Online Testing of Real-time Systems
Ph.D. Thesis

Marius Mikučionis marius@cs.aau.dk
Department of Computer Science, Aalborg University

May 10, 2010

Contents

Chapter 1

Introduction

Our lives are more and more surrounded with embedded software devices, like intelligent agents maintaining our households and mobile phones becoming a virtual equivalent to Swiss army knife crammed with increasing number and ever more interacting features. The industrial picture is even more extreme: global positioning devices provide up-to-date information for distributed logistics, robots help automate the production processes, controllers help steer chemical plants, micro-climate controllers looking after life-stock, etc.. Embedded devices provide unique services: they help us to achieve our goals and overcome human limitations such as reaction speed, measurement precision, long distance and non-disruptive communication, continuous and non-interruptive availability at a tiny cost of energy supply.

As embedded devices are being applied in broader areas, in addition to their services, devices should require little or no maintenance, hence be adaptive in a range of environments. In order to overcome those difficulties most modern devices come as a combination of specialized hardware and sophisticated software embedded into their environments.

For example, mobile phones used to be the tools just for communication and the main task was to relay a speech to another side of a network over radio waves and land lines. Today a phone is more like a mobile computing platform equipped with all kinds of physical senses which can measure the geographical location, acceleration, direction and help user orient herself in a physical world. Industrial controllers are armed with devices for measuring light, sound, temperature, pressure, motion, and devices for influencing the the state of a system like lamp, speaker, motor, valve, heater, cooler.

Figure 1.1 shows the components of an embedded system which are connected via sensors and actuators, communicate and synchronize within global time, therefore each of them must meet functional as well as real-time requirements to ensure correct functioning of a whole system. The vision is that we can deduce the properties of overall composed system by analyzing individual components and their requirements.

Recent trends show promising results in model driven development where the requirements are described in terms of design models and the models are automatically analyzed and verified by tools like model-checkers and theorem provers. Such early designs give confidence that the right system is being built in the right way, however, by their own nature, models represent mere ab-

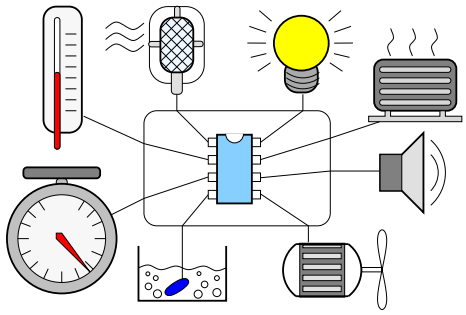


Figure 1.1: Embedded controller communicating with environment via sensors and actuators.

stractions of a system and lack implementation details thus leaving room for potential problems in the actual implementation. Thus even if rigorous correct-by-construction methodology is exercised, one can never be sure that the implementation behaves as intended by original requirements. On the other hand, testing has been a dominant verification technique in software industry which complements code inspections and other analysis techniques. Despite enormous need and effort, software testing takes about 1/3 of overall development resources, remain ad-hoc and error prone to human errors. Moreover handling of real-time aspects is even less systematic.

The goal of this thesis is to develop model-based testing tool for real-time systems by using model-checking techniques.

1.1 Testing

In general it is agreed that *testing is a structured and controlled experiment that involves running an actual system with a goal of estimating its quality.* The following describes more concrete instances and scope of this thesis:

An actual system is called an *implementation under test* (IUT). In our case the IUT is a component that can be isolated and treated as a *black-box*, whose neither structure nor state can be observed directly. We consider a *system level* testing.

The quality can be described in terms of *functional behavior* and *real-time requirements.*

The structure of an experiment setup is assumed to be *close to realistic deployment* of IUT, where the environment is realized or emulated by a tester and test harness.

The control of an experiment is automated via *tool* support.

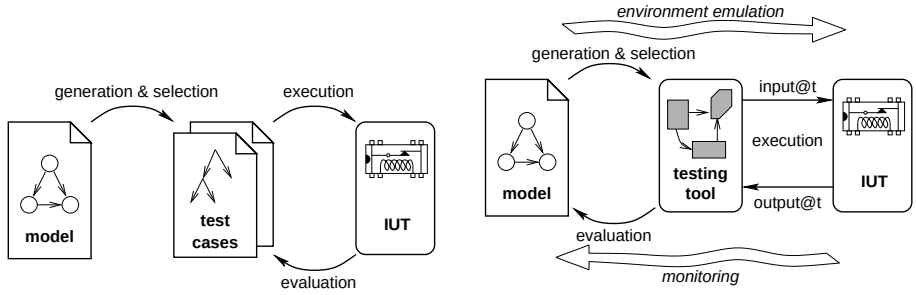
The hypothesis of an experiment is that the IUT behaves like a given set of requirements specified as a formal model and in particular we are concerned with *conformance* relation which we define later.

Figure 1.2a shows the following activities in offline testing:

Generation and selection of *test cases* from requirement specification based on a given test purpose or objective.

Execution of the test cases on IUT producing a *test result*.

Evaluation of the test results against the requirement specification producing a *test verdict*, saying whether the test has *passed* (no fault observed), *failed* (erroneous behavior observed) or is *inconclusive* (the objective of the test has not been achieved).



(a) Offline. (b) Online.

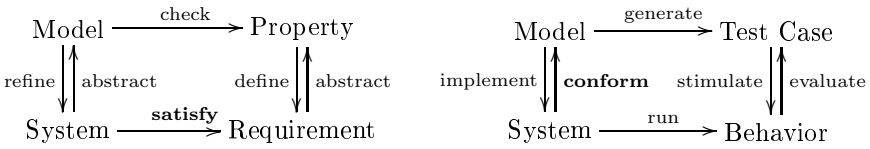
Figure 1.2: Model based testing frameworks.

Tests for interactive systems usually execute various permutations of actions in a sequence in order to exercise different functionality of the system. The possible set of test cases are exponentially large in terms of length. In order to save the storage space, model based approach allows us to combine and perform testing activities in parallel leading to on-the-fly tests where parts of the test called *test primitives* are generated on-demand while previous test primitives are executed and evaluated. Un-timed I/O systems distinguish a discrete sequence of inputs and outputs. In the timed system test setup we assume that input and output events are asynchronous and may happen independently at the very same time while global time is affecting both IUT and tester thus we prefer to call such tests as *online tests*. Figure 1.2b shows an online test framework where test generation and evaluation together with test primitive execution effectively result in an environment emulation and IUT monitoring.

1.2 Model Based Development

In this section we argue that model-checking and model-based testing are complementing activities in gaining confidence in a model and a system rather than competing. Both activities share a lot of common elements which ought to be reused.

Figure 1.3a shows relations between model, system, properties and requirements in model-checking. In a top-down development approach a developer describes a system by designing a system *model*. Then model-checking tools can be used to automatically check that the model satisfies certain *property* formula describing the *requirements* for the system. Once the model is acquired, further refinements or implementation is carried out resulting in a *system*. In



(a) Model-checking. (b) Conformance testing.

Figure 1.3: Model-checking and conformance testing.

order to use the result of model-checking and prove that the system satisfies the requirements, one has to prove that either the implementation properly refines the model (e.g. via proven code generation) or model is a proper abstraction of an implementation (e.g. via abstract interpretation). A buttom-up approach is also possible: a model is built by looking at disassembled system (e.g. by reverse engineering). Either way, establishing the connection between the system and its model involves formal proofs which are hard, has limited automation support and hence may be error-prone.

Figure 1.3b shows the relations between a model, a system, test cases and behavior in terms of observable traces. In contrast to formal proofs, model based testing is a cheap technique to establish a relation between a model and a system by empirical means: generate test cases from a model, execute and evaluate them on an actual system run. Such a relation is not proved rigorously, but observed through exposed behavior of the system, therefore some functionality may be left unexamined and faults hidden. On the other hand, testing exercises the system implementation details (including operating system and underlying physical hardware) which are not subjected in formal proofs, moreover if faults are never propagated to the output then they are irrelevant.

Figure 1.4 shows a projection of a model state space in gray, the system execution paths in black curves and stars denote functionality of interest. The model state space can be acquired via reachability analysis of a model, and system run can be deduced by observing execution. Ideally, we would want that system run would have a corresponding trace within reachable state space of a model. Then, the problem of test case generation is equivalent to finding particular sequence of stimuli that drives the system into a state of interest; and the problem of test evaluation is equivalent to checking whether the exposed state of the system is within model state space. If the system behavior falls out of the model state space, then our test should declare a failure, because our model-hypothesis about the system does not hold. In practice, especially under the black-box assumptions and as a consequence of imprecise observations, it might not be possible to deduce the state of the system precisely, thus it is more appropriate to operate on possible state set estimate of the system. Then again, the model-checking tools provide symbolic techniques for how to operate and store such states, so they may be reused for model-based testing purposes.

1.3 Thesis

We claim that *real-time model-checking techniques can be used to automate testing of real-time systems with a high degree of confidence in system's quality.*

In order to investigate the thesis we explore the following research questions:

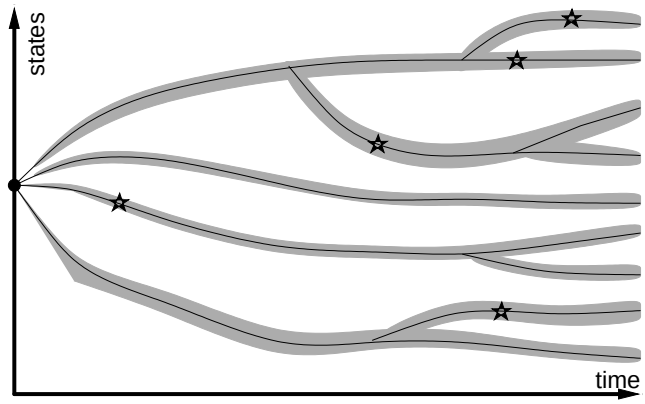


Figure 1.4: Model state space and system state trace projection over time.

