Uppaal Tron User Manual

Kim G. Larsen, Marius Mikučionis, Brian Nielsen
CISS, BRICS, Aalborg University, Aalborg, Denmark
{kgl, marius, bnielsen}@cs.aau.dk

June 17, 2009

Abstract

Uppaal Tron is an online model-based testing tool for real-time systems. This user-manual documents the implementation features of the tool and could also be used as a reference manual for building test adapters for Tron. The reader should be familiar with Uppaal tutorial [1]. Basic knowledge of process control in a shell and programming in C/C++ or Java is assumed.

Contents

1 Introduction

Uppaal Tron implementation started as part of Master thesis project and continued as part of Ph.D. thesis project by Marius Mikučionis, supervised by Kim G. Larsen and Brian Nielsen. The tool is being applied and evaluated in research, education and industrial case studies and yet is being improved.

The manual is organized in the following way: we introduce the tool in this section, discuss the system modeling assumptions, describe the test adapter framework, explain the options and diagnostic messages and outline some future work. We recommend to get accustomed to Tron through Section 1.3, proceed with formal and practical framework setup in sections 1.4, 1.5, 2 and use sections 3, 4, 5 as reference manual. Faults and feature requests should be reported to Uppaal bug tracking system: http://bugsy.grid.aau.dk/cgi-bin/bugzilla/index.cgi.

The following subsections describe features and requirements of Uppaal Tron, look'n'feel of the tool and how to get started with the demo, finally explain the formal concepts used in Tron.

1.1 Features

- Performs conformance testing: the tool checks whether the timed runs of the system under test (SUT) are specified in the system model (similar to timed trace inclusion) and no illegal (unexpected, unspecified) timed behavior is observed.

- The emphasis is on testing the timed and functional properties. Time is considered continuous, (input/output) events can happen at any real-valued moment in time, but deadlines are constrained by integers (rationals). Test data generation is also possible, but (today) data types and value selection are limited by modeling language.

- The specification is an Uppaal timed automata network partitioned into a model of the system and a model of system's environment assumptions. The model can be non-deterministic, allowing reasonable freedom for system implementations, modeling possible/tolerable time drifts, soft time deadlines.

- Test primitives are generated directly from the model, executed and the system responses checked at the same time, online (on-the-fly) while connected to the SUT, thus avoiding huge intermediate test suites.

- During testing the tool follows the environment model which can have various purposes:

  1. fully permissive environment model allows to test full conformance;
  2. a specific environment minimizes the testing effort for realistic level of conformance;

3. environment model as use cases guide through functionality of a particular interest;
4. environment model as pre-recorded test runs used to re-execute tests for debugging or regression testing.

- UPPAAL model-checking engine allows efficient and fast timed automata model exploration.
- If the environment model is non-deterministic (very often it is) then choices of inputs and time delays are randomized. So far, early experiments show that randomization results in the best location, edge and variable value coverage as opposed to heuristics based on location, edge, variable value or combined strategies. This however does not mean that offline test generation techniques cannot be better.
- In general, testing the real-time conformance is undecidable, but under digitization assumptions it is shown to be sound and complete in a time limit.

1.2 Requirements

Minimal requirements:

1. Architecture: PC, Intel Pentium compatible.
2. Operating system: Linux (2.6 version recommended) or Microsoft Windows NT/2000/XP/2003. Releases are tested on Debian GNU/Linux testing/unstable and Windows XP Professional.

Binaries for Sun Solaris (SunOS 5.10) on Sparc can be provided upon request.
Optional:

3. Sun Java 5 or 6 Software Development Kit (SDK) for java examples.
4. Apache ANT for java examples.
5. Graphviz [3] utilities for model signal-flow diagrams layouts in pictures.
6. R language and environment for statistical computing and graphics for displaying scheduling latency experiment results.
7. GhostViewer `gv` for displaying PostScript pictures generated from scheduling latency experiment.
8. GNU Compiler Collection and GNU make for dynamic library adapters (button example) on Linux.
9. Microsoft Visual Studio 2005 for dynamic library adapters (MSVC button example) on Windows.

Other software assumed:

9. ZIP archive extractor: `unzip` on Linux and Windows Explorer or WinZIP on Windows.

10. Terminal or command line prompt: `xterm` with `bash` on Linux, `cmd.exe` on Windows.
11. GNU tool set (GNU Make from Linux distribution or MinGW or Cygwin) can be used to gain an advantage of automatic build and execution `Makefile` scripts included with TRON distribution.

Linux software is available on Debian GNU/Linux via single command:

```
apt-get install sun-java6-jdk graphviz r-base gcc g++ make gv xterm
```

1.3 Getting Started

The section demonstrates how to use the tool by running a smart-lamp demo with a few mutant examples. Other examples are available through Makefile scripts which can be used with GNU make.

The following steps prepare to use the tool for your operating system.

1.3.1 Installation for Linux

1. Download UPPAAL TRON from a TRON webpage. Choose "TRON-V for Linux on Intel PC", where V is the latest version number. Some versions are marked as alpha (internal development releases) and beta (preview releases for general public), which denote the maturity and the feature completeness of the release. Please also see the version history on the download page.
2. Start terminal or command line window: launch terminal application `xterm`.
3. Check if the proper Java version is installed (i.e. if the environment variable PATH is set correctly and GNU Java[1] is not in the way): command `java -version` should show something like the following:
   ```
   java version "1.6.0"
   Java(TM) SE Runtime Environment (build 1.6.0-b105)
   Java HotSpot(TM) Client VM (build 1.6.0-b105, mixed mode, sharing)
   ```
4. Unpack UPPAAL TRON: enter `unzip uppaal-tron-V-linux.zip` at command prompt.
5. Go to `tron java` directory: `cd uppaal-tron-V-linux/java`.
6. Start another terminal in the same directory: enter `xterm &`.

1.3.2 Installation for Windows

1. Download UPPAAL TRON from a TRON webpage. Choose "TRON-V for Windows", where V is the latest version number. Some versions are marked as alpha (internal development releases) and beta (preview releases for general public), which denote the maturity and the feature completeness of the release. Please also see the version history on the download page.

[1]Some Linux distributions ship GNU Java as default Java, which is known not to work with TRON SocketAdapter and can be changed to Sun Java by administrator via `update-alternatives` or `galternatives` programs.

2. Start terminal or command line window: click Start→Run, type `cmd.exe` and hit ENTER.

3. Check if the proper Java version is installed (i.e. if the environment variable PATH is set correctly: command `java -version` should show something like the following:
```
java version "1.6.0"
Java(TM) SE Runtime Environment (build 1.6.0-b105)
Java HotSpot(TM) Client VM (build 1.6.0-b105, mixed mode, sharing)
```

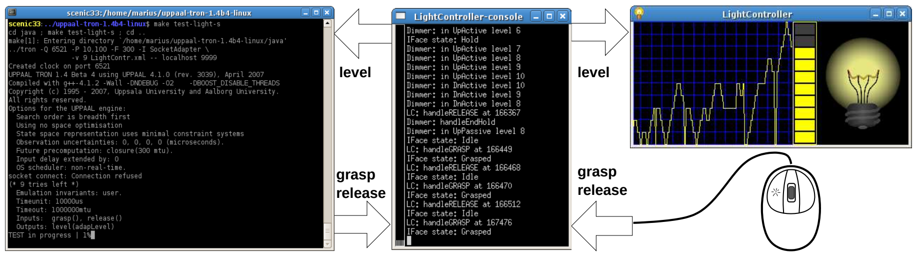
4. Unpack Uppaal Tron: use Windows Explorer or WinZIP to extract.

5. Go to `tron java` directory: `cd uppaal-tron-V-linux/java`.

6. Start another command line window in the same directory: enter `start cmd.exe` at command prompt.

1.3.3 Smart-lamp Demo

Figure 1 shows the smartlamp test setup. The LightController is the main



```
scenic33:../uppaal-tron-1.4b4-linux$ make test-light-s
cd java ; make test-light-s ; cd ..
make[1]: Entering directory `/home/marius/uppaal-tron-1.4b4-linux/java'
../tron -Q 6521 -P 10,100 -F 300 -I SocketAdapter \
                -v 9 LightContr.xml -- localhost 9999
Created clock on port 6521
UPPAAL TRON 1.4 Beta 4 using UPPAAL 4.1.0 (rev. 3039), April 2007
Compiled with g++-4.1.2 -Wall -DNDEBUG -O2    -DBOOST_DISABLE_THREADS
Copyright (c) 1995 - 2007, Uppsala University and Aalborg University.
All rights reserved.
Options for the UPPAAL engine:
  Search order is breadth first
  Using no space optimisation
  State space representation uses minimal constraint systems
  Observation uncertainties: 0, 0, 0, 0 (microseconds).
  Future precomputation: closure(300 mtu).
  Input delay extended by: 0
  OS scheduler: non-real-time.
socket connect: Connection refused
(* 9 tries left *)
  Emulation invariants: user.
  Timeunit: 10000us
  Timeout: 1000000mtu
  Inputs:  grasp(), release()
  Outputs: level(adapLevel)
TEST in progress | 1%
```

```
Dimmer: in UpActive level 6
IFace state: Hold
Dimmer: in UpActive level 7
Dimmer: in UpActive level 8
Dimmer: in UpActive level 9
Dimmer: in UpActive level 10
Dimmer: in DnActive level 10
Dimmer: in DnActive level 9
Dimmer: in DnActive level 8
LC: handleRELEASE at 166367
Dimmer: handleEndHold
Dimmer: in UpPassive level 8
IFace state: Idle
LC: handleGRASP at 166449
IFace state: Grasped
LC: handleRELEASE at 166468
IFace state: Idle
LC: handleGRASP at 166470
IFace state: Grasped
LC: handleRELEASE at 166512
IFace state: Idle
LC: handleGRASP at 167476
IFace state: Grasped
```

Figure 1: Smartlamp setup: LightController (in the middle) connected to Tron (on the left), level view window and a mouse (on the right).

executable class. The application has two interfaces: for graphical user interface (GUI) and for Tron. GUI shows level of the light as different color shades on a light bulb, adjusts a level bar and draws level history chart. GUI window sends `grasp` and `release` signals to LightController whenever GUI window is pressed or released with left button of a mouse. The LightController console prints the events happening in the application. Tron can be attached to LightController via `SocketAdapter` with an equivalent interface of `grasp` and `release` as inputs and `level` as output. Tron window shows the progress of the test run. The following is a list of commands demonstrating smartlamp application and Tron tests against it.

One can experiment with LightController via GUI without running Tron by entering the following command line:
`java -cp dist/smartlamp.jar com.uppaal.smartlamp.Main -M` 0

To run Tron test demo in virtual time framework[2] against smartlamp follow these steps:

[2]Mouse clicks are ignored here since the user is not part of virtual time framework.

1. Start smart-lamp at one command prompt:
   `java -cp dist/smartlamp.jar com.uppaal.smartlamp.Main -C localhost 8989 -M 0`

   `-C localhost 8989` sets the virtual clock to TCP/IP socket located at local host port 8989.

   `-M 0` sets mutant 0 (correct implementation) to be run.

2. Start TRON from another command prompt:
   `../tron -Q 8989 -P 10,200 -F 300 -I SocketAdapter -v 9 LightContr.xml -- localhost 9999`

   `-Q 8989` creates virtual clock on TCP/IP socket at local host port 8989.

   `-P 10,200` limits the delay choices up to 10 or 200 time units (this prevents choices of very long delays).

   `-F 300` tells to pre-compute a symbolic state set for 300 time units into the future (allows more choices from the near future).

   `-I SocketAdapter` tells to use built-in SocketAdapter.

   `-v 9` tells to (+1) to print only the progress of testing and (+8) backup the state set for verdict diagnostics in case the test fails.

   `LightContr.xml` tells to use LightContr.xml file as test specification.

   `-- localhost 9999` is a parameter to adapter, tells SocketAdapter to connect to implementation on TCP/IP socket at local host port 9999.

Run test demo in real time:

1. Start smart-lamp on one command prompt (`-C` is not used):
   `java -cp dist/smartlamp.jar com.uppaal.smartlamp.Main -M 0`

2. Start TRON on another command prompt (`-Q` is not set):
   `../tron -u 4000,4000 -P 10,200 -F 300 -I SocketAdapter -v 9 LightContr.xml -- localhost 9999`

Note that GUI mouse clicks can be used to alter the behavior of LightController in real time, hence introducing behavior mutations which may be sensed by TRON. See also Section 6 if TRON reports test failures on mutant M0 in real time.

1.3.4 Smart-lamp Mutant Exercise

For the smart-lamp mutant exercise you need the model `LightContr4.xml`, and the following command lines to start TRON and the controller:

```
../tron -Q 8989 -P 10,200 -F 300 -I SocketAdapter -v 10 LightContr4.xml --
localhost 9999
java -cp dist/smartlamp.jar com.uppaal.smartlamp.Main -C localhost 8989 -M
0
```

There are two built-in faulty mutants controlled by `-M` option: `-M 1` and `-M 2`.

The easiest way to create your own mutants is to modify the existing Light-Controller source and add mutants in the style of the existing mutants (a flag indicates what mutant to run, and use `if (mutantID)` statements to enable the faulty code. One typically needs to edit the `com/uppaal/smartlamp/SmartLamp.java` and/or `com/uppaal/smartlamp/Dimmer.java` files in `src` directory. Remember to recompile the smartlamp once edited: `ant clean jar`

1.3.5 Offline Generated Tests

We recommend executing your preset input sequences using Tron by modeling the test input/output sequence as a timed automaton and by replacing the environment with this automaton. Depending on desired timing choices Tron can be run in random, eager, lazy or bounded delay mode. An example is provided in LightContr4.xml (Template: LightCov and Envy Closure, see system section of the model). Start Tron as described below, try eager and other delay options:

```
../tron -Q 8989 -P eager -F 300 -I SocketAdapter -v 8 LightContr4.xml -- localhost
9999 silent
../tron -Q 8989 -P 10,200 -F 300 -I SocketAdapter -v 10 -w 20 LightContr4.xml
-- localhost 9999
../tron -Q 8989 -P random -F 300 -I SocketAdapter -v 8 LightContr4.xml -- localhost
9999 silent
../tron -Q 8989 -P lazy -F 300 -I SocketAdapter -v 8 LightContr4.xml -- localhost
9999 silent
```

1.3.6 Create Your Own Smart-lamp

Here you have to create both a model and an implementation. It is easiest to start with the template given in `onOffLight.xml` and `OnOffLightController.java`:

```
java -cp dist/smartlamp.jar com.uppaal.autoofflamp.Main -C localhost 8989 -M
0
../tron -Q 8989 -P 10,200 -F 300 -I SocketAdapter -v 10 onOffLight.xml -- localhost
9999
```

1.3.7 Create Your Own Implementation

The package `com.uppaal.dummy` includes a minimal dummy implementation that behaves like `dummy.xml` specification. `TestIOHandler` takes care of input/output translation and communication, where `DummyInterface` declares what kind of input methods `Dummy` implementation can handle and `DummyListener` interface declares what output methods are available. Once the source of the dummy is modified, the new implementation can be compiled by invoking ANT rule `ant clean jar` which reads `build.xml` script, cleans the old byte code, compiles and packages a new jar package in `dist` directory. The resulting package can be optimized and obfuscated with ProGuard by invoking `ant clean dist`.

1.4 Relativized Timed Conformance

Tron uses rtioco as implementation relation to specification in order to evaluate the correctness of a test experiment and to determine the test verdict. rtioco is an extension to tioco which in turn has roots in ioco by Jan Tretmans []. Explicit handling of environment assumptions is an essential feature which distinguishes rtioco from other timed conformance variations and still compatible with ultimate qualities of tioco. The environment assumptions give additional information about specific kinds of implementation behavior and help tester to focus on features of interest, closer reflect reality and hence reduce testing costs.

