquet, (p \in Channel.queue^+ \Rightarrow p \in Channel.queue)$

  thus $\forall p : Paquet(p \in Channel.queue \Rightarrow$

  $Channel.now \leq p.deadline \wedge p.deadline + Sender.clock \leq Channel.now + b)$
- send(m) : $Sender.clock^+ = 0 \leq Sender.clock, Channel.now$ is not changed.

  1. Case 1 : $p \neq m$ $\forall p : Paquet((p \in Channel.queue^+ \wedge p \neq m) \Rightarrow p \in Channel.queue)$
     thus $\forall p : Paquet((p \in Channel.queue^+ \wedge p \neq m) \Rightarrow$
     $Channel.now^+ \leq p.deadline \wedge p.deadline + Sender.clock^+ \leq Channel.now^+ + b)$
  2. Case 2 : $p = m$ (p is the new packet added by the send(m) action).
     if $p = m$, then
     $p.deadline = Channel.now + b = Channel.now^+ + b$ and thus
     $Channel.now^+ \leq p.deadline \wedge p.deadline + Sender.clock^+ \leq Channel.now^+ + b)$

$I_3 : \forall i, j : Nat, (1 \leq i \wedge i < j \wedge j \leq len(Channel.queue)) \Rightarrow$
$Channel.queue[i].deadline \leq Channel.queue[j].deadline$

- send(m) : does nothing on the first few examples and let $j = len(Channel.queue^+)$ then for all $1 \leq i < j$, thanks to I_2, we can assure that $Channel.queue[i].deadline \leq Channel.now + b$ and thus

  $Channel.queue[i].deadline \leq Channel.queue[j].deadline.$
- receive(m) : $\forall 1 \leq i \leq len(Channel.queue^+)$,

  $Channel.queue^+[i] = channel.queue[i+1]$.

  Thus $\forall i, j : Nat, (1 \leq i / \ i < j / \ j \leq len(Channel.queue^+)) \Rightarrow$

  $Channel.queue^+[i].deadline \leq Channel.queue^+[j].deadline$
- fail, timeout : no effect.
- trajectories have no effect.

I_4 : ˜$Detector.suspected \Rightarrow Detector.clock <= u2$

- trajectories : $d(Detector.clock) > 0$ iff $Detector.clock \neq u2$) (stopping condition).
- timeout : $Detector.suspected^+ = true$, thus the statement is trivially true.
- receive(m) : $Detector.clock^+ = 0 < u2$, thus the statement is trivially true.

- send, fail : no effect on either *Detector.clock* or *Detector.suspected*

I_5 :˜$Sender.failed \Rightarrow Sender.clock \leq u1$

- trajectores : $d(Sender.clock) > 0$ iff $Sender.clock) \neq u1$ (stopping condition).
- timeout, receive(m) : no effect on either *Sender.clock* or *Detector.suspected*
- fail: $Sender.failed^+ = true$, and thus the statmeent becomes trivially true.
- send(m): $Detector.clock^+ = 0 < u1$

I_6 :˜$Sender.failed \Rightarrow$
$(Channel.queue = \{\} \Rightarrow (head(Channel.queue).deadline < Channel.now + u2 - Detector.clock)$
$\wedge$
$(Channel.queue \neq \{\} \Rightarrow Channel.now + u1 - Sender.clock + b < Channel.now + u2 - Detector.clock)$

- trajectories : $d(Channel.now) = d(Detector.clock) = d(Sender.clock)$
- fail : $Sender.fail^+ = true$, and thus the condition becomes trivially true.
- timeout: no change brought to any relevant parameter.
- send(m) : let us assume that $Sender.failed$ is $false$. The value of sender.failed is unchanged by the action.

  As a result of send(m), $(Channel.queue^+ \neq \{\}$

  Channel.now and *Detector.clock* are untouched, thus

  - if we previously had $(Channel.queue \neq \{\}$ then
    $head(Channel.queue^+).deadline = head(Channel.queue).deadline$
    and
    $head(Channel.queue).deadline < Channel.now + u2 - Detector.clock^+$.
    These imply $head(Channel.queue^+).deadline < Channel.now^+ + u2 - Detector.clock^+$
  - else $head(Channel.queue^+).deadline = Channel.now^+ + b$
    and $Channel.now^+ + b + u1 - Sender.clock^+ < Channel.now + u2 - Detector.clock$.
    Since $Sender.clock^+ = 0$, $Channel.now^+ + b < Channnel.now^+ u2 - Detectorclock$,
    $head(Channel.queue+).deadline < Channel.now + u2 - Detector.clock$

- receive(m) : $Detector.clock^+ = 0$, *Channel.now* and *Sender.clock* are unchanged.