Question 1. How can testing theory be extended to support testing of real-time requirements and online execution aspects?

Question 2. How can model-checking techniques be reused to achieve online testing goals?

Question 3. How to relate a model state space with physical observations in a sound and practical way?

Question 4. Is real-time online testing feasible in practice?

The method of the thesis is to extend black-box conformance testing theory for timed systems, implement the theory in a testing tool and evaluate the testing framework on an industrial case study.

1.3.1 Structure of the Thesis

Figure 1.5 shows a structure of the thesis which covers contributions from theory, through tool implementation, experimentation and adapter framework to industrial application.

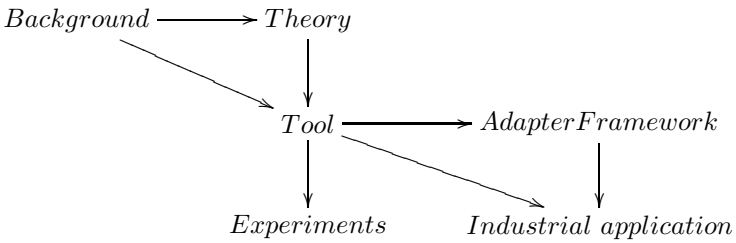


Figure 1.5: Structure of the thesis.

Chapter 2 outlines the underlying concepts and theories used throughout the thesis and describes the prior state of the art in formal methods for model-based testing. It starts with definitions, explains techniques behind black-box conformance testing, symbolic model-checking of real-time systems and gives an abstract overview of UPPAAL implementation components later reused in implementation of online testing tool.

Chapter 3 presents the central part of the thesis: the extended the theory for real-time systems testing, abstract and symbolic online testing algorithms and how an on-the-fly model-checking engine is adopted to generate, execute and monitor the test run online. In addition, a heuristic algorithm is provided to computed basic diagnostics based on last valid state estimate.

Chapter 4 describes the test adapter framework, provides methodological guidelines on how to decouple tester and IUT to gain advantage over flexible online test setup and proves that the test adapter protocol satisfies assumptions from theoretical part of the thesis.

Chapters 5 describes empirical experiments performed on the online test tool: starting with examination of correctness of basic modeling features, performance benchmarks and fault detection capability.

Chapter 6 demonstrates the tool application on an industrial case study. The study describes a number of modeling patterns for specifying typical real-time constraints as well as a bit of quantitative functionality.

Chapter 7 outline conclusions of the thesis and future work directions.

1.3.2 Contributions

The following outlines the main contributions of this thesis:

1. We formally define the real-time extensions for input/output conformance testing theory first appeared in [48] and later discovered in [13, 38]. We propose further extensions that supports design and documentation of environment assumptions. The results are published in [42].

2. Online testing tool implementation using state-of-the-art model-checker. Online algorithm published in [42, 47], and the tool has been demonstrated in [46].

3. We propose an adaptation framework for execution of tests against real-time systems.

4. We measure the performance and error capability of online testing tool by conducting various experiments. Some of the early performance results appear in [42].

5. Case study of online testing tool application on industrial time-constrained system. The first iteration of this case study is published in [43].

The principles of the online testing together with offline testing methods based on Uppaal are jointly published in [30].

1.4 Related Work

The thesis touches aspects of both theoretical and empirical study, thus we provide a brief overview of most related theoretical frameworks as well as tool implementations.

1.4.1 Theoretical Frameworks

The thesis is mostly influenced by a black box conformance testing framework by Jan Tretmans [60, 61] and its on-the-fly testing tool implementation TorX [62]. The approach is based on untimed labelled transition systems (LTS), input/output systems and ioco conformance relation, which is a promising start for considering timed systems described by timed labelled transition systems with inputs and outputs. In [7] Axel Belinfante tried to apply TorX for timed systems in an ad-hoc manner and concluded with:

> "More systematic study is needed, for example regarding the theory, regarding the modelling, and regarding (making of) the Adapter, to name just a few items."

Later several timed extensions to **ioco** relation are proposed independently by Mikučionis et al [42, 47, 48], Briones et al [12, 13] and Krichen et al [38, 40]. Table 1.1 shows a comparative summary of those frameworks similar to Figure 3.8 in [12].

	Specification	IUT	Relation	Test
Briones et al [12]	TLTS non-deterministic internal transitions no forced inputs time divergent	TLTS non-deterministic internal transitions no forced inputs time divergent weak input-enabled	**tioco**$_M$ out set: outputs with time, quiescence with (bounded) time	tree
Krichen et al [38]	Open TA with lazy, delayable and eager edges internal transitions non-blocking	TA internal transitions input-enabled non-blocking	**tioco** out set: outputs and time	total function; tree
Mikučionis et al [42]	UPPAAL TA closed by environment e non-deterministic internal transitions input-enabled non-blocking e is input-enabled	TA non-deterministic internal transitions input-enabled non-blocking	rtioco$_e$ out set: outputs and time	tree, part of e

Table 1.1: Timed **ioco** extensions.

Both [12] and [38] frameworks are motivated mostly by theory while [42] is motivated by practical reasons. In particular, [12] distinguish weak and strong input enabledness, also stress the presence of internal τ transitions – both assumptions are important for showing theoretical results, however in practice they are mere modeling artefacts and indistinguishable from non-determinism[1]. In particular [42] tool implementation assumes only weak input-enabledness in the specification and only the existance of TA structure in IUT is assumed. **tioco**$_M$ [12] is backward compatible with **ioco** [60] in an almost straight-forward way, while [38] and [42] would need to address the notion of quiescence implicitly

[1] Internal τ transitions by definition are unobservable, hence can be replaced by observationally equivalent non-deterministic TLTS. Strong and weak input-enabledness also result in equivalent observable traces

in a modeling pattern. The weakness of input-enabledness is not so apparent in [40] as authors offer a method to limit the input-enabledness assumption by parallel composition.

Interestingly, [38] distinguish analog-clock and digital-clock tests, while specification clocks in [42] are just modeling elements used to express relations between events and may have no counterparts in the real world. The distinction is most vivid in digital-clock tests [38] where tester and IUT share a global clock process issuing highest priority discrete tick events which help tester and IUT to agree and come up with homogeneous order of input and output events despite being separate entities at different physical locations of real world space. The same problem is avoided in analog-clock [38] tests.

In contrast, [42] online test tool implements a decoupled tester and IUT system, where the two independent entities are connected via input/output communication channels:

- The closed nature of UPPAAL models requires that entire system is modelled in the specification: requirements for IUT, assumptions about environment and communication between them. For sound theoretical results the framework also assumes that IUT is isolated from the rest of the world which is implicit in the theoretical frameworks above.

- Online test tool uses an auxiliary clock in the specification model used to refer to tester's own physical clock separated from IUT thus effectively resulting in a decoupled system where tester and IUT may potentially have a different view of input/output event ordering.

- The possible input/output interleaving between IUT and tester are accounted by models of communication processes included in the specification, thus making the nature of communication channels explicit, potentially exposing their realistic imperfections, such as being non-instantaneous which is no longer negligible in real-time systems.

Henrik Bohnenkamp and Alex Belinfante [11] adopts a timed conformance relation closely related to and motivated by **tioco**$_M$ [12, 13] and implements a testing framework with quiescence using timed safety automata [27]. [11] acknowledges that timed testing is not easy due to conflicting requirements of theory and physical reality: inevitable time progress implicitly impose real-time constraints on testing tool, infinitely precise notions of timed automata conflict with impracticality of measuring real-valued time.

Our approach to the above problems is to use an overapproximation of time measurements and analyze all possible behaviors from that point. As a result, the online test precision is determined by the specification and tester's clock precision, the execution is as fast as execution platform and test interfaces allow – all are taken into account explicitly in the specification without sacrificing distributed setup or real-valued precision. Moreover the developer has native modeling means of guiding the tester on what functionality is important to test, including stress tests that require fast reaction times from testing execution tool. Finally the testing tool itself is able to detect that the actual stimuli execution does not violate the required timing.

In addition, the thesis describes:

- The design of test adapter which support simultaneous input and output events between tester and IUT, time-stamping real events and relating them to the model state space. [32] may provide insight to proving the correctness of the time-stamping approach.

- Testing tool design using software parts of UPPAAL [5].

1.4.2 Tools

Test evaluation and monitoring: runtime monitoring [56], fault diagnosis [63].

Briones and Röhl [15] provide a detailed overview of three test derivation techniques from timed automata: from event recording automata [51], from deterministic timed automata [58], from testable timed systems [16].

There are many variants of test generation from timed automata based on UPPAAL alone depending on various assumptions about the IUT and test purposes:

- Optimal test generation techniques: time-optimal [31] provide a methodology on how to decorate a model and use model-checker to derive sequences which can be used as test cases; UPPAALCOVER [29] provides tool support for expressing various coverage criteria and automatically derive test sequences with optimal coverage using modified model-checker engine.

- Test case derivation using timed games: game-theoretic [21] for white-box testing when IUT is seen as opponent in a testing game; cooperative testing [20] for white-box testing when a winning strategy does not exist in general but goal is achievable with some cooperation of the IUT, with partial observability [22] where the IUT can expose part of its state and thus help finding a winning strategy.

Table 1.2 shows a brief comparison of tools which are closest to our framework. We distinguish the specification formalism, assumptions about IUT and enumerate testing characteristics that make a particular tool to stand out from others.

Reactis [34, 57] provides model-based testing via simulation, it is integrated within Simulink framework and uses Stateflow models as specification. Simulink assumes deterministic models and Reactis provides facilities to generate tests based on coverage criteria and store them as sequences of inputs and outputs which can be played against real IUT (connected to Simulink) or against Simulink models. The user is expected to inspect various plots of the observed behavior and determine whether the behavior is acceptable. If the test does not proceed as user expects, then Reactis offers features to replay and step through the model execution for diagnostic purposes.