Definition 1 augments the formal definition of rtioco [7] with engineering interpretation, which means that implementation p conforms to specification s within the environment e if and only if the observations from test execution on $\langle e, p\rangle$ are always included in possible observations described by specification $\langle e, s\rangle$ while running all possible traces of environment e.

Definition 1 Relativized timed input/output conformance relation *for input enabled timed input/output labeled transition systems $p, s \in \mathcal{S}$ and $e \in \mathcal{E}$:*

$$p \ \mathsf{rtioco}_e \ s \ \stackrel{def}{=} \ \forall\sigma \in \mathsf{TTr}\big(e\big).\mathsf{Out}\big(\langle e,p\rangle \ \mathsf{after}\ \sigma\big) \subseteq \mathsf{Out}\big(\langle e,s\rangle \ \mathsf{after}\ \sigma\big) \quad (1)$$

where:

$\mathcal{S}$ *and* $\mathcal{E}$ *are the sets of timed input/output labeled transition systems that are compatible with respect to observable inputs and outputs:* $\mathcal{S}$ *observable outputs synchronize with observable inputs of* $\mathcal{E}$ *and vice-a-versa,*

p,s and e are initial states of implementation under test, specification and environment respectively,

$\mathsf{TTr}\big(e\big)$ *is a set of timed input/output traces of e,*

$\langle e,p\rangle$ *and* $\langle e,s\rangle$ *are parallel compositions of p and e, and s and e, respectively, where processes synchronize on observable input/output action transitions,*

$\langle e,p\rangle$ after σ *means executing an observable trace σ on implementation p within environment e and returning the end state(s) of the system,*

$\langle e,s\rangle$ after σ *means evaluating an observable trace σ on specification s within environment e and returning a set of possible system specification states,*

$\mathsf{Out}\big(\text{states}\big)$ *return the list of possible output action and/or delay observations.*

Notice that the definition mentions environment twice: firstly composed with implementation (real physical entity) and secondly composed with specification (virtual abstraction or modelled entity). Formally (and ideally) these environments are the same (hence only one e is needed), but in practice it is the tester's responsibility to transform the modelled environment into the real physical entity, which means providing adapters with physical interface to implementation and behaving like environment model.

Let us examine possible cases and see why this relation is good for defining the correctness of timed behavior in black-box testing:

1. Definition is provided for timed labeled input/output transitions, which means that it is applicable to a broad class of timed systems (e.g. hybrid systems), not just the ones modelled by timed automata and is independent of modelling formalisms. Definition also does not go deeper nor dwells about the structure of p, s and e processes: no assumptions about them are made, high-level abstract specifications s and e are possible allowing all kinds of non-determinism, does not measure the state of p directly allowing black-box testing, s, e and p can be composed of many parallel processes which allow modular designs of the system and the specification.

2. Follows common intuition that outputs should be observed as they are described in the specification: neither too early nor too late if allowed at all. If tester observes delay $\delta \in \mathbb{R}_{\geq 0}$ followed by output $o \in A_{out}$ from implementation after trace σ then it means $\delta \in \mathsf{Out}\big(\langle e, p\rangle \ \mathsf{after}\ \sigma\big)$ and $o \in \mathsf{Out}\big(\langle e, p\rangle \ \mathsf{after}\ \sigma\delta\big)$. The tester should compute the largest delay d such that $d \in \mathsf{Out}\big(\langle e, s\rangle \ \mathsf{after}\ \sigma\big)$ and check whether $\delta \leq d$:

   - if $\delta \leq d$ is false then it means that specification did not allow to delay for δ times, and p does not conform to s. However, if $o \in \mathsf{Out}\big(\langle e, p\rangle \ \mathsf{after}\ \sigma d'\big)$ for some $d' \leq d$, then it means that output is allowed but observed too late (later than required after d').
   - if $\delta \leq d$ is true then $o \in \mathsf{Out}\big(\langle e, p\rangle \ \mathsf{after}\ \sigma\delta\big)$ has to be checked:
     - if true then output o is allowed and should be appended to σ trace
     - if false then output o is not allowed. However if there is d' such that $o \in \mathsf{Out}\big(\langle e, p\rangle \ \mathsf{after}\ \sigma d'\big)$ and $d' > \delta$ then it is likely that o is allowed but is observed too early (earlier than delay d'). Another possibility is that there exists $d'' < \delta$ after which o is allowed, then observation can be classified as o is allowed but observed too late (later than after delay d'').

3. Definition allows incremental test trace construction, see the output observation discussion above which also holds for input events.

4. Relation considers only the traces that are possible in environment e which gives us the power to test the selected timed behavior. The input enableness of e guarantees that any output produced by p or s is accepted and not refused, hence does not influence the correctness. There are two interesting extreme cases of environments:

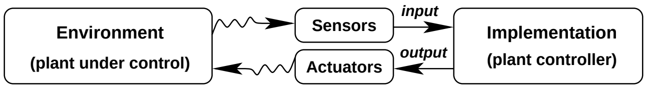
   (a) Universal environment e_U which allows all observable timed traces: $\mathsf{TTr}\big(e_U\big) = (A_{inp} \cup A_{out} \cup \mathbb{R}_{\geq 0})^*$. Then $p\ \mathsf{rtioco}_{e_U}\ s$ coincides with timed trace inclusion and is equivalent to $p\ \mathsf{tioco}\ s$.

   (b) Silent environment e_S which does not allow any inputs but merely consumes outputs and lets the time pass: $\mathsf{TTr}\big(e\big) = (A_{out} \cup \mathbb{R}_{\geq 0})^*$. This is the same as $A_{inp} = \emptyset$ where tester is allowed only to observe the behavior of implementation. Such activity is equivalent to passive monitoring of the system.

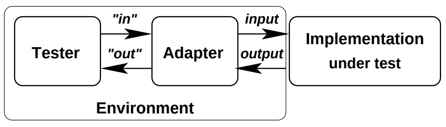
In theory black-box timed testing is undecidable due to (timed trace) language inclusion checking problem, however in [7] the online test generation algorithm for real-time systems is shown to be sound and also complete (exhaustive) under input-enableness, observability and digitization assumptions if given enough time. The assumptions are important only for theoretical completeness and can be relaxed in practice.

1.5 Online Test Setup

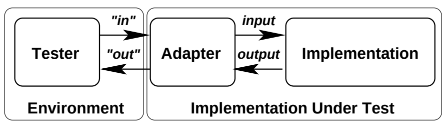
We consider closed systems, where implementation together with its environment can be isolated from the rest of the world. Figure 2a shows typical system setup during the system deployment: environment is a plant that needs to be steered and controlled, and implementation is a software/hardware controller taking inputs from the sensors embedded in the environment and producing output to actuators influencing the environment. Notice that we take the perspective of the controller or implementation when talking about inputs and outputs.



(a) System during deployment.



(b) IUT's perspective during testing.



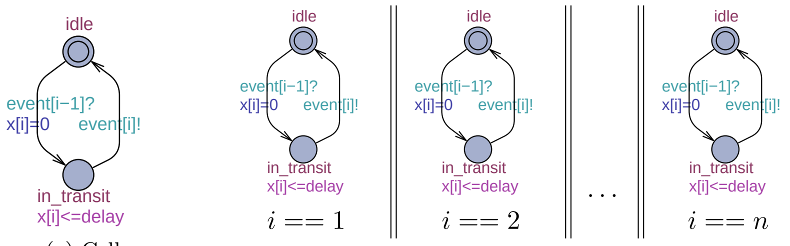
(c) Tester's perspective during testing.

Figure 2: Implementation during deployment and testing.

In Figure 2b we replace the environment, sensors and actuators with a tester and a test adapter in order to test such controller. In generic test setup the adapter translates abstract input messages into physical actions and recognizes physical outputs and encodes them into abstract messages understood by the tester. The adapter is always implementation specific, moreover adapter implementation may also contain faults, hence we arrive to TRON test setup shown as tester's perspective in Figure 2c where the adapter is shifted to be a part of the implementation under test. We rely on the assumptions that adapter is fast enough to mimic sensors and actuators and tester is fast enough to emulate the environment and therefore provide fair tests.

The system model provided as test specification should also reflect the physical setup and partitioning of component-processes as shown in Figure 2c. The inputs are controlled by the tester and the outputs are controlled by the implementation. While modelling the IUT requirements and environment assumptions is rather straightforward, the model of an adapter is often overlooked. In TRON framework we follow the semantics of time automata specification defined as labelled transition systems, where events (edge-transitions) happen atomically and instantaneously. Therefore we also treat an event as a single point in time and space, where the time defines when the event happened (relatively to the start of testing), space-location defines a component of the system and action label identifies an edge of the component process. Notice that a simple electronic signal traveling via wire corresponds to a series of events at different locations of the wire. Ultimately, physical reality does not allow measuring location and time of event precisely (precise timing cannot be measured if the location is known precisely and precise location cannot be measured at precise timing), moreover it is not possible nor desired to provide models at such detailed level, hence a reasonable abstraction is needed which still captures the important details.

First, we propose to split input/output action into two events: 1) when input action is sent by the tester (output action is sent by implementation) and 2) when input action is received by implementation (output action received by tester); this will make sure that input and output actions can pass each other as in asynchronous distributed systems. Second, model the adapter as an event buffer. One size buffer is a cell shown in Figure 3a and n-size buffer is a parallel composition of n cells composed in a sequence as in Figure 3b. Based



(a) Cell.

(b) n-size buffer.

Figure 3: Buffer automata for the adapter model, where x[i] is a clock.

on a concrete value of `delay` and on assumptions on how many actions can be generated at the same time, one can find minimal buffer size n and using [6, 5] techniques prove that such buffer is a correct abstraction of a physical one (down to atomic details).

While the input part of adapter is important for the implementation input-enableness assumptions and reflecting the possible delay in signal, the output part of adapter is merely delaying the output but has severe performance penalty if the buffer is large, hence should be kept as simple as possible.

Tron uses interval time-stamping in order to solve the problem of precise time-measuring: the action is time-stamped at the tester's interface to adapter and the time-stamp is converted to model time interval, whose bounds are the closest integers to measured time-stamp. This reflects our notion that we don't really know when the event actually happened, but somewhere in the interval, and allows us to compute an over-approximation of actual behavior of the system. The over-approximation enforces the principle "behavior is correct unless proved otherwise" and it does allow some non-conforming behavior to pass the test, but we think that it is reasonable given that the observability (ability to measure the timings) and controllability (ability to feed inputs at precise timing) are not perfect as one could expect in theory.

2 Test Specification

A Tron test specification consists of the following items:

- Uppaal model containing requirements for environment and IUT processes,
- input/output channel interface between environment and IUT processes,
- model time unit definition and
- amount of time dedicated for testing.

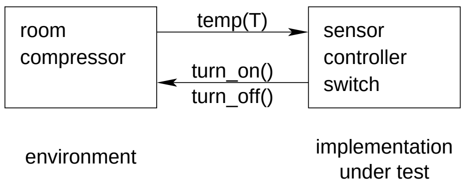


Figure 4: Fridge model setup.

We will use the fridge system from Figure 4 as a running example to demonstrate how typical system model is composed for testing using Tron. The fridge system consists of five processes: room, sensor, controller, switch and compressor. The room process controls the room temperature of the fridge: a sample room automaton is displayed in Figure 5b. The sensor process identifies whether the sensed temperature is *High*, *Med* or *Low*, see the timed automaton in Figure 5c. The controller process is controlling whether the compressor should be turned *On* or *Off* via shortcutting a switch, see Figure 5d. The switch process is relaying the signal to compressor by *turn_on* and *turn_off* like automaton in Figure 5e. The compressor process is responsible for notifying the room about the change of conditions in the fridge, i.e. if *compr* is true then the heat is taken away by the circulating liquid and if false then the heat is leaked into the fridge, see Figure 5f and Figure 5b. Assume that we want to test the software running in the controller component of our fridge system. The only way to connect to controller is through the sensor and switch interfaces as there is no "direct" connection with the controller process. Notice that the sensor and the switch introduce the communication latency[3], which is reflected by the upper bound of d time units in sensor and switch automata. Hence, the controller, the switch and the sensor models belong to the IUT requirements as there is no way to separate them. The rest of the processes (the room and the compressor) belong to assumptions about environment of IUT.

2.1 Properties of the Model

Tron allows non-determinism in the model. For some models the resulting state space can even be beyond the verification. For example, the requirements for the controller in Figure 5d are non-deterministic in two ways:

1. in action: the location *up* is allowed to be reached after *Med* or *High* actions. Similarly the location *dn* can be reached from *on* by any of *Low* or *Med* actions. Modeling that the IUT is allowed to implement either sequence.

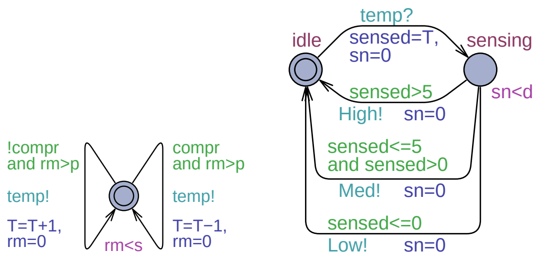
2. in time: the controller may stay in locations *up* and *dn* for any time duration up to r time units. Modeling allowed reaction time tolerance.

Moreover the communication latency in adapter adds even more unavoidable (concurrency) non-determinism to the IUT requirements. Similarly the environment processes can also be non-deterministic, e.g. the room is allowed to

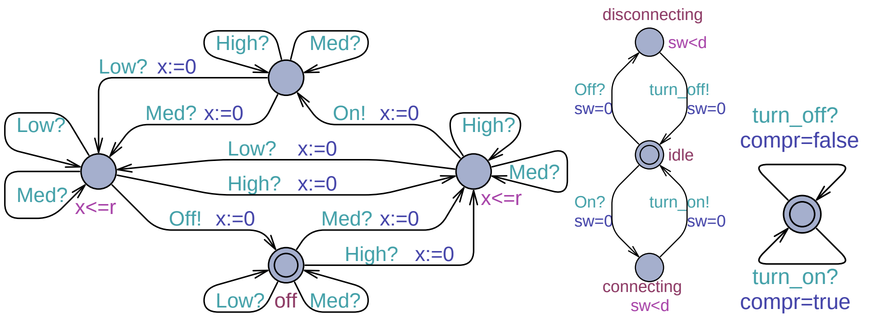
[3]Even tiniest latency is relevant as it models the concurrent nature of independent input and output signals.

```
// IUT requirements:
const int r=15;
int sensed=0;
clock x, sn;
chan High, Med, Low, On, Off;
// observable (test interface) part:
chan temp; // inputs
int T=0; // data bound to input
chan turn_on, turn_off; // outputs
// environment assumptions:
const int p=5;
const int s=30;
const int d=1;
clock sw, rm;
bool compr;
```

(a) Global declarations.



(b) room (c) sensor



(d) controller (e) switch (f) compressor

Figure 5: Model of the refrigeration system, `fridge.xml`.

update the temperature in any periods of time between p and s time units. The sensor automaton makes sure that the input (temperature changes) will always be accepted by IUT part if offered no more often than d time units intervals. Similarly the compressor automaton can accept the output at any time.

The more non-deterministic environment model is, the more discriminative power it has. Generic environments which allow any input fed at any time are the most discriminative, although they are not always practical in testing. Our room and compressor automata model a more realistic environment, where the room temperature is responsive to the state of compressor. We can also replace the room and the compressor by an automaton modelling a concrete test case which could drive the system into interesting states.

The IUT model should be at least weakly input enabled (ability to consume any input at any time) although there are no precise guidelines on how strictly this requirement should be enforced and Tron will try to obey the assumptions in IUT model. The environment model is not required to be input enabled (to accept any output at any time from IUT) and the verdict *inconclusive* will be given if the environment state can not be updated with unexpected IUT output.