  Because of the precondition for the receive(m) action, $len(Channel.queue) > 0$ and $Channel.queue \neq \{\}$.

  Thus we always have the following :

  $Channel.now^+ + u1 - Sender.clock^+ + b < Channel.now^+ + u1 + b < Channel.now^+ + u2 - Detector.clock^+$

1. Case 1 : $len(Channel.queue) = 1$. Then $Channel.queue^+ = \{\}$ and

   $$Channel.now^+ + u1 - Sender.clock^+ + b < Channel.now^+ + u2 - Detector.clock^+$$

2. Case 2 : $len(Channel.queue) > 1$. Then $Channel.queue^+ \neq \{\}$ and (I_4) $head(queue^+).deadline \leq Channel.now^+ + b$
   thus $head(queue^+).deadline < Channel.now^+ + u2 = Channel.now^+ + u2_Detector.clock^+$.

I_7 : $Detector.suspected \Rightarrow Sender.failed$.

- trajectories, send, receive have no impact on the states variables at hand.
- fail : $Sender.failed^+ = true$.
- timeout: $Detector.clock = u2$, the only state variable changed is $Detector.suspect$. Thus $head(Channel.queue^+).deadline > Channel.now^+ + u2$ (by applying I_2 with $p = head(Channel.queue^+)$)

  and since (I_5) $u1 - Sender.clock^+ \geq 0$,

  $Channel.now^+ + u1 - Sender.clock^+ + b > Channel.now^+ + u2 - Detector.clock^+$

  thus, necessarily, by (I_6), $Sender.fail = true$.

Finally, there are two small invariants left that will lead to proving that a failure is detected within a time bound of $u2 + b$.
I_8 : $Sender.clock \leq Detector.clock + b$

- trajectories : $d(Sender.clock) = d(Detector.clock)$
- fail, timeout : no impact on eiter $Sender.clock$ and $Detector.clock$.
- send(m) : $sender.clock^+ = 0 < Detector.clock^+ + b$
- receive(m): necessarily, $enQ_q n(queue) = true$ thus, by (I_3) we have :

  $earliest\_deadline(Channel.queue) + Sender.clock \leq Channel.now + b$

  thus $Sender.clock \leq b + channel.now - earliest\_deadline(Channel.queue)$

  and finally, thanks to (I_2), $Sender.clock^+ = Sender.clock \leq b + Detector.clock^+$

I_9 : $Sender.clock > u2 + b \Rightarrow Detector.suspect$ is a simple collorary of I_8 and I_4 :
$Sender.clock > u2 + b \Rightarrow Detector.clock > u2 \Rightarrow Detector.suspect$

References

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

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

[3] Dilsun Kaynar, Nancy Lynch, Sayan Mitra, and Stephen Garland. The TIOA language, version 0.21. Unpublished manuscript, 2005.

[4] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The theory of timed I/O automata. Technical Report MIT/LCS/TR-917a, MIT Computer Science and Artifical Intelligence Laboratory, 2005. Available at http://theory.csail.mit.edu/tds/reflist.html.

[5] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *Theory of Timed I/O Automata.* Synthesis Lectures on Computer Science. Morgan-Claypool Publishers, May 2006. Also, revised and shortened version of Technical Report MIT-LCS-TR-917a (from 2004), MIT Laboratory for Computer Science, Cambridge, MA.

[6] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Journal of Software Tools for Technology Transfer*, 1–2:134–152, 1997.

[7] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1996.

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

[9] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC 1987)*, pages 137–151, Vancouver, British Columbia, Canada, August 1987.

[10] 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.

[11] Sayan Mitra. *A Verification Framework for Ordinary and Probabilistic Hybrid Systems.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2007. To appear.

[12] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer Verlag, 1996.