In contrast to Reactis, STG [18, 55] uses formal conformance relation for determining the correctness of the IUT behavior. STG uses Input Output Symbolic Transition System (IOSTS, an extension of IOTS with symbolic data representation) as specification and test purpose models and constructs test cases in a form of IOSTS. The resulting IOSTS can then be translated into C++ code for execution on C++ object. STG does not offer support for real-time, but it is interesting that they provide explicit support for test purposes and use symbolic representation for data.

Tool	Specification	IUT	Test approach
Reactis [34, 57]	Simulink Stateflow, deterministic, discrete time	simulation or hardware-in-the-loop	Conformance of behavior to user expectations, offline, coverage based, Simulink integration
STG [18, 55]	NTIF (LOTOS-like, IOSTS-based), untimed, non-deterministic	C++ object	Conformance relative to test purpose, offline construction of deterministic IOSTS, symbolic data representation
Timed TorX [11]	Safety TA, non-deterministic, dense time	partially observable for absence of τ, shared clock	tioco$_M$ with quiescence, on-the-fly expansion to zone automata, absolute time, fixed precision time discretisation
TTG [39]	TA with urgency, non-deterministic, dense time, input enabled, explicit clock model	partially observable, input enabled, shared clock	tioco, offline observer construction, coverage based, discretised based on shared clock model, time relative to shared clock ticks, expanded state representation
Uppaal Tron [42]	Uppaal TA, non-deterministic, dense time, s-input enabled, e-input enabled	black-box, input enabled	rtioco$_e$, online, randomized, guided by environment model, symbolic state estimate representation, absolute time, interval time-stamps, local clock

Table 1.2: Real-time testing tool comparison.

Timed TorX [11] is a continuation of TorX adding a support for time in on-the-fly tests. The paper claims to follow conformance relation tioco$_M$ [13] and assumes that it is possible to instrument the IUT with check for quiescence and both tester and IUT share the same global time reference clock (run on the same computer). Timed TorX expands safety timed automata into zone automata using symbolic techniques [10]

Uppaal Tron [42] uses rtioco$_e$ [2], which takes the IUT environment into account, in a tradition of scientific experiments that all assumptions should be explicit and at the same time provide engineer with a way of specifying test purposes. The framework uses Uppaal timed automata with much richer modelling constructs than timed automata alone. The usage of Uppaal engine comes with a lot of benefits: symbolic representation and flexible analysis of time constraints – both crucial for performance and flexible test setup where global time reference clock is not shared with the IUT. The framework provides methodical guidelines on how to model the system including the test adapter so that the reference clock need not be shared. It is apparent that adapter model inclusion is only practical with a compact representation of state set estimates like in Uppaal and infeasible when the state space is expanded reaching exponential size like in offline testing using [39, 58]. The decoupling of the global time reference clock appears a crucial ingredient in resolving the input/output

[2] rtioco$_e$ is further extension of rtioco [48], later discovered by [13] and [38] and referred as tioco

concurrency problems which manifest as different observable I/O sequences at the tester and the IUT sides due to interleaving in the adapter. UPPAAL TRON uses a concept of interval-time-stamping to record the imprecise measurement of I/O event timing (the problem of imprecision in time measurements is also acknowledged by [11] framework) and helps inferring the current system state set estimate by an over-approximation. As a result of explicit environment and adapter models, UPPAAL TRON continuously monitors itself checking whether the environment emulation is fair according to the model, and the tool is aware that inputs may be delayed and the IUT should be treated fairly with respect to potentially delayed input arrival.

A completely different field of control theory provide a very similar framework of hardware-in-the-loop testing. In particular observer-controller setup is similar to UPPAAL TRON: developer provides a model of a plant under control, then control methodology provide a way of computing an observer component that estimates the state of plant based on its outputs, control methodology provide a way of computing a controller component providing inputs to the plant based on the state estimate from the observer. Observer-controller methodology operates on deterministic continuous functions described by differential equations and the state estimate is a single vector value which is assumed to be close to the actual plant state, the correctness of the system then depends on classical control criteria like system stability. In contrast, UPPAAL TRON considers non-deterministic model with very simple dynamics, the state estimate encodes the whole set of allowed states, and correctness of the system is determined by hard-real-time constraint and precise functional value check.

Chapter 2

Background

The goal of this chapter is to provide a semantical framework explaining the main concepts and notations used throughout the thesis. We start with timed input/output transition systems and timed automata as basic building blocks for specifying behavior of timed systems, then we propose widely accepted timed extension of conformance relation (early results on timed conformance relation from [48]), define a composition of timed systems which allow compositional specifications and form the basis for further timed extension of conformance relation, then we introduce symbolic techniques from UPPAAL to be used for timed automata specification analysis.

2.1 Basic Modeling Constructs

First, we describe the semantical layer of timed input/output transition systems and timed traces – the notion used in conformance relation. Then we define timed automata as modeling formalism and its semantics in terms of timed input/output transition systems.

2.1.1 Timed Input/Output Transition Systems

Labelled transition systems (LTS) is a popular formalism to describe the formal semantics of more complex and powerful constructs. In particular we consider timed transition systems with inputs and outputs where transitions are labelled with either real-valued number denoting the time passage or an action label expressing instantaneous input, output or internal action.

We denote the set of inputs by A_{inp}, the set of outputs by A_{out} and the set of all observable actions by $A = A_{inp} \cup A_{out}$. We assume that input and output action sets are disjoint $A_{inp} \cap A_{out} = \varnothing$. We also have an internal action label $\tau \notin A$ and use $A_\tau = A \cup \{\tau\}$ to denote the set of all action labels.

Definition. 2.1 Timed I/O transition system $\mathcal{S}$ *is a tuple* $TIOTS(S,s_0,A_{inp},A_{out},\rightarrow)$, *where* S *is a set of states,* $s_0 \in S$, *and* $\rightarrow \subseteq S \times (A_\tau \cup \mathbb{R}_{\geq 0}) \times S$ *is a transition relation, written* $s \xrightarrow{\alpha} s'$ *if* $s, s' \in S$, $\alpha \in (A_\tau \cup \mathbb{R}_{\geq 0})$ *and* $\langle s, \alpha, s' \rangle \in \rightarrow$, *satisfying the usual constraints of:*

- zero delay: *state may stay the same:* $s \xrightarrow{0} s$,

- time determinism: *if* $s \xrightarrow{d} s'$ *and* $s \xrightarrow{d} s''$ *then* $s' = s''$,
- time additivity: *if* $s \xrightarrow{d_1} s'$ *and* $s' \xrightarrow{d_2} s''$ *then* $s \xrightarrow{d_1+d_2} s''$,

where $d, d_{1,2} \in \mathbb{R}_{\geq 0}$ *are non-negative real numbers.*

In addition to concrete $\rightarrow$ transitions, Definition 2.2 provides internal transition (τ) abstracted transition relation which allows to reason about observable traces without examining system implementation details.

Definition. 2.2 *Let* $a, a_{1\ldots n} \in A$, $\alpha \in (A_\tau \cup \mathbb{R}_{\geq 0})$, $\gamma_{1\ldots k} \in (A \cup \mathbb{R}_{\geq 0})$, $d, d_{1\ldots n} \in \mathbb{R}_{\geq 0}$ *and* $s \in S$ *then:*

- $s \xrightarrow{\alpha}$ *iff* $\exists s' \in S \,.\, s \xrightarrow{\alpha} s'$, *meaning that* α*-transition is enabled in* s*;*
- $s \stackrel{a}{\Rightarrow} s'$ *iff* $\exists s' \in S \,.\, s \xrightarrow{\tau}{}^* \xrightarrow{a} \xrightarrow{\tau}{}^* s'$, *where* τ^* *means zero or more internal* τ *transitions;*
- $s \stackrel{d}{\Rightarrow} s'$ *iff* $\exists s' \in S, d_{1\ldots n} \in \mathbb{R}_{\geq 0} \,.\, s \xrightarrow{\tau}{}^* \xrightarrow{d_1} \xrightarrow{\tau}{}^* \xrightarrow{d_2} \xrightarrow{\tau}{}^* \ldots \xrightarrow{\tau}{}^* \xrightarrow{d_n} \xrightarrow{\tau}{}^* s'$, *where* $d = \sum_{i=1}^{n} d_i$, *meaning* τ*-abstracted delay* d *transition relation;*
- $s \stackrel{\sigma}{\Rightarrow} s'$ *iff* $\sigma = \gamma_1\gamma_2\ldots\gamma_k$ *and* $s \stackrel{\gamma_1}{\Rightarrow}\stackrel{\gamma_2}{\Rightarrow} \ldots \stackrel{\gamma_k}{\Rightarrow} s'$, *meaning that trace* σ *from* s *leads to* s'*;*
- $s \stackrel{\sigma}{\Rightarrow}$ *iff* $s \stackrel{\sigma}{\Rightarrow} s'$ *for some* $s' \in S$, *meaning that trace* σ *can be observed starting from* s*;*

In our testing method we are going to estimate the system state after an observable trace, thus Definition 2.3 provides a notion of a set of reachable states after and action or delay or a trace has been observed.

Definition. 2.3 *Let* $\gamma \in (A \cup \mathbb{R}_{\geq 0})$, $\sigma \in (A \cup \mathbb{R}_{\geq 0})^*$, $s \in S$, *then:*

- s after $\gamma \stackrel{def}{=} \{s' \in S \mid s \stackrel{\gamma}{\Rightarrow} s'\}$ *denotes the set of reachable states after observing* γ*;*
- s after $\sigma \stackrel{def}{=} \{s' \in S \mid s \stackrel{\sigma}{\Rightarrow} s'\}$ *denotes the set of reachable states after observing* σ*.*

Definition 2.4 specifies formally a few useful properties. Input enableness requires that the system should not block and should accept whenever the input is offered. We may distinguish strong and weak input enabliness where every state has to be able to consume input or the system may be allowed to do a sequence of internal transitions before consuming the input respectively. Note that the strong and weak input enableness are indistinguishable when dealing with observable traces. We assume that the system cannot block the time and in some theoretical results it is important to assume that the system is deterministic.