2.2 Partitioning of the Model

Input/output channels partition the Uppaal model processes and variables into environment and implementation. The goal of partitioning is to ensure that the setup of real environment and IUT is correctly reflected in the model and only the observable channels are used for communication between the two. The duration of model time unit specifies how much of the real world time in microseconds elapses when Uppaal clock gets incremented by one. The maximum amount of desired testing time is specified by "timeout for testing" in model time units (one Uppaal clock increment).

Currently the procedure for partitioning the system is by specifying input/output channel interface. The partitioning should be consistent (no process/variable should be assigned to both environment and IUT) and complete (all processes should belong to either environment or IUT). Given a user defined set of observable I/O channles, Tron attempts to partition a model of a whole system by iteratively applying the following rules:

- Events on input/output channels are observable and events on other channels (that are not declared as inputs/outputs) are non-observable or internal.
- Internal channel belongs to environment if it is used by an environment process. Respectively, internal channel belongs to IUT if it is used by IUT process. The model is inconsistent and cannot be partitioned if the internal channel is used by both environment and IUT.
- Process belongs to the environment if it uses the internal environment channel respectively. Respectively, process belongs to IUT if it uses the internal environment channel.
- A variable belongs to the environment if it is accessed by an environment process without observable input/output channel synchronization. Respectively, a variable belongs to the IUT if it is accessed by IUT process without observable input/output channel synchronization. A variable is not categorized (allowed to be either) if accessed consistently during observable input/output channel synchronization.
- Process belongs to environment if it accesses environment variable without observable channel synchronization. Respectively, process belongs to IUT if it accesses IUT variable without observable channel synchronization.

If the partitioning is not consistent or incomplete Tron will complain with warnings.

Tron also uses the partitioning to identify environment invariants from IUT invariants for accurate environment emulation, where otherwise all invariants would be treated globally (according to Uppaal timed automata semantics) and IUT invariant would force Tron to take action before it is violated. When interface configuration is done, Tron outputs the list of environment processes whose invariants are used in environment emulation.

In practice to help getting the partitioning accepted by Tron, the `-i dot` option can be used to produce a decorated signal flow diagram that can be visualized by *graphviz* [3] tools. This option expects I/O channels fed by the

following EBNF rule:
`"input"` *(channel)*∗ `"output"` *(channel)*∗
The option will also accept the text following the *preamble* rule from Figure 16 (all parameters in parenthesis are ignored). The end of the input stream is detected by keywords `precision` or `timeout`, or simply by end-of-file signal. The output stream can be laid-out and visualized graphically by *dot*[4] [2]. The diagram shows how processes are communicating similarly to UML deployment diagram [] except that associations are displayed as arrows indicating the direction of signal flow. Diagrams have the following legend:

⬭ represents a process.

□ represents a data variable (clock or integer).

◇ represents an internal channel.

◈ represents an observable channel.

→ represents a signal flow: from a process to a channel – the process is transmitting on the channel, from a channel to a process – the process is receiving on channel, from a process to a variable – the process is updating (writing to) the variable, from a variable to a process – the process is reading value of the variable. The transmitting and updating arrows are bold. The label above arrow enumerates the simultaneous channel synchronizations during data update, dash denotes an update without a channel synchronization (internal transition).

blue items (processes, variables and channels) belong to IUT.

green items (processes, variables and channels) belong to environment.

gray items may belong to either IUT or environment. Gray data variables are good candidates for value passing over channel.

red items could not be partitioned consistently or have some suspicious properties (like variable is updated but is never read).

The error stream is allocated for warnings and errors. The verbosity of error stream is controlled by `-v` option: 0 (none), 1 (only errors), 2 (only errors and warnings), 3 (diagnostic trace of partitioning with errors and warnings).

Example. Suppose the system model is provided in `fridge.xml` file and the test interface is specified in `fridge.trn` file shown in Figure 6a. Then the partitioning image `fridge.eps` and partitioning diagnostics can be obtained by the following `bash` command line:

```
tron fridge.xml -i dot -v 3 < fridge.trn | dot -Tps -o fridge.eps
```

The command executes TRON with system model `fridge.xml`, asks for partitioning in dot format (`-i dot`), sets the error stream verbosity level to all diagnostics (`-v 3`), feed the interface description as input stream from `fridge.trn` file. The output stream with graph data is redirected to *dot* process which is asked to produce PostScript (`-Tps`) image of the graph layout and write it to `fridge.eps` file (`-o fridge.eps`). The user should observe diagnostics in the

```
input temp(T);
output turn_on(),
        turn_off ();
precision 1000;
timeout 10000;
```

(a) `fridge.trn`

```
Inputs: temp
Outputs: turn_off, turn_on
Adding "room" using "temp" by rule "transmitters on
        input channels belong to Env"
Adding "compressor" using "turn_off" by rule "receivers
        on output channels belong to Env"
Adding "sensor" using "temp" by rule "receivers on input
        channels belong IUT"
Adding "High" because of "sensor" by rule "internal
        channel belongs to IUT if it is used by IUT"
Adding "Low" because of "sensor" by rule "internal
        channel belongs to IUT if it is used by IUT"
```

(b) Diagnostics sample.

Figure 6: The files in automatic model partitioning

error streams whose content is similar to Figure 6b. The first two lines of Figure 6b show the input and output channels separated by comma. The later lines show which items were partitioned using a particular rule. If the partitioning is not successful, the user should look at the diagnostics, find the first line where process, channel or variable was assigned to wrong side and fix the problem in the model. Figure 7 shows the sample image of the partitioning. The image

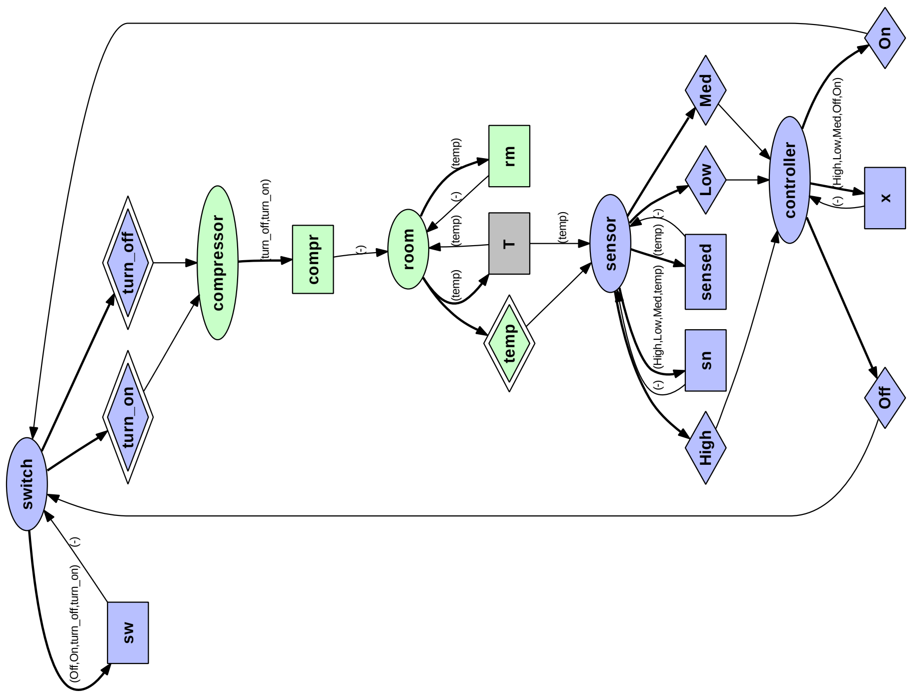


Figure 7: Decorated signal flow diagram (`fridge.eps`) of the system model.

might have different layout each time it is generated as *dot* gets different initial random seed.

[4]The other utilities can also be useful, but *dot* usually gives the best results as quality of the layout depends on the minimization of edge crossings (NP-hard problem).

3 System Adaptation for Testing

The test system developer must provide a test adapter in order to adapt the system for testing. The adapter is responsible for translating symbolic input descriptions into concrete physical input actions, recognizing physical outputs and translating them back to symbolic output representations that testing tool understands. The TRON driver implements `Reporter` interface which is used to configure test interface (define observable inputs and outputs in the model) and report the outputs detected by adapter. The `TestAdapter` interface is used by TRON driver to feed the inputs. Figure 8 shows the interface between TRON and the test adapter: the TRON driver exports a `Reporter` interface which is referenced by adapter component and adapter is exporting a `TestAdapter` interface which is referenced by driver component. The connection establishment, test interface configuration and physical I/O are adapter implementation specific.

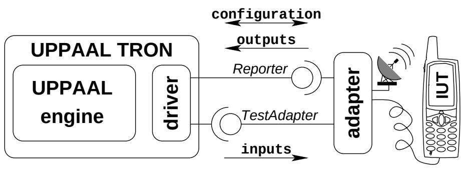


Figure 8: Adapter API and physical interface.

The adapter is specified by `-I name` command line option where `name` is the name of the adapter. If the adapter is provided in a dynamically linked library then the name refers to the library file name. The adapter may support command line arguments too: the adapter parameters are specified at the end of TRON command line starting with double dash `--`, otherwise the adapter will get an empty list of arguments.

Table 1 summarizes advantages and disadvantages of adapter APIs. Textual API (Section 3.5) is probably the easiest way to communicate with TRON which does not require any software development skills except knowledge of the trace format, however it is slow due to continuous I/O stream parsing and encoding. DLL API (Section 3.1) is the fastest as adapter and TRON share the same memory space and hence I/O copying is minimized, however it requires low level C programming knowledge, careful memory management and tedious thread programming. TCP/IP (Section 3.3) seems to be a fair trade off between the previous two: it can be used with almost any programming language, it provides perfect process isolation and it is relatively fast.

Adapter API	**DLL**	**TCP/IP**	**Textual**
Technology	Executable linking	Networking	Standard I/O streams
Performance	Fastest	Limited by network	Slow due to parsing
Flexibility	Architecture specific	Cross platform	Platform independent
Isolation	All resources shared	Remote process	Operating system
Tools	C/C++	Socket programming	Text editor, TRON

Table 1: Brief comparison of supported adapter APIs.

In addition we provide sample Java adapter implementation using TCP/IP API in a way that it hides the complexity of socket programming and provides pure Java API (Section 3.4).

3.1 Dynamically Linked Library (DLL) Interface

Dynamic library interface is the most intimate connection to TRON as the user-supplied adapter is loaded into TRON process address space and events are transfered via function calls. The adapter name is a path to a dynamically linked library file. The path can be either absolute or relative: at first, TRON driver attempts to load a library at specified path as host's dynamic linker (`ld.so(8)` on Linux) is configured (e.g. use LD_LIBRARY_PATH etc.) and if it fails it attempts to load it relatively from the current directory assuming that the file is in the current directory. Here we will assume that the C language is chosen to develop a dynamic library adapter.

Figure 9 shows the symbol signatures that TRON expects to be exported in the dynamically linked library. The `extern "C"` scope specifies that C-function name mangling should be used instead of C++ (needed if compiled by `g++`). The C++ name mangling is very different across various compilers (and their versions) hence is discouraged for portability purposes, although the internal implementation can be a mixture of C and C++ code. The function `adapter_new` is called by TRON to initialize the adapter. The function takes a pointer to `Reporter` structure (TRON driver interface, see Figure 11) and command line arguments. It should create a `TestAdapter` interface to the adapter (see Figure 10) and configure `Reporter` interface. Function `adapter_delete` is called by TRON to cleanup and release the resources associated with adapter, normally it contains at least a call to `TestAdapter` destructor. The library should

```
extern "C" {
    TestAdapter* adapter_new(Reporter* r, int argc, const char* args[]);
    void adapter_delete(TestAdapter* adapter);
}
```

Figure 9: Dynamically linked library (DLL) interface functions.

be compiled in such a way that the functions appear as dynamic symbols, i.e. use `-shared -fPIC -DPIC` options for GCC to compile and use `objdump -T` to inspect what symbols are exported.

Figure 10 shows the `TestAdapter` interface to the adapter. The `start` and `perform` function pointers should be assigned to point to the code that initiates testing (allocate necessary resources, establish connection, reset IUT, etc.) and perform an input action. The testing time starts counting when the function call from `start` returns. The `perform` function is responsible for delivering the input to IUT, it takes three parameters: channel identifier `chan`, the number of parameters `n` and an variable value array `data` of size `n`. The channel identifiers should be acquired from the `Reporter` interface during the `adapter_new` call and the parameter count should be consistent with the number of variables bound to the particular channel. The input action is time-stamped by before and after `perform` function call time-stamps. The easiest way to implement `TestAdapter` interface is to inherit it (or extend in Java terms), provide `start` and `perform` (non-member) function implementations (which probably access

```
struct TestAdapter {
    void (*start)(TestAdapter* adapter);
    void (*perform)(TestAdapter* adapter, int32_t chan, uint16_t n,
                                              const int32_t data[]);
    Reporter* const rep;
    TestAdapter(Reporter* r): rep(r) { start  = 0; perform = 0; }
};
```

Figure 10: TestAdapter: C-interface to adapter (`tron/adapter.h`).

`adapter`-implementation members) and set the `start` and `perform` function pointers to the function implementations. It is expected that `perform` executes fast without blocking, e.g. it should just put the input event into the queue (perhaps protected by POSIX thread mutex lock) and return, whereas another adapter thread should read from the queue and deliver the actual input. Note that `TestAdapter` constructor sets the `NULL` as default values for `start` and `perform` function pointers to ensure that the developer sets them to meaningful addresses.
Important: the `TestAdapter::perform` function implementation should not call `Reporter::report_now` function as the adapter may deadlock.

Figure 11 shows the `Reporter` interface to Tron driver. In the beginning of testing, the `adapter_new` should use it to configure the driver by specifying input and output channels, attaching variables, setting the model time unit and timeout values. Functions `getInputEncoding` and `getOutputEncoding` declare a channel as observable input and output respectively. They also return a non-negative integer value denoting the channel identifier to be used in `perform`, `report_now` and other function calls. Functions `addVarToInput` and `addVarToOutput` associate the variable names with given channels: the specified variable values will be attached to each event on the given channel as data parameters in `perform` and `report_now` function calls. All functions return non-negative integer value upon success and a negative value indicates an error code. Function `getErrorMessage` can be used to extract a character string explanation of the error code.

```
struct Reporter {
    void (*report_now)(Reporter*, int32_t chan, uint16_t n, const int32_t data[]);
    int32_t (*getInputEncoding)(Reporter*, const char* inputChanName);
    int32_t (*getOutputEncoding)(Reporter*, const char* outputChanName);
    int32_t (*addVarToInput)(Reporter*, int32_t chan, const char* variable);
    int32_t (*addVarToOutput)(Reporter*, int32_t chan, const char* variable);
    int32_t (*setTimeUnit)(Reporter*, const int64_t& microsecs_per_unit);
    int32_t (*setTimeout)(Reporter*, int32_t timeout_in_units);
    const char* (*getErrorMessage)(Reporter*, int32_t error_code);
};
```

Figure 11: Reporter: C-interface to Uppaal Tron driver (`tron/adapter.h`).