Definition. 2.4 *Some properties of* $TIOTS(S, s_0, A_{inp}, A_{out}, \rightarrow)$*:*

- strongly input enabled: $\forall s \in S$, $\forall a \in A_{inp} \,.\, s \xrightarrow{a}$*;*
- weakly input enabled: $\forall s \in S$, $\forall a \in A_{inp} \,.\, s \stackrel{a}{\Rightarrow}$*;*

- time non-blocking: $\forall s \in S, \forall d \in \mathbb{R}_{\geq 0}$, $\exists \sigma = d_1 o_1 \ldots o_n d_{n+1}$ *s.t.* $s \stackrel{\sigma}{\Rightarrow}$ *and* $\sum_i d_i \geq d$;
- deterministic: $\forall s \in S$, $\forall \gamma \in (A \cup \mathbb{R}_{\geq 0})$ *whenever* $s \stackrel{\gamma}{\rightarrow} s'$ *and* $s \stackrel{\gamma}{\rightarrow} s''$ *then* $s' = s''$.

2.1.2 Timed Automata

It is tedious work to express models in labelled transition systems and is even more complicated to analyze them. It is especially true when modelling real-time constraints where time delays form infinitely many transitions. Timed automata provide compact and precise way of expressing real-time behavior, and there are feasible analysis methods for them. This section describes timed automata [1] formalism and gives formal definition on reasoning about them and next section describes the feasible symbolic analysis method used by real-time model-checkers.

Definition. 2.5 *A* timed automaton *with actions A is a tuple TA(L, ℓ_0, X, E, I):*

- *L is a set of locations,*
- *$\ell_0 \in L$ is an initial location,*
- *X is a set of $\mathbb{R}_{\geq 0}$-valued clocks which evolve at the same rate,*
- *$E \subseteq L \times G(X) \times A_\tau \times 2^{R(X)} \times L$ is a super set of directed edges with:*
  - *guarding expressions G over clocks X of the following form:* $g ::= \textbf{true} \mid \textbf{false} \mid x \sim c \mid x_1 - x_2 \sim c \mid g \wedge g$ *where $x, x_{1,2} \in X$, $c \in \mathbb{Z}$ and $\sim \in \{\leq, <, =, >, \geq\}$,*
  - *action from $A_\tau = A \cup \{\tau\}$, and*
  - *reset expressions from R which are of the form: $x := c$ where $x \in X$ and $c \in \mathbb{N}$,*
- *$I : L \mapsto G(X)$ is an invariant expression mapping for each location.*
- *Let $\overline{a}$ denote the complementary action of action $a \in A$, such that $\overline{a!} = a?$ and $\overline{a?} = a!$.*

Definition. 2.6 *The semantics of a* TA(L, ℓ_0, X, E, I) *with actions A is described by the following TIOTS(S, s_0, A_{inp}, A_{out}, $\rightarrow$):*

- $S = \{\langle \ell, \overline{v} \rangle \mid \ell \in L, \overline{v} \in \mathbb{R}_{\geq 0}^{|X|}\}$, $s_0 = \langle \ell_0, \overline{0} \rangle$,
- $A_{inp} = \{a? \mid a \in A\}$, $A_{out} = \{a! \mid a \in A\}$,
- *Delay transition:*

$$\frac{d \in \mathbb{R}_{\geq 0} \quad \forall \delta \leq d. \quad \overline{v} + \delta \models I(\ell)}{\langle \ell, \overline{v} \rangle \stackrel{d}{\rightarrow} \langle \ell, \overline{v} + d \rangle},$$

where clock values are updated uniformly by $\delta \in \mathbb{R}_{\geq 0}$ increment: $\overline{v} + \delta = \langle v_1, v_2, \ldots, v_{|X|} \rangle + \delta = \langle v_1 + \delta, v_2 + \delta, \ldots, v_{|X|} + \delta \rangle$,

- *Action transition:*

$$\frac{\alpha \in (A \cup \{\tau\}) \quad \langle \ell, g, \alpha, r, \ell' \rangle \in E \quad \overline{v} \models g \quad r(\overline{v}) \models I(\ell')}{\langle \ell, \overline{v} \rangle \xrightarrow{\alpha} \langle \ell', r(\overline{v}) \rangle},$$

where clock values are updated by reset expression $r = \cup_{i=1}^{|r|}(x_{j_i} := c_{j_i})$*:* $r(\overline{v}) = r(\langle_{k=1}^{|X|} v_k\rangle) = \langle_{k=1}^{|X|} r(v_k)\rangle$*, where* $r(v_k) = c_{j_i}$ *if* i *is the largest s.t.* $j_i = k$*, and* $r(v_k) = v_k$ *if* i *does not exist s.t.* $j_i = k$*.*

2.2 Correctness Relations

So far we have defined modeling formalism and its semantics. In this section we look at two popular implementation relations: timed trace inclusion and timed input/output conformance.

2.2.1 Timed Traces

Timed I/O transition systems capture many process details, however externally only the input/output and time details can be observed. In a black box system testing setup only the observed behavior can be considered. Definition 2.7 formally specifies the set of observable timed trace for a given TIOTS.

Definition. 2.7 Timed traces *is a set of strings of input/output actions and real-valued delays beginning from* $s \in S$ *of* $\mathcal{S} = TIOTS(S, s_0, A_{inp}, A_{out}, \rightarrow)$*:*

$$\mathsf{TTr}(s) \stackrel{def}{=} \{\, \sigma \in (A_{inp} \cup A_{out} \cup \mathbb{R}_{\geq 0})^* \mid s \stackrel{\sigma}{\Rightarrow} \,\}$$

In model based testing ultimately we want our implementation to behave like our model, i.e. the implementation behavior should be matched by the behavior of a model. If we describe our implementation and our model in TIOTS terms, then we should be able to compare all possible observable behaviors in terms of timed traces. Definition 2.8 specifies the intended relation between the implementation and its specification which intuitively says that the implementation should have only the behavior specified in the specification and no other traces should be possible.

Definition. 2.8 *Let* m *denote an initial state of the implementation,* s *denote an initial state of the specification, then* timed trace inclusion *relation is:* $\mathsf{TTr}(m) \subseteq \mathsf{TTr}(s)$*.*

In practice it is not feasible to compare the sets of traces, since they can be infinitely large and with real-valued time domain they become uncountably infinite. That is why testing can only reveal some faults but never prove their absence and we need to find a better way to use the limited resources to get the highest possible confidence that the implementation will behave like the specification.

2.2.2 Timed Input/Output Conformance

Tretmans [60] defines conformance relation ioco for untimed black-box systems based on observable input/output sequences. Intuitively m ioco s means that

tester considers all observable behavior traces σ produced by specification s, apply the trace on an implementation m and check that subsequent observation from the implementation m is allowed by specification s. The definition of ioco includes a concept of quiescence as special form of outputs when the IUT does not produce any output for an infinite amount of time.

In [48]) we extend the untimed conformance ioco relation for timed systems by replacing the domain of input/output/quiescence traces with timed input/output traces, and replacing discrete output/quiescence observations with timed outputs. Definition 2.9 shows what the expected observations are when the system is expected to be in a state mentioned in the specific state set.

Definition. 2.9 Observable outputs from the given system state:

$$\mathsf{Out}(s) = \{\alpha \in (A_{out} \cup \mathbb{R}_{\geq 0}) \mid s \stackrel{\alpha}{\Rightarrow}\}, \quad \mathsf{Out}(S) = \bigcup_{s \in S} \mathsf{Out}(s)$$

Intuitively it means that the system may produce a behavior described by the output action or a silent delay of a given duration if it is in one of the states mentioned in a state set. Note that in special cases where the set of states is empty the set of possible outputs is also empty, and if the set is non-empty then it also includes element $0 \in \mathbb{R}_{\geq 0}$ as $s \xrightarrow{0} s$ for any state s by Definition 2.1.

The conformance relation is extended in a similar fashion in Definition 2.10 which says that machine m conforms to timed specification s if and only if the machine m produces only the behavior described in the specification s after any possible trace generated by specification s.

Definition. 2.10 Timed input-output conformance relation*:*

$$m \ \mathsf{tioco} \ s \stackrel{def}{=} \forall \sigma \in \mathsf{TTr}(s).\mathsf{Out}(m \ \mathsf{after} \ \sigma) \subseteq \mathsf{Out}(s \ \mathsf{after} \ \sigma)$$

Such conformance relation extension works in the same spirit as ioco in the following senses when we need to establish the tioco relation:

1. in order to establish relation we have to try all (timed) traces σ allowed by specification s, which also implies that $s \ \mathsf{after} \ \sigma \neq \varnothing$ and $0 \in \mathsf{Out}(s \ \mathsf{after} \ \sigma)$ according to Definition 2.1 and Definition 2.9;

2. execute each (timed) trace σ on machine m, compute the possible states of specification s and check the response of the machine m against the possible responses described in the specification s, hence there are the following options:

   - m immediately issues output action a, meaning that $\mathsf{Out}(m \ \mathsf{after} \ \sigma) = \{a, 0\}$ and hence a and 0 should also be matched with outputs in the $\mathsf{Out}(s \ \mathsf{after} \ \sigma)$ set: $0 \in \mathsf{Out}(s \ \mathsf{after} \ \sigma)$ as in step 1, so consider the following options:
     - $a \in \mathsf{Out}(s \ \mathsf{after} \ \sigma)$, hence $\mathsf{Out}(m \ \mathsf{after} \ \sigma) \subseteq \mathsf{Out}(s \ \mathsf{after} \ \sigma)$;
     - $a \notin \mathsf{Out}(s \ \mathsf{after} \ \sigma)$, hence $\mathsf{Out}(m \ \mathsf{after} \ \sigma) \not\subseteq \mathsf{Out}(s \ \mathsf{after} \ \sigma)$, m tioco s is false and m ti~~o~~co s; on the other hand it means that the output a was either produced too early or it was not allowed at all (as in ioco);

- m stays silent for d amount of time and does not output anything, meaning that $\mathsf{Out}(m \text{ after } \sigma) = [0, d]$, so consider the following options:
  - $[0, d] \subseteq \mathsf{Out}(s \text{ after } \sigma)$, hence $\mathsf{Out}(m \text{ after } \sigma) \subseteq \mathsf{Out}(s \text{ after } \sigma)$;
  - $[0, d] \not\subseteq \mathsf{Out}(s \text{ after } \sigma)$, hence $\mathsf{Out}(m \text{ after } \sigma) \not\subseteq \mathsf{Out}(s \text{ after } \sigma)$, m tioco s is false and m ti~~o~~co s; but on the other hand $\exists\delta.[0, \delta] \subseteq \mathsf{Out}(s \text{ after } \sigma)$ and $0 \leq \delta \leq d$ which means that specification s allowed silent delay up to δ time and m has violated a timing deadline for producing further outputs before δ time elapses, or specification has a deadlock after δ time delay;

3. if conformance has not been violated so far then the output produced in the previous step can be appended to the trace σ and testing may continue further iteratively.

Intuitively, inputs are controlled by the tester and outputs are controlled by the implementation. The time flow is controlled by neither, but any silent time delay can be interrupted by either input or output. Hence issuing an unacceptable output or delaying too long is the only way the implementation traces could diverge from traces in the specification.

This notion of timed conformance also agrees with independently developed ones: [38] and even further extended to incorporate backward compatible quiescence tioco_M [13] and multi input/output mioco [14].

The relation tioco still requires checking uncountably many traces but importantly it separates the testing task into natural test phases: trace generation from specification ($\sigma \in \mathsf{TTr}(s)$), trace execution (computing m after σ), trace evaluation (s after σ) and verdict assignment by checking that implementation output response after the trace execution is legal according to (included into) specification.

Theorem 2.1 shows that definitions 2.8 and 2.10 are equivalent if we assume that inputs and outputs cannot be refused by the receiving party.

Theorem. 2.1 *Given an implementation* $\mathcal{M} = TIOTS(M, m, A_{inp}, A_{out}, \rightarrow)$ *and a specification* $\mathcal{S} = TIOTS(S, s, A_{inp}, A_{out}, \rightarrow)$*, which are at least weakly input enabled then timed trace inclusion and real-time input-output conformance relations are* equivalent*:*

$$m \text{ tioco } s \iff \mathsf{TTr}(m) \subseteq \mathsf{TTr}(s)$$

Proof.

$\Rightarrow$ *Assume* m tioco s *but* $\mathsf{TTr}(m) \subseteq \mathsf{TTr}(s)$ *does not hold.*
Then $\exists\rho \in \mathsf{TTr}(m)$ *but* $\rho \notin \mathsf{TTr}(s)$.
Let ρ *be the shortest such trace.*
Let $\rho = \rho'\gamma$*, where* γ *is either an action or delay.*
Then $\rho' \in \mathsf{TTr}(m)$ *and* $\rho' \in \mathsf{TTr}(s)$*, since* ρ *is the shortest trace of* m *but not of* s *and* ρ' *is shorter than* ρ.
γ *cannot be input as* $\mathcal{M}$ *and* $\mathcal{S}$ *are input enabled.*
γ *cannot be output nor delay as then:* $\gamma \in \mathsf{Out}(m \text{ after } \rho')$ *and* $\gamma \notin \mathsf{Out}(s \text{ after } \rho')$.

$\Leftarrow$ *Assume* $\mathsf{TTr}(m) \subseteq \mathsf{TTr}(s)$ *but* m ~~tioco~~ s.
Then $\exists \rho \in \mathsf{TTr}(s)$ *and* $\exists a \in (A_{out} \cup \mathbb{R}_{\geq 0})$ *s.t.* $a \in \mathsf{Out}(m \text{ after } \rho)$ *but* $a \notin \mathsf{Out}(s \text{ after } \rho)$.
But then $\rho a \in \mathsf{TTr}(m)$ *and* $\rho a \notin \mathsf{TTr}(s)$, *hence* $\mathsf{TTr}(m) \not\subseteq \mathsf{TTr}(s)$.

Q.E.D.

The input enableness assumption is important in tioco relation in order to ensure that no hidden behavior can be invoked in the implementation that is outside the specification. This restriction is too strong in practice where conformance to partial system specification is in question . If we relax the input enableness assumption then tioco relation becomes weaker than timed trace inclusion in a sense that it checks only the behavior described in the specification, thus enabling testing against partial system specifications. Alternatively [14] explores the possibility of testing with input refusal and bounded quiescence.

Later in Section 3.1 we will look further how the input enableness assumptions could be combined with assumptions about an environment, test purposes and pre-generated test cases and constrain test traces even more, which reasonably reduces the space of traces to be executed and effectively minimizes the cost of testing.

2.3 Compositional Models

Real life processes can hardly be represented by a single transition system in a comprehensive way to humans. In order to apply divide-and-conquer principle it is desirable to divide a system into several more-or-less independent components running in parallel, thus it makes sense to reason about parallel composition of two or more components. In particular, our testing framework assumes that the system is at least composed of implementation and its environment that it is embedded into. The following sections describe the semantics of composing two transition systems which result in yet another transition system which may in turn be used in another composition and then show how timed automata can be composed into networks.

2.3.1 Composition of Transition Systems

Parallel composition is a widely used operation of creating larger systems out of many smaller sub-systems. We use the composition of transition systems to ease the creation of complex systems. For example the coffee machine transition system could have been made of two processes: 1) user interface consuming the input at any time and 2) coffee brewing functionality. We also use the composition to formalize the communication between the implementation under test and its environment during the normal use and its tester during the testing phase. Definition 2.11 formally defines the composition of two TIOTSs which produces a more complex TIOTS.

Definition. 2.11 Composition *of two systems* $\mathcal{S} = TIOTS(S, s_0, A^{\mathcal{S}}_{inp}, A^{\mathcal{S}}_{out}, \rightarrow)$ *and* $\mathcal{E} = TIOTS(E, e_0, A^{\mathcal{E}}_{inp}, A^{\mathcal{E}}_{out}, \rightarrow)$ *is a system* $\mathcal{S} \| \mathcal{E} \stackrel{def}{=} TIOTS(S \times E, \langle s_0, e_0 \rangle, A_{inp}, A_{out}, \rightarrow)$:

- *Inputs:* $A_{inp} = A^{\mathcal{S}}_{inp} \cup A^{\mathcal{E}}_{inp}$,

- *Outputs:* $A_{out} = A^{\mathcal{S}}_{out} \cup A^{\mathcal{E}}_{out}$,
- *Transition relation for* $a \in (A^{\mathcal{S}}_{inp} \cap A^{\mathcal{E}}_{out}) \cup (A^{\mathcal{S}}_{out} \cap A^{\mathcal{E}}_{inp})$, $\beta \in A_{inp} \cup A_{out} \cup \{\tau\}$ *and* $d \in \mathbb{R}_{\geq 0}$*:*

$$\frac{s \xrightarrow{a} s' \quad e \xrightarrow{a} e'}{\langle s, e\rangle \xrightarrow{\tau} \langle s', e'\rangle} \qquad \frac{s \xrightarrow{\beta} s'}{\langle s, e\rangle \xrightarrow{\beta} \langle s', e\rangle} \qquad \frac{e \xrightarrow{\beta} e'}{\langle s, e\rangle \xrightarrow{\beta} \langle s, e'\rangle} \qquad \frac{s \xrightarrow{d} s' \quad e \xrightarrow{d} e'}{\langle s, e\rangle \xrightarrow{d} \langle s', e'\rangle}.$$

Intuitively, the composed system has a set of inputs (outputs) from both components which are not paired with the outputs (inputs) of the opposite component. The paired input-output component actions may become an internal action of the composed system. The inter-component action hiding is not necessary in our framework, but it is considered a realistic and clean modelling practice that components are connected pair-wise. The communication is synchronous in a sense that components cannot make an input (output) action on their own unless it is synchronized with corresponding output (input) action in another component. The delay transitions are executed synchronously in all components as time runs globally at the same rate.

A system is said to be *closed* if all input and output actions are synchronized.

2.3.2 Networks of Timed Automata

In this section we give a formal definition for parallel composition of timed automata resulting in a timed automaton.

Definition. 2.12 *An (open)* timed automata network $N_{TA} = \mathrm{TA}(L, \overline{\ell}_0, X, E, I)$ *is a timed automaton structure obtained from a parallel composition of timed automata:* $N_{TA} = (T_1 \parallel T_2 \parallel \ldots \parallel T_n)$*, where:*

- $L = \prod_{i=1}^{n} L_i$*, where* L_i *is a set of locations in* T_i
- $\overline{\ell}_0 = \langle \ell_1, \ell_2, \ldots, \ell_n \rangle$*, where* ℓ_i *is an initial location of* T_i*,*
- $X = \bigcup_{i=1}^{n} X_i$*, where* X_i *is a set of clocks in* T_i*,*
- $\langle \overline{\ell}, g, \alpha, r, \overline{\ell}' \rangle \in E$ *if either is true:*
  - $\overline{\ell}' = \overline{\ell}[\ell'_i/\ell_i]$ *and* $\langle \ell_i, g, \alpha, r, \ell'_i \rangle \in E_i$*, or*
  - $\overline{\ell}' = \overline{\ell}[\ell'_i/\ell_i, \ell'_j/\ell_j]$*,* $\langle \ell_i, g_i, a, r_i, \ell'_i \rangle \in E_i$*,* $\langle \ell_j, g_j, \overline{a}, r_j, \ell'_j \rangle \in E_j$*,* $g = g_i \wedge g_j$*,* $r = r_i \cup r_j$ *and* $\alpha = \tau$*,*
- $I(\langle \ell_1, \ell_2, \ldots, \ell_n \rangle) = \wedge_{i=1}^{n} I(\ell)$.