Figure 12 shows the interaction between Tron and adapter library. First Tron asks operating system to load the specified adapter DLL and lookup the adapter functions. Then TRON calls `adapter_new` which configures the testing interface by calling back the `Reporter` interface. When `adapter_new` returns, TRON partitions the model and calls `start` to start testing. The following

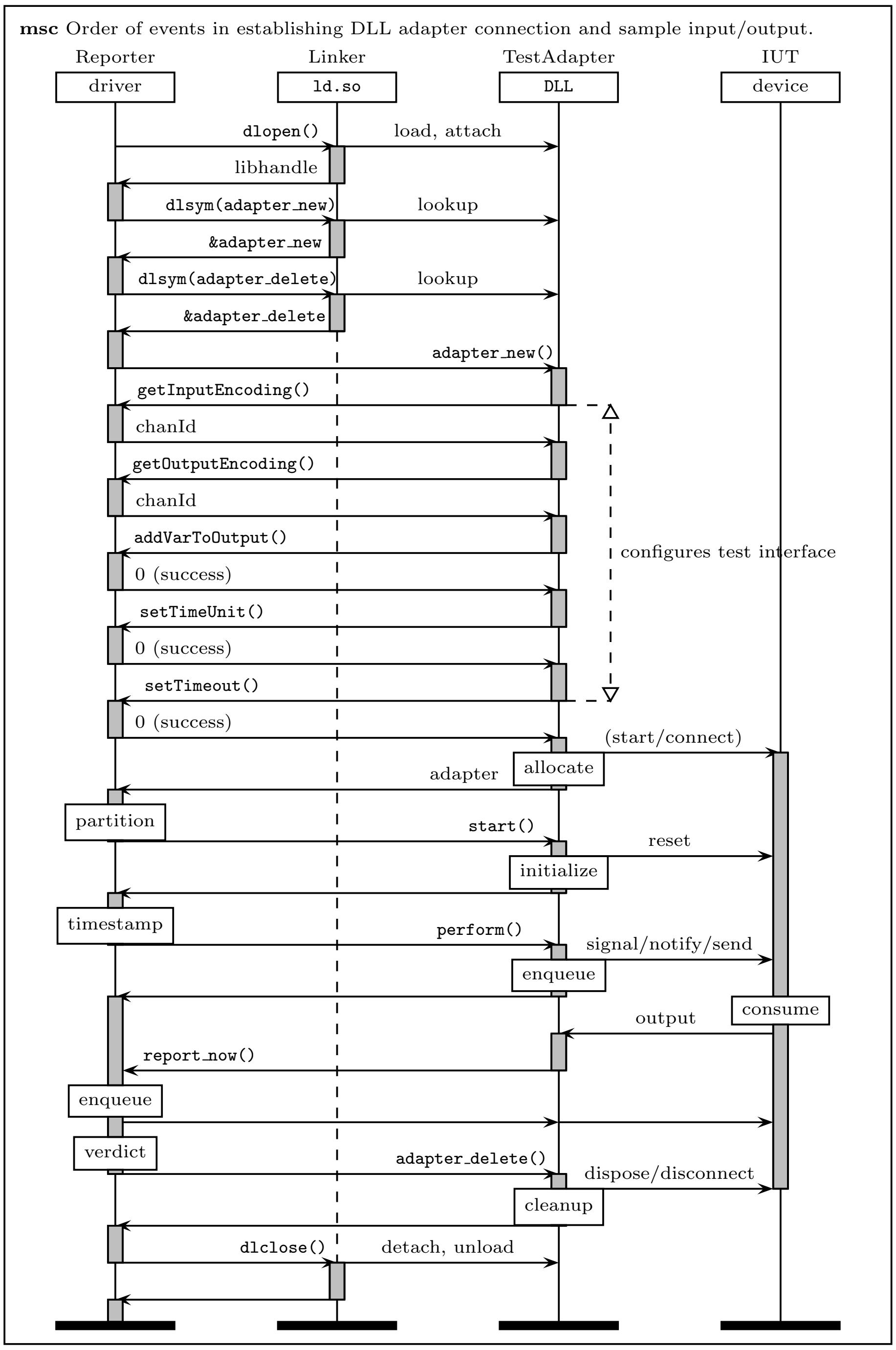


Figure 12: Sample event sequence in dynamic library adapter during testing.

actions are executed during the sample test run:

allocate: the adapter allocates resources and starts threads necessary to establishing physical connection to IUT.

partition: TRON checks whether model time unit and testing timeout parameters are set (exits with error message if they are not set) and attempts to partition the system model. The partitioning errors are reported to standard error stream, but testing is not stopped assuming that the developer knows what she is doing.

initialize: the adapter finishes any initializations left and resets the IUT into an initial state.

timestamp: TRON looks-up at its clock and records the moment of absolute test start, further time-stamps will be relative to this moment.

enqueue at TestAdapter: the adapter transfers (copies) necessary information about an input, schedules an immediate execution of the input event and returns immediately. Note that it may be dangerous to call IUT routine directly as it may result in producing an immediate output and may deadlock the adapter protocol, however it is fine for another IUT thread to produce output while adapter is enqueueing input.

consume: IUT receives and consumes the input.

enqueue at Reporter: the driver records the moment of the output event, copies the event into the queue and returns immediately.

verdict: TRON comes up with a verdict, records the test run statistics and prepares to terminate. Note that verdict is executed before cleanup in order to preserve the test results against potential faults in a cleanup code.

cleanup: the adapter terminates connection to IUT and releases resources it has allocated before. Note that adapter's structures (during allocation and I/O handling) should be allocated separately and the adapter may use its own memory allocator (independently of what TRON is using), hence it is ordered to cleanup its own memory separately. It is recommended that adapter memory is allocated statically (e.g. use static arrays for buffers) and dynamic allocations avoided as much as possible.

3.2 TRON Loaded as Dynamically Linked Library

Sometimes it is desired that the implementation or adapter would be the main executable and TRON is the library. We provide `libtron.so` [5] executable library to accomplish just that.

The interface is basically the same as dynamically linked library in Section 3.1 except that TRON is loaded from `libtron.so` and its `main` function with all usual command line parameters is called explicitly. The main executable needs to include header `tron/adapter.h` and to be linked with

[5] Linux only for now.

`-ltron` option. The main program has to call `set_main_adapter_new` and `set_main_adapter_delete` to register the adapter functions that are normally exported for DLL (see Section 3.1). The `main` function of TRON is then called by invoking `tron_main` function which should specify the adapter called "main", i.e. use `-I main` command line option. The TRON library `libtron.so` has to be available during compilation and runtime, which can be accomplished by copying it to system library directory (`/usr/lib/`) or setting the `LD_LIBRARY_PATH` environment variable accordingly (consult `ld` manual). The compiler and linker may request specific version of `libtron.so`, e.g. 1.5, which is usually called `libtron.so.1.5` (and `libtron.so` is just a symbolic link to it).

A complete setup with source files and GNU makefile script is provided in `button-main` example.

3.3 TCP/IP Socket Interface

Tron has a build-in adapter called `SocketAdapter` to communicate with remote IUTs (or yet another adapter framework) via TCP/IP sockets. The adapter requires arguments to configure the socket layer. It may either configured as client (initiator of connection to adapter/IUT) or a server (awaits connections from adapter/IUT). This adapter is easier to develop and use than DLL as it does not require platform specific knowledge and provides process isolation. The provided API and configuration procedure is similar to that of DLL interface described in Section 3.1 except it is network packet based.

`SocketAdapter` expects arguments, either a) port number to create server socket and listen for incoming connections or b) a hostname and a port number of the remote listening socket.

Once the connection is established the adapter consists of two threads: one listening (for outputs) and the other sending inputs, hence input-output communication can be completely asynchronous.

The listening thread responds to the packet-commands listed below. The commands can be put into one or across several network packets, but Tron is sending one packet per command (since 1.4 beta 3). In the beginning the `SocketAdapter` listens for the configuration commands which start with one-byte command identifier and are synchronous (i.e. Tron will immediately reply with a result). Once `requestStart` command is sent, Tron time-stamps the start of testing and adapter switches to asynchronous mode for test execution.

`getInputEncoding` registers the specified channel as input and returns the identifier for that channel.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Request:	1	N	chanName (N bytes)								
Reply:	chanId or error										

`getOutputEncoding` registers the specified channel as output and returns the identifier for that channel.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Request:	2	N	chanName (N bytes)								
Reply:	chanId or error										

`addVarToInput` binds specified variable to an input channel. Returns the result (success or error) of an operation.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Request:	3	chanId				N	varName (N bytes)				
Reply:	error code										

`addVarToOutput` binds specified variable to an output channel. Returns the result (success or error) of an operation.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Request:	4	chanId				N	varName (N bytes)				
Reply:	error code										

`setTimeUnit` sets the value of one model time unit in real world units. Returns the result (success or error) of an operation.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Request:	5	seconds				microseconds					
Reply:	error code										

`setTimeout` sets the timeout for testing value in model time units. Returns the result (success or error) of an operation.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Request:	6	timeout									
Reply:	error code										

`requestStart` finalizes adapter configuration, partitions the model, and starts asynchronous testing phase. Returns 0 telling that testing phase has been started, or terminates the connection and exits if configuration errors are found.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Request:	64										
Reply:	0										

`getErrorMessage` requests the description of an error code (issued during configuration). Returns a message string explaining the error code.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Request:	127	error code									
Reply:	N	message (N bytes)									

unrecognized command. If TRON fails to recognize a command ($X \in \{0\} \cup [7,63] \cup [65,126] \cup [128,255]$) during adapter configuration it will send back a string with explanation, close the connection and exit.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Request:	X										
Reply:	-1				N	message (N bytes)					

Asynchronous test execution commands are listed below.

`perform` TRON sends an input command to a remote adapter. In virtual time, the remote adapter should acknowledge the reception by sending a reply (make sure the remote socket is protected from simultaneous writes as acknowledgement may interfere with output reporting). If virtual time framework is not used, then no acknowledgement is needed.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Sends:	chanId				varN		varVal (N×4 bytes)				
Expects in virtual time:	acknowledgment										
Expects in real time:											

`report_now` The remote adapter sends an output command from IUT. In virtual time, TRON will acknowledge the reception, thus the sender thread should wait for it. If virtual time is not used, then there will be no acknowledgement sent. Make sure that socket write operation is protected from multiple thread access as several outputs may clash.

Bytes:	0	1	2	3	4	5	6	7	8	9	...
Send:	chanId				varN		varVal (N×4 bytes)				
Expect in virtual time:	acknowledgment										
Expect in real time:											

The following is a list of entities used in `SocketAdapter` protocol:

N is an unsigned byte meaning the number of bytes the next entity in the packet is occupying (like in n-string format).

chanName a character string meaning a channel name used in UPPAAL model. The terminating zero can be omitted (like in n-string format).

chanId is a signed 32-bit integer identifying a channel in the UPPAAL model. The identifier is greater than zero and bound by the total number of channels in the system. Values less or equal to zero are reserved for error codes (see error below in this list).

varName is a character string meaning a variable name used in UPPAAL model. The terminating zero can be omitted (like in n-string format).

seconds is a signed 32-bit integer meaning the number of seconds in one time unit (precision).

microseconds is a signed 32-bit integer meaning the number of microseconds which is added to the amount of seconds to get the full value of one time unit (precision).

timeout is a signed 32-bit integer meaning the number of time units before testing timeout (end of testing) is registered (and verdict test passed is issued).

error is a signed 32-bit integer meaning an error code when previous operation has failed. The error code is less or equal to zero, negative means error and the description can be retrieved by `getErrorMessage` command. Zero and positive values mean success and positive values mean channel identifier (chanId).

message is a character string describing an error state.

varN is an unsigned 16-bit integer meaning the number of variable values that follow right after it.

varVal is an array of N signed 32-bit integers meaning the variable values bound to a channel synchronization.

acknowledgement is 32-bit signed integer, used only in virtual time to acknowledge the reception of an input/output event by both (TRON and adapter) sides. The packet is marked with the 31^{st} (the most significant) bit set to 1. After the 31^{st} bit is cleared (set to 0) the resulting integer means the number of input/output packets received since last reception. The current implementation transfers only one input/output event per packet, hence the integer is typically set to one. Note that this does not conflict with channel identifiers as they are always positive and have 31^{st} bit set to 0.

All numbers are converted from native host to network (big-endian) byte order (see `htons(3)` and `htonl(3)`) before sending over network.

3.4 Sample Java Interface

The TRON distribution includes a smart lamp example which uses the `SocketAdapter` at TRON side and provides a reference implementation of `SocketAdapter` protocol in Java. The Java interface is made to be similar to C function interface discussed in Section 3.1 which implements and hides the `SocketAdapter` transport layer. The initialization process is slightly different, as the Java program is started independently from TRON process, also the error handling is done via more convenient Java exception mechanism, where error codes are automatically decoded. The TRON distribution also includes JavaDoc comments and generated HTML documentation of this Java interface.

Figure 13 shows the `Reporter` interface for Java programs. The base class `VirtualThread` denotes that it is also suitable for virtual time framework (see Section 3.6 for details). In order to establish a connection to TRON, one

```
public class Reporter extends VirtualThread {
  public Reporter(Adapter adapter, int port);
  public Reporter(Adapter adapter, String host, int port);
  public int addInput(String channel) throws TronException, IOException;
  public int addOutput(String channel) throws TronException, IOException;
  public void addVarToInput(int channel, String variable)
                          throws TronException, IOException;
  public void addVarToOutput(int channel, String variable)
                          throws TronException, IOException;
  public void setTimeUnit(long microsecs)
                          throws TronException, IOException;
  public void setTimeout(int timeout_in_units)
                          throws TronException, IOException;
  public String getErrorMessage(int error_code);
  public void report(int chanId);
  public void report(int chanId, int[] params);
  public boolean isConnected();
  public void disconnect();
  public void shutdown();
  public void run();
}
```

Figure 13: Reporter: Java interface to TRON driver.

must provide a reference to the `Adapter` interface implementation and call the `Reporter` constructor. The first constructor creates server socket on a specified port number and creates a waiting thread. The second constructor just starts a waiting thread. The connection is established by the waiting thread either by accepting another connection or connecting to a remote socket depending on the constructor used, and once the connection is established it will ask the `Adapter` object to configure the testing interface via the `Adapter.configure` method.

The configuration should consist of calls to adding input and output channels (`addInput` and `addOutput`), associating variables with channels (`addVarToInput`, `addVarToOutput`) and setting the timing information (`setTimeUnit`, `setTimeout`) as in Section 3.1. The methods may throw `IOException` upon usual socket connection problems or `TronException` (see Figure 15) if bad parameters are supplied.

The `Reporter` interface also provides two versions of `report` method to report about the produced output: the first one should be used if output does not have any variable values associated and the second one requires the list of variable values in the `params` array. The method `isConnected` returns `true` if the connection is established. The method `disconnect` disconnects the current tester with a possibility for another connection and `shutdown` disconnects and stops the waiting thread leaving no possibility for further connections. The method `run` is used by the waiting thread and normally should not be used (unless developer knows what she is doing).

The `Adapter` interface consists of two methods: `configure` for configuring test interface for new tester connection and `perform` for accepting the inputs from tester. The parameter `chanId` is the identifier of a channel received from `Reporter.addInput` calls and `params` is an array of attached variable values.

```
public interface Adapter {
    public void configure(Reporter reporter) throws TronException, IOException;
    public void perform(int chanId, int[] params);
}
```

Figure 14: Adapter: Java interface to adapter.

```
public class TronException extends IOException {
    public TronException(String message) { super(message); }
}
```

Figure 15: TronException thrown upon testing interface configuration error.

3.5 Interactive Text Interface

Tron has a build-in adapter called `TraceAdapter` for interacting via standard input and output streams. The adapter uses ANTLR [8] generated parser to recognize textual commands, which may seem suboptimal, but it is an ideal tool to experiment with an Uppaal model in virtual time framework, where test traces can be rerun and re-inspected for clues on what went wrong during real test execution.

`TraceAdapter` accepts two optional arguments: path to a file containing the *trace preamble* and trace interpretation mode. The *trace preamble* provides the test interface definition which configures TRON and prepares test driver for test execution. The file format should follow the grammar depicted in Figure 16, where The terminals `ChanID`, `VarID` and `INT` stand for channel name (identifier as in UPPAAL model), variable name (identifier as in UPPAAL model) and integer number accordingly. Figure 6a shows an example of trace preamble. The

```
preamble: inputs outputs precision timeout ;
inputs    : "input" ( siglist )? ";" ;
outputs   : "output" ( siglist )? ";" ;
precision : "precision" INT ";" ;
timeout   : "timeout" INT ";" ;

 siglist   : signature ("," signature)* ;
signature : ChanID "(" ( idlist )? ")" ;
 idlist    : VarID ("," VarID)* ;
```

Figure 16: EBNF grammar for file provided to `TraceAdapter` as argument.

interpretation mode can be either: `-t` for testing (default), `-m` for monitoring or `-e` for emulation. The testing mode declares input channels as inputs and output channels as outputs. The monitoring mode declares all channels as outputs (even the ones declared in input section) which in effect puts TRON into position where no inputs are generated and only the validity of outputs and delays is checked. The monitoring mode can be used to re-execute the trace as it was observed on a test driver level (see `-D` option in Section 4.1 and Section 4.2 to obtain such traces). The emulation mode declares all channels as inputs (even the ones declared in output section) which has an effect that TRON is in charge of generating all observable events on its own where user can control only the time delays (when run in virtual time). The emulation mode can be used to generate random tests without having built any implementation.

Figure 17a shows the grammar of language the `TraceAdapter` is expecting from standard input. The *trace* consists of a sequence of *commands*. Current `TraceAdapter` implementation supports three types of *commands*:

input asks the adapter to delay and wait until one of the input actions is received, all not mentioned inputs are going to be ignored.

output asks the adapter to deliver one output action while expecting to also receive specified input actions at the same time[6].

delay prepares to delay for a specified time moment while expecting the delay to be interrupted by specified inputs at specified times. The *timestamp* may give an interval of time, where the `TraceAdapter` chooses the exect time moment on a random basis. `TraceAdapter` terminates with an error message if unexpected (not mentioned, or at wrong time) input arrives. Instead of elaborate list of expected input actions one may want to specify symbol `*` which stands for “expect anything” (not mentioned in the grammar).

[6]FIXME: current implementation does not check the inputs.

```
trace :  (command)* ;
command : "input" expect ("," expect)* ";"
        | "output" action ("," action)* ";"
        | "delay" timestamp ("," expect)* ";"
        ;

action     : ChanID "(" (valuelist)? ")" ;
valuelist  : INT ("," INT)* ;

expect     : action (timestamp)? ;
timestamp: ("@")? "[" time "," time "]" ;
time       : FLOAT | INT ;
```

(a) EBNF grammar of trace.

```
delay [2.0,3.0];
output trigger();
delay 11.0, reply()[0.0,10.0];
delay [0.0,1.0];
output send(4);
input receive(16);
output one2many();
delay [11.0, 15.0];
output many();
input reply()[0.0,0.0];
input reply()[0.0,0.0];
delay 10.0;
```

(b) Trace from `tracer` example.

Figure 17: Grammar and a sample trace for `TraceAdapter` input stream.

The moments in time can be specified in various ways by using *timestamp* rule: optional symbol `@` specifies that timing should be calculated on absolute time basis, i.e. the proceeding numbers mean the time moments from the start of testing, otherwise the numbers are relative to the current time moment, then the interval of two *time* points follow, where the *time* can be expressed in integer number (interpreted as microseconds) or in floating point number (interpreted in model time units). Figure 17b shows a sample trace.

Exercise. Make your own model of a system with periodic behavior and compose a few traces to "test" some interactive I/O properties of your model, make one trace file per property. Use `repeater` script from `tracer` example to produce infinite traces from your trace fragments.

3.6 Virtual Time Framework

The purpose of the virtual time framework is to provide "lab" conditions for testing software where the value of a global reference clock is controlled and detached from physical time. Such framework allows to test time delays specified in software in ideal conditions where the time spent on computation and communication is treated as zero. If the computation and or communication time is known and needed to be taken into account, then such delays can be replaced by "timed-wait" calls and an abstraction of control software can be tested under ideal conditions.

The virtual time framework is implemented using one global virtual clock, whose value is incremented only when all threads (registered in the framework) request to delay and block until specified timeout expires. The clock value is incremented to the smallest time value needed to unblock at least one thread, and then the corresponding threads are unblocked to proceed. This simple idea is implemented using monitor programming paradigm within a subset of POSIX [4] thread functions (Portable Operating System Interface 1003.1b-1993 realtime extension).

Figure 18 shows the usage of monitor paradigm in producer-consumer problem implemented in C++ (Figure 18a) and Java 5 (Figure 18b) programming languages.

```
#include <pthread.h>
#include <deque>
class MyMonitor {
  pthread_mutex_t lock;
  pthread_cond_t cond;
  std::deque<int> buffer;
  MyMonitor():
    lock(PTHREAD_MUTEX_INITIALIZER),
    cond(PTHREAD_COND_INITIALIZER) {}
  void put(int value) { // produce
    pthread_mutex_lock(&lock);
    buffer.push_back(value);
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&lock);
  }
  int get() { // consume
    int value;
    pthread_mutex_lock(&lock);
    while (buffer.empty())
      pthread_cond_wait(&cond, &lock);
    value = buffer.front();
    buffer.pop_front();
    pthread_mutex_unlock(&lock);
    return value;
  }
}
```

(a) Sample monitor in C/C++.

```
import java.util.Vector;
class MyMonitor {
  Vector<Integer> buffer;
  MyMonitor() {
    buffer = new Vector<Integer>();
  }
  /* produce items with put(item) */
  synchronized void put(int value) {
    buffer.add(new Integer(value));
    notifyAll();
  }
  /* consume items with get() */
  synchronized int get()
          throws InterruptedException
  {
    while (buffer.isEmpty())
      wait();
    return buffer.remove(0).intValue();
  }
}
```

(b) Sample monitor in Java.

Figure 18: Sample monitor implementations for producer-consumer problem.

A few common thread-programming rules to avoid trouble:

- Unlocking order should be in reverse order of locking, i.e. lock acquisition and release should be nested like scopes to prevent circular dependencies and hence deadlocks.
- Condition signalling/broadcasting should be protected by an associated mutex lock, otherwise signals may be lost.
- A single mutex can be associated with many conditions, but each condition should be associated with only one mutex, i.e. the condition should be protected by the same mutex lock in all cases when it is used.

Exercise. Make a mutant of your IUT where one of the above rules does not hold and run TRON test against it. (Do not change the adapter code as it might kill TRON as well.)

The following sections explain how to adopt the implementation for virtual time framework.

3.6.1 Dynamic Library IUT

TRON binary itself exports a set of functions necessary to implement POSIX-like monitor. Figure 19 shows the list of POSIX functions to be replaced by TRON implementations in order to work with virtual clock, please lookup the POSIX programmer's manual (included in most Linux distributions) of these functions for detailed descriptions.

Figure 20 shows the list of symbols TRON is exporting. The symbols refer to corresponding POSIX function implementations and more. Almost all

```
int pthread_create(pthread_t*, pthread_attr_t*, void* (*start)(void*), void*);
int pthread_join(pthread_t, void**);
int pthread_mutex_init(pthread_mutex_t*, const pthread_mutexattr_t*);
int pthread_mutex_destroy(pthread_mutex_t*);
int pthread_mutex_lock(pthread_mutex_t*);
int pthread_mutex_unlock(pthread_mutex_t*);
int pthread_cond_init(pthread_cond_t*, const pthread_condattr_t*);
int pthread_cond_destroy(pthread_cond_t*);
int pthread_cond_wait(pthread_cond_t*, pthread_mutex_t*);
int pthread_cond_timedwait(pthread_cond_t*, pthread_mutex_t*, const struct timespec*);
int pthread_cond_signal(pthread_cond_t*);
int pthread_cond_broadcast(pthread_cond_t*);
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Figure 19: POSIX thread functions.