Note that $TIOTS(T_1 \| T_2 \| \ldots \| T_n)$ is the same as $TIOTS(T_1) \| \, TIOTS(T_2) \| \ldots \| TIOTS(T_n)$.

2.4 Symbolic Techniques

Symbolic techniques make the analysis of timed systems feasible by providing the finite partitioning of the infinite state space into symbolic zones. The idea of this symbolic technique is to group concrete states into sets of states which

can be described in a finite symbolically encoded way, then the set(s) of successor states (again encoded symbolically) can be computed by manipulating the symbolic description of the set.

In the network of timed automata case, the state is described by a location vector and a clock valuation vector. We assume that the set of locations in a timed automata network is countable and bounded, hence implying a finite number of location vector values. On the other hand, the space of clock valuations $\mathbb{R}^{|X|}_{\geq 0}$ is unbounded and uncountably large. The normalization technique [8, 9] is used to bound the values of clocks in timed automata analysis and a concept of symbolic zone is used to capture the boundaries of possible clock values instead of enumerating all concrete real-values. A *symbolic zone* represents a (potentially infinite) convex set of clock valuations bounded by constraints. Definition 2.13 defines the zone formally.

Definition. 2.13 *Let $\overline{v} \in \mathbb{R}^{|X|}_{\geq 0}$ be the automaton's valuation of clocks X and constraint system $g \in G(X)$, then a* zone *is as set of valuations satisfying constraint g: $z \stackrel{def}{=} \{\overline{v} \mid \overline{v} \models g\}$.*

For testing purposes, the most important operations on zones are defined in Definition 2.14.

Definition. 2.14 *Let $\overline{v}$ be the automaton's current valuation of clocks in X and $z, z' \subseteq \mathbb{R}^{|X|}_{\geq 0}$ be zones of X clock valuations, then the following are* zone operations*:*

$$
\begin{array}{rrcl}
\textit{Emptiness:} & z = \varnothing & \stackrel{def}{=} & \nexists \overline{v} \in \mathbb{R}^{|X|}_{\geq 0} \text{ s.t. } \overline{v} \in z \\
\textit{Containment:} & z \subseteq z' & \stackrel{def}{=} & \forall \overline{v} \in z,\ \overline{v} \in z' \\
\textit{Intersection:} & z \wedge z' & \stackrel{def}{=} & z \cap z' = \{\overline{v} \mid \overline{v} \in z \ \wedge\ \overline{v} \in z'\} \\
\textit{Reset:} & z_r & \stackrel{def}{=} & \{r(\overline{v}) \mid \overline{v} \in z\} \text{ where } r \subseteq R(X) \\
\textit{Future:} & z^{\uparrow} & \stackrel{def}{=} & \{\overline{v} + \delta \mid \overline{v} \in z,\ \delta \in \mathbb{R}_{\geq 0}\}
\end{array}
$$

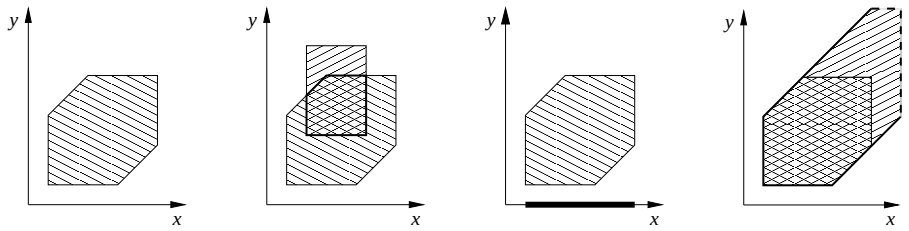
Recall Definition 2.5 where we chose to use only integers in guards and reset operations on purpose to restrict the space of timed automata whose verification problem is decidable in PSPACE (look for region construction in [1, 2]). Constraints can be extended to allow rational numbers $\mathbb{Q}$ as $|\mathbb{Q}| = |\mathbb{Z}|$ by the following method: multiply all numbers by a product of all rational number denominators found on timed automaton. This will make sure that only integer numbers are used and resulting timed automaton is equivalent to original one. However, constraints cannot be extended to contain real numbers as it would make verification undecidable.

Figure 2.1 illustrates the main operations over zones for $n = 2$ clocks as operations on a 2-dimensional polyhedra.

Definition 2.15 shows how to use zone operations to compute transitions over states symbolically.

Definition. 2.15 Symbolic transition *for timed automata network $TA(L, \overline{\ell}_0, X, E, I)$:*

$$
\frac{\gamma \in (A \cup \{\tau\}) \quad \langle \overline{\ell}, g, \gamma, r, \overline{\ell}' \rangle \in E \quad z^{\uparrow} \wedge g \neq \varnothing \quad z' = (z^{\uparrow} \wedge g)_r \wedge I(\overline{\ell}') \neq \varnothing}{\langle \overline{\ell}, z \rangle \overset{\gamma}{\rightsquigarrow} \langle \overline{\ell}', z' \rangle}
$$



(a) z (b) $z \wedge z'$ (c) $(z)_{y:=0}$ (d) $z^{\uparrow}$

Figure 2.1: Sample zone, zone intersection, clock reset and future operation.

Symbolic transition semantics corresponds closely to the timed automata semantics in Definition 2.6, in a sense that $\langle \overline{\ell}, z \rangle \overset{\gamma}{\rightsquigarrow} \langle \overline{\ell}, z' \rangle$ implies for all $\overline{v}' \in z'$, $\langle \overline{\ell}, \overline{v} \rangle \overset{\delta}{\rightarrow}\overset{\gamma}{\rightarrow} \langle \overline{\ell}, \overline{v}' \rangle$ for some $v \in z$ (cf. [10]). Then the soundness and completeness can be formulated as follows:

- Let $\langle \overline{\ell}, z \rangle \overset{\gamma}{\rightsquigarrow} \langle \overline{\ell}', z' \rangle$ be a symbolic transition, then all concrete states $\langle \overline{\ell}', \overline{v}' \rangle$ s.t. $\overline{v}' \in z'$ are reachable via $\langle \overline{\ell}, \overline{v} \rangle \overset{\delta}{\rightarrow}\overset{\gamma}{\rightarrow} \langle \overline{\ell}', \overline{v}' \rangle$ for some $v \in z$.

- Let $\langle \overline{\ell}, \overline{v} \rangle \overset{\delta}{\rightarrow}\overset{\gamma}{\rightarrow} \langle \overline{\ell}', \overline{v}' \rangle$ be any concrete computational path induced by timed automata network, with $\overline{v} \in z$ and let $\langle \overline{\ell}, z \rangle \overset{\gamma}{\rightsquigarrow} \langle \overline{\ell}, z' \rangle$ be a corresponding symbolic transition, then $\overline{v}' \in z'$.

Yi et al [64] prove the soundness and correctness of symbolic analysis.

From implementation point of view it is important that invariant and guard expressions are conjunctions of atomic expressions over clocks, hence time constraints form *convex polyhedra* without exclusion zones and only disjunction needs additional structures to capture unions (federations) of zones. UPPAAL uses difference bound matrices (DBM) [19, 23] and clock difference diagrams (CDD) [6] to carry out the above mentioned zone operations (and more, see [5] for more details).

2.4.1 Reachability Algorithm

Reachability algorithm play important role in solving model validation and system verification problems since most of them (deadlock freeness, liveness, safety properties) can be reduced to a reachability problem and posed as a search query for states satisfying a certain expression.

UPPAAL implements a typical model-checking algorithm which generates the state space of a system model via symbolic state transducer $\overset{\gamma}{\rightsquigarrow}$ and checks whether the newly generated states satisfy the given property. Algorithm 1 shows an abstract idea of forward reachability algorithm based on backward reachability algorithm presented in [44], which also provides a proof of soundness and correctness of the algorithm.

The idea behind this algorithm is to start with initial state in a waiting list (line 1), generate new states from waiting list (line 3, 7 and 8), check the property on new states (line 4) and keep track of already explored states (line 6) to detect loops (line 5). The algorithm is guaranteed to terminate assuming that the symbolic partitioning of a state space is finite and ensuring that no

Algorithm 1: An algorithm for symbolic forward reachability analysis.

Input: property P and an initial state $\langle \overline{\ell}_0, \overline{0} \rangle$ of TA network N
Result: if $N \models P$ then YES else NO

```
1  passed := {}, waiting := {⟨ℓ̄0, 0̄⟩};
2  repeat
3  |  get ⟨ℓ̄, z⟩ from waiting;
4  |  if ⟨ℓ̄, z⟩ ⊨ P then return YES;
5  |  else if ∀⟨ℓ̄, z'⟩ ∈passed z ⊈ z' then
6  |  |  add ⟨ℓ̄, z⟩ to passed;
7  |  |  for all ⟨ℓ̄, z⟩ ⇝ ⟨ℓ̄', z'⟩ do
8  |  |  |  put ⟨ℓ̄', z'⟩ to waiting;
9  until waiting = {} ;
10 return NO
```

state is visited twice by avoiding re-exploring of the old states. UPPAAL uses symbolic techniques based on DBM library [10, 23, 45, 54] at lines 4, 5 and 7.

In addition to traditional timed automata, UPPAAL supports modeling extensions: bounded integer types, arrays, safe C structure alike types, urgent and committed locations, urgent, broadcast and prioritized channel synchronizations which ease the modeling task. Unfortunately it is very easy to create models with symbolic state space too large to fit in conventional computer's operating memory. Hence to make approach still usable in practice UPPAAL also employs a number of optimization techniques: reduce the amount of symbolic states stored in memory [4, 26] at line 6 and uses discrete state hashing for checking condition at line 5 just to name a few.

2.4.2 UPPAAL Architecture

UPPAAL is a model-checker for timed automata networks extended with integer variables and C-like structures and expression updates. Figure 2.2 shows the structure of UPPAAL engine: specification parser builds data structures to represent the system model, the system representation module holds abstract syntax tree with symbol names, state space representation and manipulation module is responsible for symbolic state storage and operations on them, property parser reads the verification properties and builds a query representing expression structure which can then evaluate the given property on a symbolic state, the checker modules define high level structure of operations over symbolic states and control how the state space is explored and finally user interface provides user control over system specification and property editing, state exploration in simulation and verification and displays the system state information and verification results. UPPAAL TRON reuses the lower half of modules (except the modules related to property queries) also some parts of reachability checker are used to support UPPAAL architecture specific infrastructure.

The various checker modules are organized using pipelines of operations over symbolic states. Figure 2.3 shows how operations are connected to implement Algorithm 1:

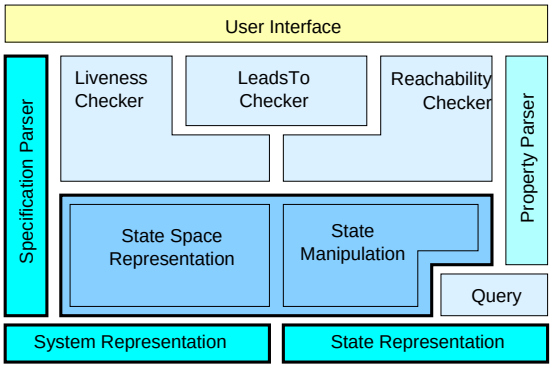


Figure 2.2: The layered architecture of UPPAAL engine [5].

1. the initial state is put into *Delay* filter where the future over a symbolic zone is computed,

2. then the symbolic state is pushed to *PassedWaitingList* which checks whether the symbolic state has already been explored and recorded in a passed list,

3. if a symbolic state is not recorded in the passed list, then it is pushed to *Query* filter which checks whether the state satisfies the property,

4. if the *Query* filter does not terminate the search, the symbolic state is pushed further to *Transition* filter which generates a list of enabled outgoing transitions,

5. afterwards a *Copy* filter prepares a fresh copy of a symbolic state for each outgoing transition,

6. *Successor* filter then computes a successor symbolic state for each transition and pushes them to the *Delay* filter.

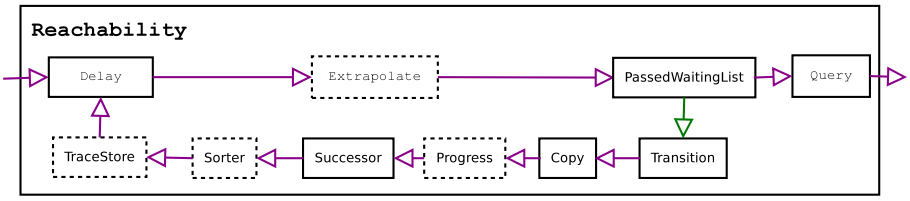


Figure 2.3: Reachability algorithm pipeline in UPPAAL [5].

Other filters are optionally included into the pipeline loop based on the user supplied settings: *Extrapolate* controls the extrapolation settings, *Progress* counts how many states per second are pushed through loop, *Sorter* controls the order of transitions, *TraceStore* stores the information needed to reconstruct the trace of symbolic states.

2.5 Discussion

We have prepared the necessary concepts to develop real-time testing theory further and assembled the ingredients to be used in building online testing tool.

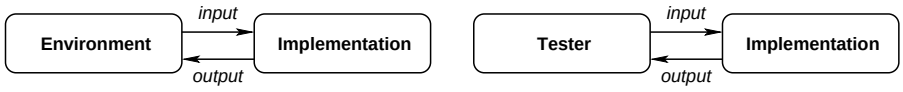
Chapter 3

Online Testing of Real-time Systems

This chapter establishes the core online testing framework at both a theoretical and a practical implementation level. Section 3.1 introduces relativized timed conformance relation as a further extension to timed ioco. Section 3.2 introduces an abstract algorithm for online test, shows its soundness and correctness at theoretical level, and exhibits the set of functions needed to implement this algorithm. Section 3.3 shows how to use symbolic operations from timed automata model-checking in order to carry out an online test. Section 3.4 concludes this chapter by showing how online testing algorithm can be organized using UPPAAL architecture.

3.1 Relativized Timed Conformance Relation

We assume that at system level our IUT is going to be deployed in a closed system, where inputs and outputs are exchanged with its environment, like it is shown in Figure 3.1a. During testing it is desirable to mimic the realistic deployment conditions as much as possible: on one hand it is desirable to test the implementation in situations that are feasible in its original environment to ensure the relevance of tests, on the other hand it is desirable to minimize the testing effort by not testing situations that are unrealistic in deployment. Therefore we propose a test setup shown in Figure 3.1b, where the tester takes a role of environment by emulating its behavior, sending only relevant inputs and checking whether the outputs are correct.



(a) During deployment. (b) During testing.

Figure 3.1: Setup of IUT.

Specifically we propose relativized timed input/output conformance relation with the following goals in mind:

1. It should define a correctness relation between IUT and its formal specification (model), preferably retain compatibility with timed I/O conformance relation.
2. It should allow test developer to specify explicit assumptions about the environment that IUT is going to be embedded during deployment.
3. It should provide direction and structure for real-time tests, facilitate optimizations in order to have better control over time and resources spent on testing.

Naturally, the tester should be equipped with a specification containing both: the assumptions about environment and requirements for implementation. We propose that the environment assumptions are modeled by $e_M \in \mathcal{E} \subseteq TIOTS(E,e,A^{\mathcal{E}}_{inp},A^{\mathcal{E}}_{out},\rightarrow^{\mathcal{E}})$, IUT requirements specification is modeled by $s \in \mathcal{S} \subseteq TIOTS(S,s,A^{\mathcal{S}}_{inp},A^{\mathcal{S}}_{out},\rightarrow^{\mathcal{S}})$, real environment is $e_R \in \mathcal{E}$ and IUT itself is $p \in \mathcal{S}$. IUT p and requirements s have the same sets of inputs and outputs and they both are compatible with environments e_M and e_R in a sense that their inputs and outputs match and we take the perspective of IUT when naming what is input and what is output: $A^{\mathcal{E}}_{out} = A^{\mathcal{S}}_{inp} = A_{inp}$ and $A^{\mathcal{E}}_{inp} = A^{\mathcal{S}}_{out} = A_{out}$.

As noted before, our ideal model of environment assumptions e_M should not differ from the real environment e_R under which p is deployed, thus $e_M = e_R = e$ and the test execution means running e composed in parallel with p. The composition of e and p forms a closed system, but the communication between them is observable (to the tester, which plays role of e) and thus it is slightly different than Definition 2.11. Definition 3.1 provides a formal meaning for composition with observable input/output actions.

Definition. 3.1 *Given two systems* $\mathcal{S} = TIOTS(S,\ s_0,\ A_{inp},\ A_{out},\ \rightarrow)$ *and* $\mathcal{E} = TIOTS(E,\ e_0,\ A_{inp},\ A_{out},\ \rightarrow)$, *an* observable composition *is a system* $\mathcal{S}\|\mathcal{E} \stackrel{def}{=} TIOTS(S \times E,\ \langle s_0, e_0\rangle,\ A_{inp},\ A_{out},\ \rightarrow)$, *where the transition relation for* $a \in (A_{inp} \cup A_{out})$ *and* $d \in \mathbb{R}_{\geq 0}$ *is defined by the following rules:*

$$\frac{s \xrightarrow{a} s' \quad e \xrightarrow{a} e'}{\langle s,e\rangle \xrightarrow{a} \langle s',e'\rangle} \qquad \frac{s \xrightarrow{\tau} s'}{\langle s,e\rangle \xrightarrow{\tau} \langle s',e\rangle} \qquad \frac{e \xrightarrow{\tau} e'}{\langle s,e\rangle \xrightarrow{\tau} \langle s,e'\rangle} \qquad \frac{s \xrightarrow{d} s' \quad e \xrightarrow{d} e'}{\langle s,e\rangle \xrightarrow{d} \langle s',e'\rangle}.$$

For clarity and simplicity reasons we require that $A_{inp} \cap A_{out} = \emptyset$ and $\mathcal{S}||\mathcal{E}$ does not participate in other compositions, i.e. the system $\mathcal{S}||\mathcal{E}$ is closed although the synchronization is observable. The operations $\mathsf{Out}()$ and after apply for observable composition in the same way like for any other TIOTS.

Definition 3.2 specifies the relation between IUT and a system specification represented by state $\langle p,e\rangle$ which is composed of IUT model state s and environment model state e.

Definition. 3.2 Relativized timed input/output conformance relation. $p, s \in \mathcal{S}$ *and* $e \in \mathcal{E}$ *are input-output compatible:*

$$p\ \mathsf{rtioco}_e\ s \quad \stackrel{def}{=} \quad \forall \sigma \in \mathsf{TTr}(e).\mathsf{Out}(\langle e,p\rangle\ \mathsf{after}\ \sigma) \subseteq \mathsf{Out}(\langle e,s\rangle\ \mathsf{after}\ \sigma) \quad (3.1)$$

Intuitively, the definition says that an IUT state p conforms to a specification state s having an environment e when for every environment trace σ the response

from IUT after exercising σ is included in the specification s after matching trace σ.