```
1  int (*tron_thread_create) (pthread_t*, const pthread_attr_t*, void* (*start)(void*), void*);
2  int (*tron_thread_join) (pthread_t, void**);
3  int (*tron_mutex_init) (pthread_mutex_t*, const pthread_mutexattr_t*);
4  int (*tron_mutex_destroy) (pthread_mutex_t*);
5  int (*tron_mutex_lock) (pthread_mutex_t*);
6  int (*tron_mutex_unlock) (pthread_mutex_t*);
7  int (*tron_cond_init )(pthread_cond_t*, const pthread_condattr_t*);
8  int (*tron_cond_destroy)(pthread_cond_t*);
9  int (*tron_cond_wait) (pthread_cond_t*, pthread_mutex_t*);
10 int (*tron_cond_timedwait) (pthread_cond_t*, pthread_mutex_t*, const struct timespec*);
11 void (*tron_cond_signal) (pthread_cond_t*);
12 void (*tron_cond_broadcast) (pthread_cond_t*);
13 void (*tron_gettime) (struct timespec*);
14
15 typedef enum TKMode_t { TKHostClock, TKLogClock, TKExtClock };
16 TKMode_t TKMode; // read-only variable for time keeping mode
17 int setHostClock();
18 int setLogicalClock(bool reg=true);
19 int setLogicalClockService(bool reg=true, int port=0x1979);
20 int setSocketClock(const char* host, int port=0x1979, bool reg=true);
```

Figure 20: Tron functions to replace a subset of POSIX.

function signatures are the same as their POSIX analogs, the only exceptions are condition signalling (functions always succeed) and getting value of clock (`gettimeofday` operates on `timeval` structure rather than `timespec` which is more convenient when working with timedwait). The symbols are of function-pointer type in order to be able to turn on or off the virtual time framework without recompiling. The value of variable `TKMode` can be used to determined what time-keeping mode is used (usually it is not necessary): `TKHostClock` means the host clock, i.e. the underlying OS POSIX layer is called directly, `TKLogClock` means the local logical (virtual) clock, `TKExtClock` means the remote logical clock. The functions at lines 16-18 can be used to set a particular time framework (also not necessary as it is done by `-Q` command line option). The local logical clock also creates a local TCP/IP server socket and listens for remote connections (see Section 3.6.2), so only one instance of local logical clock should be used, the other processes should use the remote clock accessed via TCP/IP sockets (e.g. Section 3.6.3). The parameter `reg` controls whether the calling thread should also be added to the pool of virtual threads, this is usually needed only for the main process thread as all other threads (created via `tron_thread_create`) are automatically added once the main thread sets-up the required framework.

The implementation of `tron_` functions are linked inside TRON binary file. The trick is that dynamic loader looks-up and resolve the `tron_` symbols automatically also for any dynamic library loaded as adapter. Currently this works very well on Linux (see the `button` example) but not on Windows (suggestions for possible solutions are welcome).

Exercise. Convert the code in Figure 18a to use virtual time framework.

3.6.2 Remote Virtual Clock Service

POSIX threads are good for synchronizing threads within the same process address space, however it does not help to communicate with remote IUTs. An alternative could be to use Remote Procedure Calls (RPCs) or some Common Object Request Broker Architecture (CORBA) library, however such solutions require special permissions or tend to be big libraries while virtual clock is simple and does not need complicated data passing. In this section we describe how to access the virtual clock in TRON process via TCP/IP sockets which is lightweight, mature and pervasive throughout operating systems today.

Virtual clock framework is turned on by `-Q` option (Section 4.1): TRON can either create its own clock server when `-Q` has a port number as argument or "`log`" (implies default port number 6521) or use external virtual clock with a machine address and a port number (e.g. connect to another instance of TRON).

Virtual clock is always associated with socket server and threads are associated with client sockets. The protocol is designed so that each thread is identified by a separate socket connection: one duplex connection per thread. All thread operations are carried out in the context of that connection. Moreover, all socket communications are synchronous for client thread, meaning that it is trivial to use and there is no need for complicated locking mechanisms to protect socket connection from multi-threading nor creating special data structures. It is important that client threads do not share their connections with other threads as such sharing is meaningless and asks for trouble.

Virtual clock protocol consists of a set of commands corresponding to POSIX layer. The commands are carried out synchronously: client sends a virtual clock command with its arguments and waits for a response containing the result of operations. Server may respond with a delay if the command was timed-wait related, thus effectively putting the client thread into blocked state until the required (virtual) time delay elapses.

The protocol starts with client thread establishing connection to a clock server and sending its name (a human friendly identifier, useful for debugging) in ASCII N-string format (first byte denotes the length of a string, then up to 255 bytes of the string itself). The new connections automatically register a new thread in virtual time framework. After the name is sent (thread registered), the client thread may start using virtual clock by sending commands.

The following is a list of commands used in virtual time protocol:

Mutex initialize. Initializes new mutex variable.

Bytes:	0	1	2	3	4
Request:	3				
Response:	mutex ID				

Mutex destroy. Deletes mutex with specified ID. Response is empty, i.e. there is no result to wait for.

Bytes:	0	1	2	3	4
Request:	4	mutex ID			
Response:					

Mutex lock. Locks a mutex with the specified ID. Response contains Tron code from Table 2.

Bytes:	0	1	2	3	4
Request:	5	mutex ID			
Response:	code				

Mutex unlock. Unlocks a mutex with the specified ID. Response contains Tron code from Table 2.

Bytes:	0	1	2	3	4
Request:	6	mutex ID			
Response:	code				

Condition initialize. Initializes new condition variable.

Bytes:	0	1	2	3	4
Request:	7				
Response:	condition ID				

Condition destroy. Deletes a condition with the specified ID. Response is empty, i.e. there is no result to wait for.

Bytes:	0	1	2	3	4
Request:	8	condition ID			
Response:					

Conditional wait. Release the specified mutex, wait until the specified condition is triggered, re-acquire the mutex and return an operation code. Response contains Tron code from Table 2.

Bytes:	0	1	2	3	4	5	6	7	8
Request:	9	condition ID				mutex ID			
Response:	code								

Conditional timed wait. Release the specified mutex, wait until the specified condition is triggered or time has elapsed, re-acquire the mutex and return an operation code. Time is specified as *absolute* signed 32-bit integer values from *beginning of era* (see **Get time** command below). Response contains Tron code from Table 2.

Bytes:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Request:	11	condition ID				mutex ID				seconds				microseconds			
Response:	code																

Conditional delay. Release the specified mutex, wait until the specified condition is triggered or time has elapsed, re-acquire the mutex and return an operation code. Time is specified as *relative* signed 32-bit integer values from *current time* (see **Get time** command below). Response contains Tron code from Table 2. The command is provided as a shorthand for a common combination of **Get time** and **Conditional timed wait**.

Bytes:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Request:	11	condition ID				mutex ID				seconds				microseconds			
Response:	code																

Condition signal. Notifies one of the threads waiting on the specified condition. There is no response to wait for.

Bytes:	0	1	2	3	4
Request:	12	condition ID			
Response:					

Condition broadcast. Notifies all of the threads waiting on the specified condition. There is no response to wait for.

Bytes:	0	1	2	3	4
Request:	13	condition ID			
Response:					

Get time. Returns the absolute time-stamp of current time since era in two 32-bit integer numbers. Era, or the value of 0 in virtual time denotes the moment the virtual clock was created.

Bytes:	0	1	2	3	4	5	6	7	8
Request:	14								
Response:	seconds				microseconds				

Thread quit. Removes the registration of the thread and releases the associated resources so that other threads may continue using the virtual clock without this one. The deactivated threads should activate before termination (see **Activate thread**). There is no response to wait for.

Bytes:	0	1	2	3	4
Request:	127				
Response:					

Thread deactivate. Temporarily (until activation) removes the current thread from virtual time accounting. This is normally used *only* by special adapter threads (e.g. `SocketAdapter`) which wait for incoming actions from elsewhere (e.g. socket connection) rather than for regular condition variable notifications. The deactivated threads do not participate in time accounting but they are still important in notifying other threads about incoming actions. All other threads should not use deactivation mechanism at all.

Bytes:	0	1	2	3	4
Request:	1				
Response:	code				

Thread activate. Activates the deactivated thread (see **Thread deactivate**). Should be used *only* by special adapter threads (like one in `SocketAdapter`) just before termination.

Bytes:	0	1	2	3	4
Request:	2				
Response:	code				

Table 2 describes possible 32-bit number codes returned by TRON specific to virtual time framework via TCP/IP. The names are taken from POSIX C identifiers whose actual values may be different on various operating systems, thus the native error codes are translated to unique values in this table.

All the integers are converted to network byte order (see `htonl(3)` C function manual).

Some languages (like C and Java) provide a lot of options for configuring socket connections, hence consider disabling Nagle algorithm to send data as soon as possible and always do an explicit flush operation to make sure that the

Table 2: Tron error codes for virtual time via TCP/IP sockets.

Name	Code	Description
OK	0	No error, operation succeeded or condition has been triggered.
ERROR	64	Unexpected error: uncommon failure that is not handled by this error code translation.
ETIMEDOUT	65	Specified time has elapsed.
EINTR	66	Interrupted system call.
EBUSY	67	Device or resource is busy.
EINVAL	68	Invalid argument: invalid values or different mutexes supplied for concurrent operations on the same condition variable.

command and its arguments are dispatched. Other languages (like Python) rely on constructing TCP packets explicitly. Tron implements data buffering and treats the incoming flow of commands as a stream rather than packets, thus it is able to deal with both types of network APIs.

3.6.3 Virtual Clock for Java

Tron distribution contains sample Java implementation of virtual clock protocol via TCP/IP sockets that can be enabled in combination with `SocketAdapter` implementation in Java.

Virtual time framework in Java uses `VirtualThread` which extends `Thread` class and takes care of establishing connection to virtual clock. Thread synchronization is implemented through `VirtualLock` and `VirtualCondition` classes which implement interfaces from `java.util.concurrent .locks` package (available in Sun JDK since Java 5). The synchronization methods identify the calling `VirtualThread` objects and use their methods to carry out virtual time commands, thus in effect these methods use the context (socket connection) of particular thread to carry out operations on virtual clock without sharing or mixing with other threads. Eventually all synchronizations are resolved inside virtual clock server process.

Unfortunately the `synchronized` keyword is not supported directly and has to be changed to equivalent code using interfaces in `java.util.concurrent .locks` package.

The following is a list of actions needed to adopt virtual time framework for any Java application:

- All Java threads should extend `VirtualThread` class instead of `java.lang .Thread`. Note that this isolates the application from events in (graphical) user interface.
- Monitor methods should be modified as follows:
  - Synchronized methods and sections should be replaced by blocks surrounded by `VirtualLock.lock` and `VirtualLock.unlock()`.
  - `java.lang.Object.wait()` should be replaced with `VirtualCondition .await()` surrounded with appropriate `VirtualLock` object `lock()` and `unlock()` methods.

  - `java.lang.Object.notify()` and `java.lang.Object.notifyAll()` replace with `VirtualCondition.signal` and `VirtualCondition .signalAll()` respectively.

- Before any thread creation, set the remote virtual clock via `VirtualThread .setRemoteClock(String, int)` method call (once is enough).

Exercise. Convert the code in Figure 18b to use virtual time framework.

4 Testing

This section describes the features of test execution process of TRON. We start by describing the command line options, proceed with how to read and interpret test logs and explain the test verdict and diagnostics information.

4.1 Command Line Options

The following is a list command line options that developer can use to control the behavior of TRON. Each item starts with the key controlling the feature, followed by the description of feature. Some options affect the UPPAAL engine directly (marked with a star *) while others are completely TRON specific.

`-A`* Use convex-hull approximation.

`-B path` provide a file path to store benchmark log (default /dev/null), see Section 4.2.

`-D path` specify a file path to store driver log (default /dev/null), see Section 4.2.

`-F future` specifies how far into the future (in model time units) TRON should pre-compute the internal transition closure of a state-set estimate in order to make reasonable test choices. The setting limits the delay in symbolic-future operations in order to prevent TRON from exploring too far of internal and non-interactive (without observable input/output events) behavior. Default is 0, which means that TRON will take immediately enabled transitions and will not take any internal time-guarded transitions (without choosing to delay and satisfy their guards first). Larger values are recommended to reach more choices, and smaller values are preferred to reduce the performance penalty required for future pre-computations. For periodic systems good heuristic candidates are: the duration of the longest period or least common multiple of all periods. The feature can be disabled by setting -1: then internal transition closure computation will be turned off and not a single internal transitions will be considered when computing available input choices; this might be reasonable only if there are very few internal transition edges or the input/output events are very far apart in time (e.g. further than `-P` setting) and hence disabling is not recommended in general.

`-H n`* sets the hash table size for bit state hashing to 2^n (default 27). The setting influences the three passed-waiting lists (state-sets) in TRON. The default value come from reachability algorithm where the hash-table has to store entire system state space. During testing however, the state-sets are typically much smaller and n can be safely around 10 (1024 entries) to save some memory.

`-I name` specifies the implementation, or rather the location of the adapter to implementation where name is a file path to a dynamically linked library with adapter to an implementation, or one of the following built-in adapters:

`TraceAdapter` standard input/output stream adapter, see Section 3.5;

`SocketAdapter` remote TCP/IP socket adapter, see Section 3.3.

`-P delay` specifies the delay choice strategy (see also Section 4.4). The `delay` can be one of the following:

`eager` : delay as little as possible before firing a chosen action-transition. The choice is typically bound by the guards on edges (and invariants on the target location vector), TRON will choose the minimum or 0 if no guard is on the chosen edge.

`lazy` : delay as much as possible before firing a chosen action-transition. The choice is typically bound by invariants on current (and target) location vector, TRON will choose the maximum allowed or infinity (actually until the testing timeout) if no invariant is specified.

`random` : delay randomly within the bounds specified by the environment model (default). The choice is typically bound by the guards on a chosen edge and invariants on current (and target) environment location vector, hence the choice is randomly resolved to fit into this interval.

`short,long` : try random delay bounded by one of positive integer numbers: (`short` and `long`). The numbers specify the longest delay choice allowed in model time units, the interpretation "short" and "long" is arbitrary and not enforced, but rather a hint that periodic systems often have two or more periods of very different granularity. The concrete delay choice is still random and based on the specification (bounds will be ignored if specification require longer delays) but choices are guaranteed to be shorter or equal to max(`short`, `long`). This is useful to limit delays if there are states without invariants and developer wants more interactive (with more observable actions) test runs.

Notice that the delay strategy is orthogonal to `-P` option: `-F` controls how many action transitions are available (reachable) to choose from, while `-P` chooses the delay based on the information on chosen action transition.

`-Q log` sets logical (simulated or virtual) time clock instead of the default real host's clock (and does *not* start clock service).

`-Q port -Q host:port` Parameter `port` specifies that virtual clock service should be started on port number `port` or all clock requests forwarded to remote virtual clock service on machine called `host` and port number `port`. See Section 3.6 for details about TRON's virtual clock services.

`-S filename` Append the verdict, I/O and duration to file (default /dev/null), see Section 4.2.

`-U`* Unpack reduced constraint systems before relation test.

`-V` prints version information and exits.

`-X integer` initializes random number generator by a given integer value (default value is read from the host's system clock).

`-h` prints a short version of this option list description and exits.

`-i <dot|gui>` prints a signal flow diagram of the system and exits. There are two output formats available:

- `dot` : dot [3] graph, expects formated standard input (see Section 2.2):
  `"input"` *(channel)∗* `"output"` *(channel)∗*
- `gui` : non-partitioned flow information for TRON GUI;

`-o filename` Redirect output to file instead of stdout, see also `-v` and Section 4.2.

`-s <0|1|2>`* selects the exploration order of reachability algorithm. This should not have a significant impact on TRON performance, unless `-F` value is large and there are many internal transitions in the model. There are the following options:

- **0** : Breadth first (default)
- **1** : Depth first
- **2** : Random depth first

`-u inpDelay,inpRes,outDelay,outRes`

`-u inpRes,outRes` Observation uncertainty intervals in microseconds:

- `inpDelay` : the least delay that takes to deliver input,
- `inpRes` : possible additional delay for delivering input,
- `outDelay` : the least delay that takes to observe output,
- `outRes` : possible additional delay for observing output.

`-l latency` Specifies the maximum input scheduling latency in microseconds when offering the input. The value will be subtracted from the upper bound of the input timing which should prevent missing the input deadlines (verdicts like "input executed too late" and driver warnings like "DRIVER: 1193663117.714029s has passed, now it's 1193663117.714033s"). This option is similar to input observation uncertainty except that it does not affect the time-stamping after the input has been executed.

`-v <0+1+2+4+8+16>` sets verbosity of a test log printed to standard output stream (or file specified by `-o` option). The verbosity specifies what information should be included in the test log, see Section 4.2 for log description. The values of interest should be added to produce final verbosity number:

`= 0` : only verdict, disable engine event output (default),

`& 1` : progress indicator for interactive experiments,

`& 2` : test events applied in the UPPAAL engine,

`& 4` : available input and delay choices for emulation,

`& 8` : backup state set and prepare for final diagnostics,

`&16` : dumps current state set on each state set update.

If partitioning option `-i` is used instead of test run then partitioning messages can be controlled by the following verbosity values:

`0` : none,

`1` : errors,

`2` : errors and warnings (default),

`3` : errors, warnings and diagnostics.

`-w integer` specify additional number of model time units in attempt to test (violate) invariants. Useful under assumption that invariants are not used in the model of environment. This option is obsolete starting from version 1.4b1, where IUT invariants are removed from environment emulation (hence invariants tested under given environment) if system model partitioning is properly done (no partitioning errors are detected).

`-q` be quiet and do not display the copyright message.

UPPAAL engine also reacts to the following OS environment variables:

`UPPAAL_DISABLE_SWEEPLINE` : disable sweepline method,

`UPPAAL_DISABLE_OPTIMISER` : disable peephole optimiser,

`UPPAAL_OLD_SYNTAX` : use version 3.4 syntax for parsing old system models.

The value of these environment variables do not matter, defining them is enough to activate the features in question.

4.2 Logging

There are four ways to log test runs:

Engine log contains information about operations performed in UPPAAL engine. Messages follow the TRON online test algorithm. The engine events are sent to standard output by default, and can be redirected to a file via `-o` option. The verbosity of messages can be adjusted by `-v` option. The purpose is to display the current status of an online test run.

Driver log contains test interface description and time-stamped information about input and output events. The log file is specified by `-D` option and follows the `TraceAdapter` format (see Figures 16 and 17a). The purpose is to log input and output events precisely and to enable the trace replay with `TraceAdapter` in monitoring mode, potentially with different options.

Statistics log contains one line summary per one test run. Log file is specified by `-S` option. The purpose is to record many test runs in one file to provide statistical measures on how many inputs and outputs have been performed, how many test runs passed and failed. The statistics log contain the following columns:

1. The initial random seed of a test run. By default it is UNIX timestamp in seconds since the Epoch, see `-X` option in Section 4.1.
2. The test verdict of a test run in one word.
3. The number of inputs sent to an IUT.
4. The number of outputs received from an IUT.
5. The duration of a test run in model time units.

Here is an example of a statistics log:

```
1160727325 PASSED 13195 23753 100000
1163934755 FAILED 2 13 38
1163934756 INCONC 2 13 18
```

Benchmark log contains a one line timing measurement per one UPPAAL engine operation (**after delay** or **after action** updates) for benchmark purposes. The log file is specified by `-B` option. The purpose is to help tuning the UPPAAL engine for testing purposes. The file consists of four columns:

1. Zero or one: "0" stands for **after delay** and "1" stands for **after action** operations.
2. The state set size before the operation.
3. The state set size after the operation.
4. The high resolution (OS specific) time estimate of operation duration in nano-seconds.

4.3 Time Stamping

One of the key activities in test run evaluation is time-stamping the real I/O events and mapping those real time stamps into model time and back. Online testing has an additional challenge for testing tool: how to incorporate time delays spent by the testing tool itself and provide objective verdict if test fails (e.g. the testing tool may delay too long due to expensive computations and thus fail to fulfill the assumptions of environment model). TRON offers over-approximating method to match real time values into model time. In order to explain the idea behind this method we go through input offering scenarios incrementally: in virtual time framework, in naive real time and real time with observation uncertainties. At the end of this section we explain the details of mapping real time instances into model time instances and back together with observation uncertainties.

4.3.1 Virtual Time

Consider the following input offering scenario shown in message sequence chart (MSC) in Figure 21:

1. TRON asks what time is now and saves the value into variable t.
2. TRON converts the real time interval $[t, t+F]$ to model time interval $[L, U]$, where F is the future horizon constant from `-F` option.
3. TRON asks UPPAAL to update state set with delay and τ-transitions for all delays between L and U model time stamps. The result is saved into variable S.
4. TRON asks UPPAAL about what input and output events are available from a given state set S. The set of inputs is saved into variable *inps*.
5. TRON chooses some input action i randomly from the set of input actions. The input action is enabled at model time interval $[L_i, U_i]$.
6. TRON computes the real time interval $[l_i, u_i]$ corresponding to the model time interval $[L_i, U_i]$.
7. TRON chooses a specific target time instance t_{tgt} from real time interval $[l_i, u_i]$. By default, TRON chooses a random instance, or applies the delay choice strategy specified by `-P` option otherwise.
8. TRON asks driver to delay until the t_{tgt} time instance. Notice that so far there were no delay requests since the first `getTimeNow` call, hence there was virtually no delay (zero virtual time) until this step and the only delay in this scenario happens in this step.
9. After delay, TRON observes that there were no outputs and immediately asks driver to offer an input i.
10. Driver does not make any delays and passes the input i to adapter and stamps this input as executed at t_e real time instance. Note that t_e is equal to t_{tgt} as there was no virtual time delay since t_{tgt} instance was reached.
11. TRON maps the real time stamp t_e of the input action into model interval $[L_e, U_e]$ (this is potentially much narrower interval than $[L_i, U_i]$).
12. TRON asks UPPAAL to update (affectively filter and constrain) the state set to describe system states within model time interval $[L, U]$.
13. TRON asks UPPAAL to compute a new state set after action i.

Output time-stamping is much simpler: driver can be interrupted at any time by incoming output and thus time-stamp immediately. The output event with its time-stamp is discovered by TRON during the "wait" requests, the real time-stamp is converted to model time-stamp and applied to state set in the same way as input events.

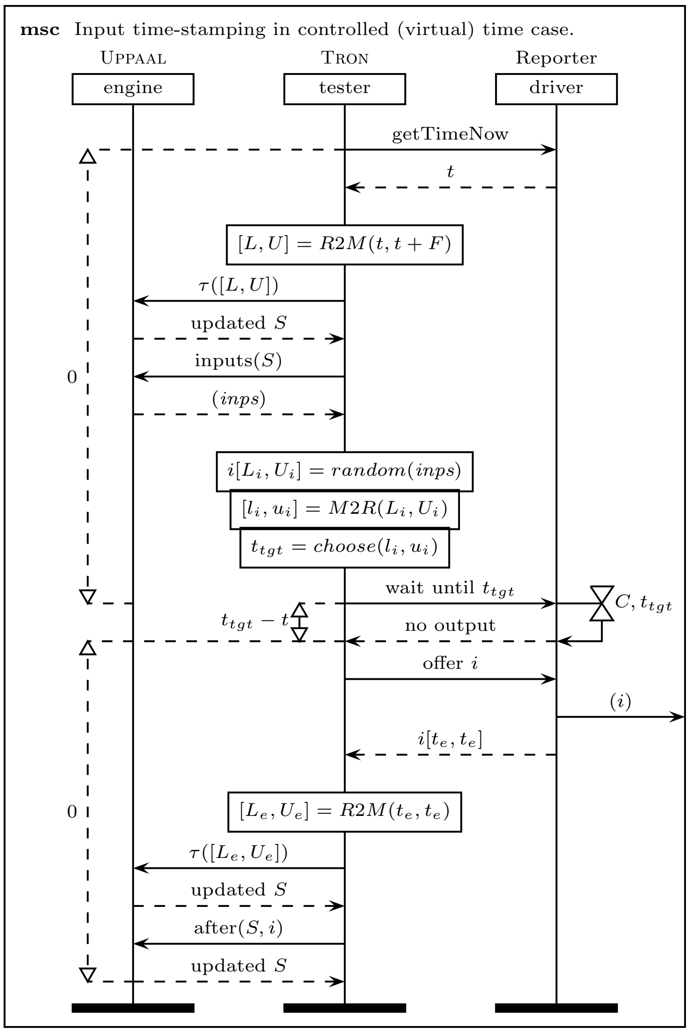


Figure 21: Scenario for offering an input to IUT and relevant timestamps in virtual time case.

4.3.2 Naive Real Time

From Figure 21 it is evident that in virtual time framework the time spent for computing, choosing and delivering the input is being ignored, and only explicit delays are counted. This assumption does not hold in real time and thus algorithm has to be adjusted to accommodate such delays. Figure 22 shows input offering scenario adjusted for real time, which differs from virtual time in the following ways:

1. The performance of input action calculation is unpredictable as it depends on the complexity of a system model and on particular state set, hence this delay is reflected in choosing the timing for the input: the interval is constrained from below by an extra time-stamp t_c. This reduces the driver warnings that the t_{tgt} instance of time is already in the past at the

time of “wait until” request. We still rely that the window for input is big enough to incorporate the chosen input: $t_c < u_i$, and hence any driver warning about t_{tgt} being in the past is a sign that TRON does not keep up with the requirements (boundary U_i) from the environment model. If t_c happen to be after u_i already before offering this input, then the input is discarded and another input is chosen instead (the whole input computation is restarted).

2. The time-stamping of the input execution is performed by two time stamps: between t_{try} and t_{done}, i.e. just before sending input and just after the send. The acquired model time interval $[L_e, U_e]$ denotes that the input happened somewhere in between, hence all possibilities has to be incorporated into the state set.

4.3.3 Internal Latency

So far, we still rely on the fact that TRON is woken up at precisely t_{tgt} moment and further input deliver happen instantaneously. This is not always true and cannot be predicted in all operating systems due to latency (jitter) in process scheduling and communication, however it is still important to be able to offer the input without violating u_i boundary. In this section we show how TRON adjusts input offering with a latency parameter `-l` $\mathcal{L}$ that specifies the worst latency duration. The latency is incorporated into $M2R$ function mapping which subtracts this amount of real-time from original u_i value, thus discarding the inputs which are too late with respect to upper boundary and local latency taken into account.

4.3.4 External Latency

Often test adapter introduces significant delays (communication latency) and I/O buffering. The only fair way to test such systems is to model test adapter as part of IUT. One way of adapter modeling is to provide explicit model in system specification (e.g. add timed automata processes for adapter). Typical adapter receives a signal, puts it into buffer, delays the signal (“signal is traveling”) and retransmits the signal for destination process. In this section we show how to acquire I/O timing characteristics of such adapter.

Figure 23 shows how IUT and tester use digital clocks to timestamp I/O events. For simplicity we assume perfect digital clock, which updates the time value with a period of it’s resolution, and time (value) is synchronized globally. We assume that adapter causes a delay D_1 for output and D_2 for input. We also assume that timestamping code runs instantly without delay, otherwise this deterministic delay can be added to adapter delay. At IUT side I/O happens at t_1 and t_{10} instances, however due to its digital clock time sampling IUT may think it happens at t_2 and t_{11}. Similarly at tester side I/O happens at t_3 and t_7, while tester timestamps these events at t_4 and $[t_6, t_9]$. Then observe the following inequalities:

$$\left\{\begin{array}{rcl} t_3 - D_1 = & t_1 & = t_3 - D_1 \\ t_2 - s_{iut} \leq & t_1 & < t_2 + R_1 \\ t_4 \leq & t_3 & < t_4 + R_2 \end{array}\right. \Rightarrow \left\{\begin{array}{rcl} t_4 - D_1 & \leq t_1 < & t_4 - (D_1 - R_2) \\ t_4 - (D_1 + R_1) & < t_2 < & t_4 - (D_1 - R_2) \end{array}\right. \tag{2}$$

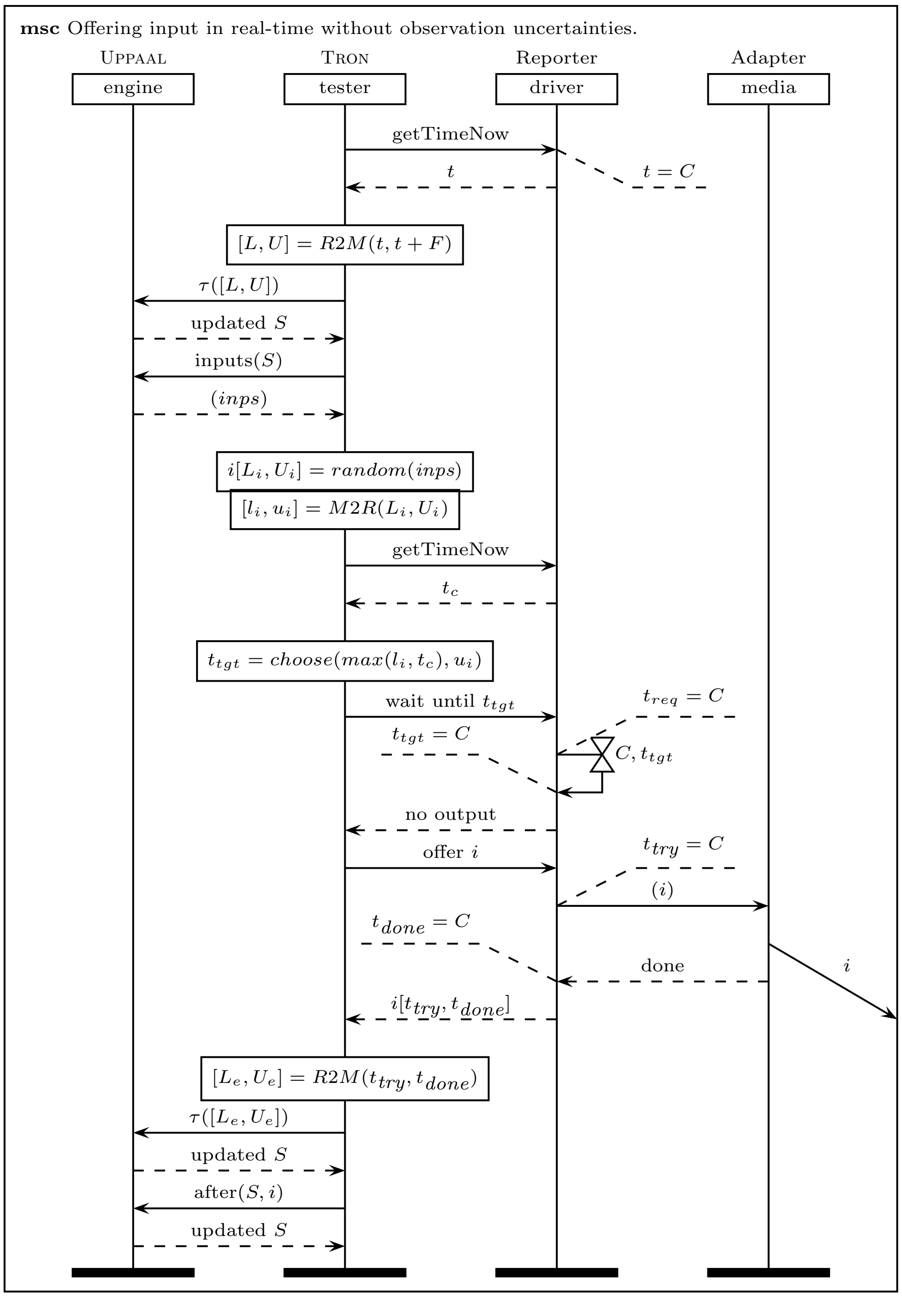


Figure 22: Scenario for offering an input to IUT and relevant timestamps in real time case without observation uncertainties.

$$\left\{\begin{array}{c} t_7 + D_2 = t_{10} = t_7 + D_2 \\ t_{10} - R_1 < t_{11} \leq t_{10} \\ t_6 \leq t_5 \leq t_7 \leq t_8 < t_9 + R_2 \end{array}\right. \Rightarrow \left\{\begin{array}{r} t_6 + D_2 \leq t_{10} < t_9 + D_2 + R_2 \\ t_6 + D_2 - R_1 < t_{11} < t_9 + D_2 + R_2 \end{array}\right. \tag{3}$$

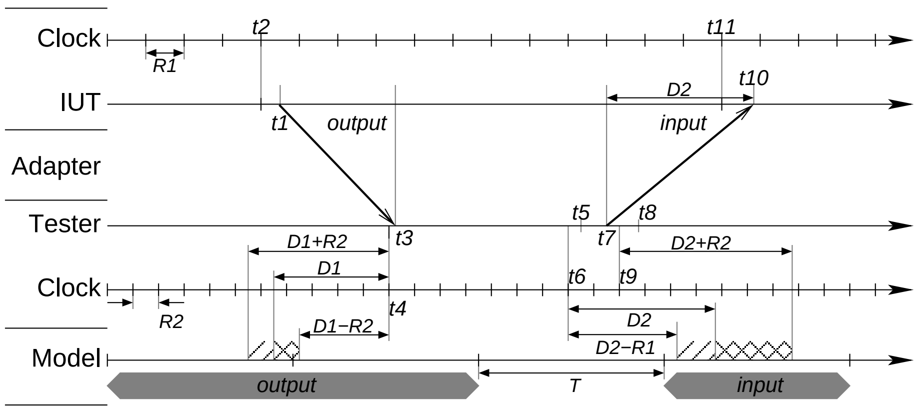


Figure 23: I/O delays in the adapter: IUT sends output at t_1 while its clock with resolution R_1 is showing t_2, the output is delayed by adapter by duration D_1 and sensed by tester at t_3 while testers clock with resolution R_2 is showing t_4; before sending input tester looks up its clock at t_5, observes value t_6, sends input at t_7, looks up the clock again at t_8 and observes value t_9, then input arrives at IUT at t_{10} while IUT's clock is showing t_{11}; the real time values are then mapped onto model time scale with resolution of T (real time value of one model time unit).

Therefore tester may conclude that at IUT side output happens at $\big(t_4 - (D_1 + R_1), t_4 - (D_1 - R_2)\big)$ and input happens at $\big(t_6 + D_2 - R_1, t_9 + D_2 + R_2\big)$. Therefore adapter has a minimum $\delta^{inp}_{min} = D_2 - R_1$ and a maximum $\delta^{inp}_{max} = D_2 + R_2$ delays for input, and a minimum delay $\delta^{out}_{min} = D_1 - R_2$ and a maximum $\delta^{out}_{max} = D_1 + R_1$ for delays output. These delays are marked in Figure 23.

In the following we show how to incorporate real world imperfections:

- If clocks are not perfect and have some kind of jitter (latency distribution), then the clock resolution values R_1 and R_2 can be described by the largest possible time steps.
- If adapter has a non deterministic delay then the values of D_1 and D_2 can be described by shortest and longest adapter delays.

Therefore, if R_1, R_2, D_1, D_2 are distributions rather than constant values, then:

$$\delta^{inp}_{min} = \min(D_2) - \max(R_1) \tag{4}$$
$$\delta^{inp}_{max} = \max(D_2) + \max(R_2) \tag{5}$$
$$\delta^{out}_{min} = \min(D_1) - \max(R_2) \tag{6}$$
$$\delta^{out}_{max} = \max(D_1) + \max(R_1) \tag{7}$$

These external latency boundaries can be built into the IUT requirements model or provided to Tron by `-o` $\delta^{inp}_{min}, \delta^{inp}_{max} - \delta^{inp}_{min}, \delta^{out}_{min}, \delta^{out}_{max} - \delta^{out}_{min}$ option. Further details and assumptions for the latter option are in the following sections.

4.3.5 Observation Uncertainties

A straightforward adapter modelling way is to provide one process per one signal and have as many processes as there can be signals at one time, then reuse these processes to handle infinitely many signals. Such model is quite generic (fits many systems) but contains high degree of non-determinism (varying signal speed) and parallelism (even if signal ordering is deterministic) which lead to large state sets just to be able to handle many simultaneous I/O events. Many events at the same time is more of an exception than a rule and thus such blind modeling is may have poor average performance and greatly obfuscates test diagnostics.

Tron provides an alternative way of modeling adapter latencies via observation uncertainties: Tron does not know when the input signal reaches IUT, only the moment of input dispatch is timestamped locally; the same applies to outputs, Tron does not know when IUT has sent an output signal, only the arrival of output signal is timestamped. Knowing basic communication jitter characteristics allows Tron to compute a precise estimate of when I/O actually happened. We assume that communication of input signal takes at least δ^{inp}_{min} and at most δ^{inp}_{max} of real time and output signal takes at least δ^{out}_{min} and at most δ^{out}_{out} of real time. Then the local I/O timestamps can be adjusted by these parameters to calculate the remote timestamps and get the estimate when I/O has been sent/received from IUT perspective, thus affectively abstracting away the whole adapter layer and its complexity. Figure 24 shows how I/O timing uncertainties are incorporated into input offering scenario. This still has an important assumption and price to pay:

- The adapter communication delay has to fit onto environment and IUT model synchronization time:
  - IUT model is assumed to be input enabled, thus there are no additional assumptions for IUT requirements model.
  - Environment may have constraints for inputs: lower bound l_i is not directly affected as input estimate can only be delayed, but upper bound u_i can be violated, thus we assume that this boundary is able to consume adapter latency: $t_{done} + \delta^{inp}_{max} < u_i$ – this can be checked during test run and environment model adjusted. Then, the latest moment for input scheduling is $u_i - \delta^{inp}_{max} - \mathcal{L}$ and obviously it cannot be earlier than l_i, hence we assume that environment model satisfies $u_i - \delta^{inp}_{max} - \mathcal{L} < l_i$ for all inputs – this too can be checked during test run and the environment model adjusted to fit this assumption.
  - IUT model may have constraints over outputs and thus not entire interval of output timestamps may be applicable and thus some parts of interval may be discarded. Note that we compute an interval of all possible output timestamps, including the actual output timing, thus at least one point in that interval is required for IUT to pass this test step and it is safe to assume that others did not actually happen. If output did happen at the time the IUT constraints did not allow but it was included in the interval timestamp, then IUT actually failed this test step, but Tron have no possibility of detecting such possibility, thus further testing is based on some false assumptions

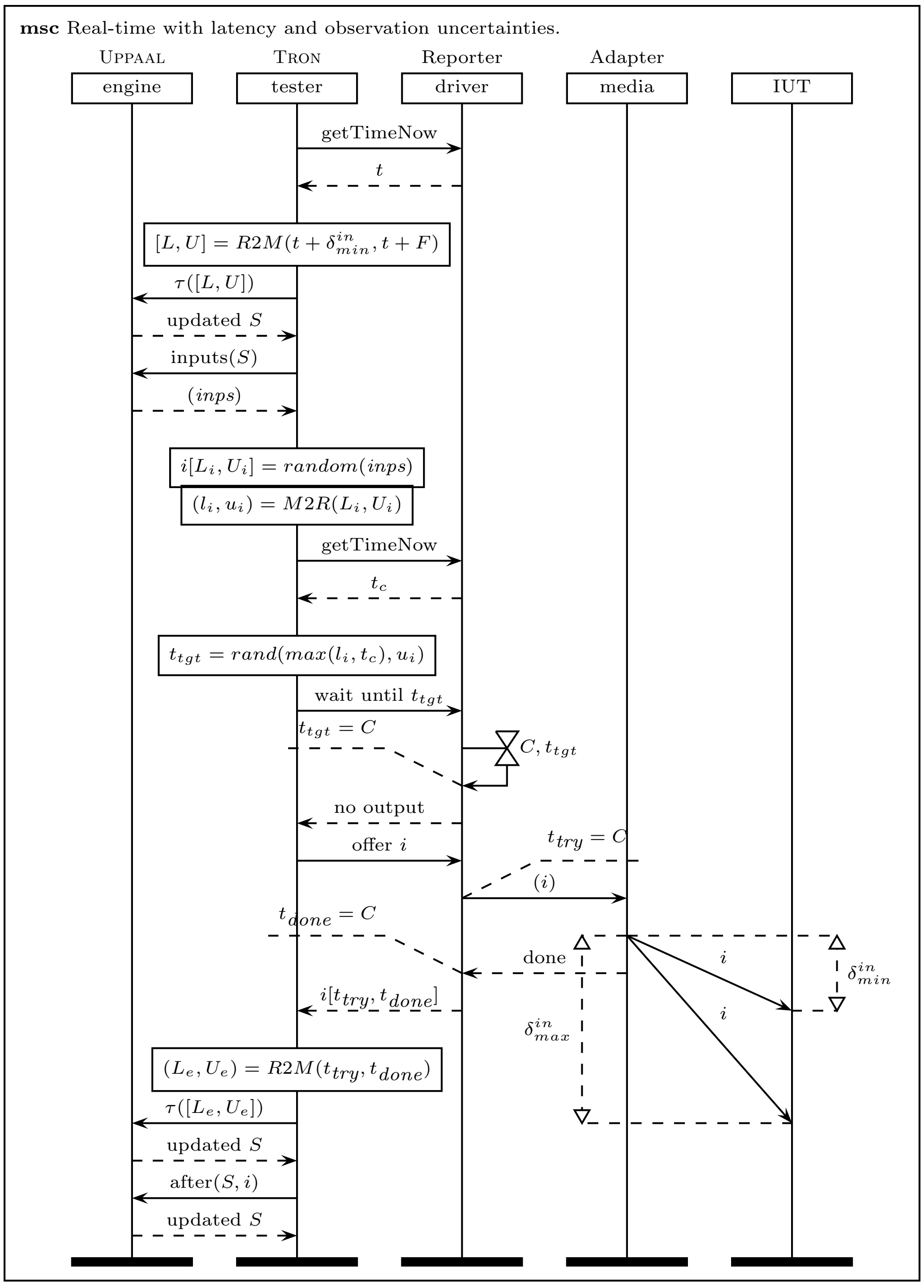


Figure 24: Scenario for offering an input to IUT and relevant timestamps in real time case with observation uncertainties, assuming $F \geq \delta^{inp}_{max}$.

which hopefully will come out as failure at some later step, and if it does not, then it is safe to conclude that such failure is not observably detectable (under our testing assumptions) and thus we should not care.

- Environment model is assumed to be enabled for all possible outputs at any time, thus there are no additional assumptions for outputs in environment model. If the environment model is not enabled then there will be false assumptions about output timestamp and therefore we cannot allow it.

  For example, the offered input should be possible at all instances between $t_{try} + \delta^{inp}_{min}$ and $t_{done} + \delta^{inp}_{max}$.

- If several I/O events are timestamped by overlapping intervals then all possible event orderings have to be considered as it is not possible to determine which event happened first. This may have some performance penalties but only when multiple events clash within an adapter (not common) thus preserving good average performance.

4.3.6 Model Time and Real Time

T is a real time value (in microseconds) of one model time unit; $\mathcal{L}$ is input scheduling latency; δ^{inp}_{min}, δ^{inp}_{max}, δ^{out}_{min} and δ^{out}_{max} are observation uncertainty parameters describing adapter I/O latency distribution (jitter). Bound strictness notation:

$$\begin{array}{rcc}
x \text{ satisfies } \textit{strict lower} \text{ bound } L & \Leftrightarrow & L < x \\
x \text{ satisfies } \textit{non-strict lower} \text{ bound } L & \Leftrightarrow & L \leq x \\
x \text{ satisfies } \textit{strict upper} \text{ bound } U & \Leftrightarrow & x < U \\
x \text{ satisfies } \textit{non-strict upper} \text{ bound } U & \Leftrightarrow & x \leq U
\end{array}$$

From Figure 23 we can derive the following formulas to convert model time units to real time and back:

R2M real time to model time for estimating *input delivery*:

$$L_{inp} = \begin{cases} \text{strict } \left\lfloor \frac{l_{inp}+\delta^{inp}_{min}}{T} \right\rfloor & \text{if } \left\{ \frac{l_{inp}+\delta^{inp}_{min}}{T} \right\} > 0 \\ \text{non-strict } \left\lfloor \frac{l_{inp}+\delta^{inp}_{min}}{T} \right\rfloor & \text{otherwise} \end{cases} \tag{8}$$

$$U_{inp} = \begin{cases} \text{strict } \left\lfloor \frac{u_{inp}+\delta^{inp}_{max}}{T} + 1 \right\rfloor & \text{if } \left\{ \frac{u_{inp}+\delta^{inp}_{max}}{T} \right\} > 0 \\ \text{non-strict } \left\lfloor \frac{u_{inp}+\delta^{inp}_{max}}{T} \right\rfloor & \text{otherwise} \end{cases} \tag{9}$$

R2M real time to model time for estimating *output origin*:

$$L_{out} = \begin{cases} \text{strict } \left\lfloor \frac{l_{out}-\delta^{out}_{max}}{T} \right\rfloor & \text{if } \left\{ \frac{l_{out}-\delta^{out}_{max}}{T} \right\} > 0 \\ \text{non-strict } \left\lfloor \frac{l_{out}-\delta^{out}_{max}}{T} \right\rfloor & \text{otherwise} \end{cases} \tag{10}$$

$$U_{out} = \begin{cases} \text{strict } \left\lfloor \frac{u_{out}-\delta^{out}_{min}}{T} + 1 \right\rfloor & \text{if } \left\{ \frac{u_{out}-\delta^{out}_{min}}{T} \right\} > 0 \\ \text{non-strict } \left\lfloor \frac{u_{out}-\delta^{out}_{min}}{T} \right\rfloor & \text{otherwise} \end{cases} \tag{11}$$

M2R model time to real time for *input scheduling*:

$$l_{inp} = \begin{cases} L_{inp} \cdot T - \delta^{inp}_{min} + \varepsilon & \text{if } L_{inp} \text{ is strict} \\ L_{inp} \cdot T - \delta^{inp}_{min} & \text{otherwise} \end{cases} \tag{12}$$

$$u_{inp} = \begin{cases} (U_{inp} - 1) \cdot T - \delta^{inp}_{max} - \mathcal{L} & \text{if } U_{inp} \text{ is strict} \\ U_{inp} \cdot T - \delta^{inp}_{max} - \mathcal{L} & \text{otherwise} \end{cases} \tag{13}$$

ε is the smallest countable value of real time unit ($1\mu s$), it is independent from any clock resolution. Its purpose is to avoid scheduling input at the exact lower bound.

Then $[l_{inp}, u_{inp}]$ is a real time interval for which input can be delivered safely without violating constraints. If $l_{inp} > u_{inp}$ then environment requirements are too strict for such test adapter, and it is not possible to schedule such input reliably.

Note that TRON subtracts almost whole last time unit from upper bound as TRON does not know the exact timing offset within one time unit, e.g. consider situation where environment requires immediate input after some output is observed, then safe upper bound u_{inp} should be less or equal to lower bound l_{inp} (i.e. now, at the time of output) and not within one time unit as symbolic zones might suggest in the middle of time unit.

Notice that latency and observation uncertainty features can be turned off by just using value 0 (default).

4.4 Input Choices

If environment model permits several different input actions, then TRON chooses a random one and the exact delay to be performed before offering the chosen input is decided by one of the following strategies specified in `-P` option:

Random delay is chosen by a random function from an interval of possible delays computed by UPPAAL engine. This is a default setting.

Eager delay is the shortest delay from an interval of possible delays computed by UPPAAL engine.

Lazy delay is the longest delay from an interval of possible delays computed by UPPAAL engine.

Bounded by s or l delay is chosen by a random function from an interval of possible delays constrained by either upper bound s or l. If both s and l are shorter than the shortest allowed delay, then the shortest allowed delay is chosen. s stands for a "short delay" and l is "long delay", and the choice between them is resolved by a random function. The "short" and "long" semantics is not enforced but provided as a hint to developer that they can be used to constrain choices for "fast" (low time granularity) and "slow" (high time granularity) inputs.

5 Diagnostics

Currently TRON provides a verdict and simple conclusion based on last good state set. Algorithm 1 shows the pseudo-code for drawing the conclusion. *Action* is class containing data about actual input/output observed: channel, values for associated data, the interval of estimated execution time (*lowerBound* and *upperBound*). *Choice* is class containing data about possible choice for input stimuli: channel, values for associated data, the interval of enabled time (*minBound* and *maxBound*). *Choice* objects are generated in UPPAAL engine, while *Action* objects are created, decoded and time-stamped by driver.

Function `Conclusion(`*StateSet backup, Action a, Choice c*`)`: verdict – based on last good state set.

```
$A_{in}$ =EnvOuput(backup); $A_{out}$ =ImpOuput(backup);
if $a$ then   // did state set become empty upon observable I/O?
    if $a.isInput$ then          // there was choice c for this input
        print "Decided to input c but executed as a";
        print "The target state was:" c.targetState;
        if $c.maxBound < a.lowerBound$ then
            return Inconc("Input executed too late...");
        else if $a.upperBound < c.minBound$ then
            return Inconc("Input executed too early...");
    else                                          // a is an output
        print "Got unacceptable output a";
        print "Expected outputs:" $A_{out}$;
        boolean tooLate=false, tooEarly=false;
        forall $c_o \in A_{out}$ do   // Let's look at all possible outputs
            if $a.chan{=}{=}c_o.chan$ then   // bug: data part is ignored
                if $a.upperBound < c_o.minBound$ then
                    tooEarly=true;                 // disjunct intervals?
                if $a.lowerBound > c_o.maxBound$ then
                    tooLate=true;                  // disjunct intervals?
        if $tooLate \wedge \neg\, tooEarly$ then
            return Failed("Output produced too late");
        else if $\neg tooLate \wedge tooEarly$ then
            return Failed("Output produced too early");
        else return Failed("Observed unacceptable output");
else          // there was no observable I/O, except time delay
    print "Could not delay any more (to the last time-window)";
    if $A_{out} \neq \emptyset$ then
        print "Output expected:" $A_{out}$;
        if $A_{in} \neq \emptyset$ then
            print "IUT expected input" $A_{out}$;
            return Inconc("Tester could have offered input, but did not
            observe either...");
        else return "Failed(IUT failed to produce output in time");
    else if $A_{in} \neq \emptyset$ then
        print "Input expected:" $A_{in}$;
        return Inconc("Tester failed to offer an input in time");
    else return Inconc("Model contains deadlock(s)");
return Inconc(Empty stateset. Please report it.);
```

6 Limitations and Workarounds

6.1 Modeling

Not all UPPAAL models are suitable for testing using TRON, e.g. most commonly used partial order reduction techniques (including symmetry reduction) should be abandoned here, since it restricts only some (specific) order of events which is not always the case in the real world. We recommend to follow the system model partitioning as close as possible (discussed in Section 2.2).

6.2 Platforms

Common versions of Linux and Windows implement soft-real-time schedulers which means that a processor assignment to a process may be postponed, threads may not run immediately after they acquire necessary resources and get unblocked and hence program execution may be delayed. The delay is called *scheduling latency* and soft-real-time schedulers give only probabilistic guarantees that a process will eventually get the processor. Linux strives to guarantee 1ms scheduling latency under low load (few processes demanding a processor) and 10ms latency under high load (many processes demanding processor at the same time). Fast and fair schedulers for desktop computers are still being actively developed (see e.g. Ingo Molnar's work on O(1) and CFS schedulers, available in Linux by default since v.2.6.25). Hard real-time schedulers provide firm guarantees but require different approach and needs more investigation, perhaps test generation algorithm redesign (e.g. look-ahead for more events) to gain more predictable performance in cases where short response time is needed.

To make matters even worse, the communication between TRON and IUT does not happen instantaneously (as common in models), hence *communication latency* also plays role in real-time testing. Normally the operating system sockets implement algorithms to optimize the network usage which result in accumulating (buffering) and delaying short messages.

As a result, one may experience some strange behavior, such as TRON reporting a test failure on a supposedly correct implementation (IUT did not get the processor to produce the required output in time), TRON reporting test inconclusive as TRON failed to offer input in time (TRON did not get the processor in time).

The virtual time framework is proposed as an abstraction from scheduling and communication latencies, see Section 3.6 for details. The following is a list of tips-and-tricks to address the issues above if the final implementation needs to be tested and the virtual time framework is not an option:

1. Make sure that computer is not heavily loaded:

   Linux: enter `uptime` at command prompt and see what is the load average. Load is an estimate how many processes ask for the processor at the same time. Loads above 1 are considered to be high. Use `top` to inspect which processes use processor the most.

   Windows: Use Task Manager to inspect running processes: click Start→Run, type `taskmgr` and hit enter.

Notice that "nice" programs (low priority computing in the background, such as SETI@Home) pollute the processor cache and result in larger scheduling latencies for interactive tasks. Cache pollution is even more noticeable on processors with reduced cache (e.g. Intel Celeron line).

2. Multi-core or multi-processor computer is preferred.

3. Use latest stable Linux kernel if possible (see `uname -a`), as the scheduler is constantly being improved and tuned for interactive tasks. Windows scheduler seems completely unpredictable.

4. TRON automatically attempts to create a real-time priority thread with round-robin scheduling. Usually such requests are denied with ordinary user privileges, but granted if run with super-user (`su`). Such priority will preempt almost any process on the system including terminal and entire windowing system, so consider this option only if confident that test does not need manual interruption.

5. Avoid using graphical user interface (GUI), as GUI programs are identified as interactive and are given a priority boost, hence may interfere. Smartlamp example has `-N` command line option to disable the GUI and use only the necessary threads.

6. Disable Nagle's algorithm in TCP/IP sockets to reduce the communication latency:

   Java: `Socket.setTcpNoDelay(true)`.

   C: `setsockopt(socket, IPPROTO_TCP, TCP_NODELAY, &1, sizeof(int))`.

7. Add "adapter" models reflecting the input and output signal delays between TRON and IUT. Try to keep adapter models simple: avoid output buffering if possible, expect as few simultaneous outputs as possible. Long output buffering chains in the model with non-deterministic IUT model may dramatically degrade TRON performance (as TRON will have to be prepare long in advance for possible output even if no output have happened). Notice that this is not a problem for input "adapter" models (as TRON decides on input events). Possible output event analysis performance can be the main bottleneck for how fast TRON can issue inputs.

8. Experiment with `-u` option which specifies that input and output events may get delayed (in the adapter) for some amount of time. The two-parameter variation is safe to use, but the four-parameter variation is not completely implemented and may have correctness issues.

References

[1] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In *Proc. of 4th Int. School on Formal Methods for the Design of Computer, Communication, & Software Systems (SFM-RT04)*, number 3185 in LNCS, pages 200–236. Springer, 2004.

[2] Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing Graphs with dot.* AT&T Labs, February 2002.

[3] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. In *Software – Practice and Experience*, Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA, June 1999. AT&T Labs, John Wiley & Sons, Ltd.

[4] IEEE. *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language].* IEEE, New York, NY, USA, 1996.

[5] Henrik Ejersbo Jensen. *Abstraction-Based Verification of Distributed Systems.* PhD thesis, Aalborg University, June 1999.

[6] Henrik Ejersbo Jensen, Kim Guldstrand Larsen, and Arne Skou. Scaling up uppaal automatic verification of real-time systems using compositionality and abstraction. In *FTRTFT '00: Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–30, London, UK, 2000. Springer-Verlag.

[7] K.G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In *Formal Approaches to Testing of Software*, Linz, Austria, September 21 2004. Lecture Notes in Computer Science.

[8] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software – Practice and Experience*, 25(7):789–810, July 1995. http://www.antlr.org/.