If we omit the input-enableness assumption, then the conformance relation has the following interesting cases:

1. e is not input-enabled, i.e. environment is not always able to consume what the specification or the implementation offer as an output. It means that there exists a trace $\sigma \in \mathsf{TTr}(e)$ such that $\sigma o \notin \mathsf{TTr}(e)$ but $\sigma o \in \mathsf{TTr}(p)$ and $\sigma o \in \mathsf{TTr}(s)$, where $o \in A_{out}$. Then $o \notin \mathsf{Out}(\langle p, e\rangle \text{ after } \sigma)$ and $o \notin \mathsf{Out}(\langle s, e\rangle \text{ after } \sigma o)$. This means that the conformance relation still holds (no illegal behavior has been observed), but the tester cannot continue the σ test run (i.e. appending o to σ) as $\sigma o \notin \mathsf{TTr}(e)$ and in practice, the tester should issue verdict inconclusive.

2. p refuses an input at the same time as s refuses the input, i.e. there exists a trace $\sigma i \in \mathsf{TTr}(e)$ such that $\sigma \in \mathsf{TTr}(p)$ but $\sigma i \notin \mathsf{TTr}(p)$ and $\sigma i \notin \mathsf{TTr}(s)$, where $i \in A_{inp}$. Then $\langle p, e\rangle \text{ after } \sigma i = \emptyset$ and $\langle s, e\rangle \text{ after } \sigma i = \emptyset$, then $\mathsf{Out}(\emptyset) = \emptyset$ and the conformance relation still holds as $\emptyset \subseteq \emptyset$ is true. On the other hand, it does not make sense to continue the test run as all resulting traces with prefix σi will have the same result. Here rtioco agrees with tioco with respect to correctness.

3. p refuses input but s is able to consume it, i.e. there exists a trace $\sigma i \in \mathsf{TTr}(e)$ such that $\sigma \in \mathsf{TTr}(p)$ but $\sigma i \notin \mathsf{TTr}(p)$ and $\sigma i \in \mathsf{TTr}(s)$, where $i \in A_{inp}$. Then $\langle p, e\rangle \text{ after } \sigma i = \emptyset$, $\mathsf{Out}(\langle p, e\rangle \text{ after } \sigma i) = \emptyset$ and conformance relation holds no matter how s behaves further. The result is the same as with tioco .

4. s refuses input, but p accepts the input, i.e. there exists a trace $\sigma i \in \mathsf{TTr}(e)$ such that $\sigma i \in \mathsf{TTr}(p)$ but $\sigma i \notin \mathsf{TTr}(s)$, then $\langle p, e\rangle$ has a successor state after trace σi: $0 \in \mathsf{Out}(\langle p, e\rangle \text{ after } \sigma i \neq \emptyset)$, whereas $\mathsf{Out}(\langle s, e\rangle \text{ after } \sigma i) = \emptyset$ and $0 \notin \emptyset$. So in this case rtioco is more powerful than tioco in a sense that the latter does not allow testing the traces outside $\mathsf{TTr}(s)$ in the first place, hence they would not be tested at all. The only correct response from p in this case would be to refuse to accept the input i.

5. p and s are both at least weakly input-enabled. Then the correctness depends on the relation between p and s within e. In an extreme case with fully permissive environment e_U we have $\mathsf{TTr}(e_U) = (\mathbb{R}_{\geq 0} \cup A_{inp} \cup A_{out})^*$ and p rtioco_{e_U} $s = p$ tioco s since inputs can be refused by neither p and s and outputs together with delays are always checked before appended to a trace prefix. The only difference is that tioco does not challenge the delays outside s, while rtioco_{e_U} would try all possible delays even if the further trace does not reveal any new information with regards to conformance. Such intimate treatment of s in tioco could be seen as an optimization to generate traces only relevant to s, but it actually puts the tester into weaker position to avoid testing delays extremely close to maximum allowed delay. Consider a specification with a deadline for output: specification can delay up to deadline without issuing output action or consume input offered by the tester, IUT simply delays and refuses to output anything, when the deadline approaches the tester has a

choice to offer an input or detect a deadlock, if the system is in a deadlock situation then there is no way of knowing if it was IUT failure to deliver output action before deadline or it was the tester's fault not to deliver input before deadline. In rtioco this unnecessary stress is removed by the model of environment which serves as a guide to trace generation and at the same time helps to determine diagnostic information.

For passive monitoring purposes one can also compose a model of the environment which does not allow any inputs to be offered (hence no test generation needed) but accepts any outputs that the IUT can produce.

As a result, a test engineer can achieve model the environment under various assumptions ranging from a concrete to abstract over-approximations and still specify exhaustive testing as an option (easy to specify but expensive to execute).

Theorem. 3.1 *Let p, s and e be input enabled systems, then relativized timed I/O conformance relation coincides with timed trace inclusion:*

$$p \ \mathsf{rtioco}_e \ s \ \Leftrightarrow \ \mathsf{TTr}(p) \cap \mathsf{TTr}(e) \subseteq \mathsf{TTr}(s) \cap \mathsf{TTr}(e) \tag{3.2}$$

Proof.

$\Rightarrow$. *Assume* $p \ \mathsf{rtioco}_e \ s$ *but* $\mathsf{TTr}(p) \cap \mathsf{TTr}(e) \not\subseteq \mathsf{TTr}(s) \cap \mathsf{TTr}(e)$.
Then for some $\sigma \in \mathsf{TTr}(p) \cap \mathsf{TTr}(e)$ *but* $\sigma \notin \mathsf{TTr}(s) \cap \mathsf{TTr}(e)$. *Thus* $\sigma \notin \mathsf{TTr}(s)$.
Let σ *be a trace with minimal length,* $\sigma \neq \varepsilon$.
$\sigma = \sigma'\gamma$, *where* $\gamma \in A \cup \mathbb{R}_{\geq 0}$. *Then* $\sigma' \in \mathsf{TTr}(p) \cap \mathsf{TTr}(e)$, $\sigma' \in \mathsf{TTr}(s)$:

1. $\gamma \in A_{inp}$. $e \stackrel{\gamma}{\Rightarrow}$ *but* $s \stackrel{\gamma}{\not\Rightarrow}$, *however* s *is input enabled. Contradiction.*
2. $\gamma \in A_{out} \cup \mathbb{R}_{\geq 0}$. $\gamma \in \mathsf{Out}(s \ \mathsf{after} \ \sigma') \Leftrightarrow \sigma'\gamma \in \mathsf{TTr}(s)$.

$\Leftarrow$. *Assume* $\mathsf{TTr}(p) \cap \mathsf{TTr}(e) \subseteq \mathsf{TTr}(s) \cap \mathsf{TTr}(e)$ *but* $p \ \mathsf{rtioco}_e \ s$.
Then $\exists \sigma \in \mathsf{TTr}(e). \exists o \in \mathsf{Out}(\langle p, e\rangle \ \mathsf{after} \ \sigma)$ *(*$\sigma o \in \mathsf{TTr}(\langle p, e\rangle)$*), but* $o \notin \mathsf{Out}(\langle s, e\rangle \ \mathsf{after} \ \sigma)$ *(*$\sigma o \notin \mathsf{TTr}(\langle s, e\rangle)$*).*
The we know that $\sigma o \in \mathsf{TTr}(e)$ *and* $\sigma o \in \mathsf{TTr}(p)$, *but* $\sigma o \notin \mathsf{TTr}(\langle s, e\rangle) = \mathsf{TTr}(s) \cap \mathsf{TTr}(e)$. *Contradiction.*

If s is not input enabled for some input in some state but p is, then there is a trace ρ such that $\mathsf{Out}(\langle e, s\rangle \ \mathsf{after} \ \rho) = \emptyset$ but $\mathsf{Out}(\langle e, p\rangle \ \mathsf{after} \ \rho) \neq \emptyset$ therefore $p \ \mathsf{rtioco}_{e_U} \ s$ does not hold and test fails. This way tester can discover hidden functionality within p that is not accessible and not defined by s, such functionality cannot be detected by tioco or ioco .

3.2 Abstract Online Testing

The goal of testing is to establish the correctness relation between a system model and an IUT. The goal of online test is to produce test inputs and adopt to test execution while the test is being executed and evaluated. Online testing avoids generating full test (suite) in advance in favor of saving time and memory while dealing only with a limited scope of a current system state estimate.

Avoiding full test generation is important for non-deterministic systems, where tests take form of a decision tree rather than an event sequence as typically dealt by offline tests. Timed specifications, remote and black-box systems are inherently non-deterministic because of:

- Concurrent processes in the system whose order of execution is unspecified or arbitrary. In addition, the input/output communication is typically done through concurrent buffers.
- Internal transitions in a black-box system may fire at non-deterministic times or not fire at all and hence are not visible from outside.
- Execution time uncertainties due to complex caches in processor cause input/output behavior to be unpredictable.
- Non-determinism is used as a means of abstraction over requirements allowing several possible implementations or hidden or unknown behavior..

Online test combines several testing activities executed at the same time:

- Generation of test primitives (inputs, expected outputs and their timings) by analyzing the system specification.
- Execution (and execution recording) of test primitives by using test adapter to translate abstract input description into physical input actions and recording physical output event by translating them into an abstract output description.
- Evaluation of a test assigns a verdict *pass* or *fail* to an observed test trace by analyzing a system specification.

From an engineering point of view, test generation combined with test execution can be viewed as an environment emulation as the tester plays role of an environment when deciding what input to offer. Test evaluation becomes monitoring as the tester is concerned only evaluating the correctness of IUT. At the same time, test generation and evaluation are concerned with specification analysis and are very similar: one is searching for relevant inputs and the other is checking that an observed output is a possible (allowed) output. Hence it is natural to use model-checking techniques to analyze specifications and share and reuse the specification analysis effort between generation and evaluation activities.

Monitoring determines whether the observed behavior is correct or not according to specification. Using the rtioco relation, monitoring evaluates whether the observed output can be matched by the specification, this in turn requires knowledge of the current system state which is not directly observable in a black-box setting. Moreover, a correct environment emulation also requires some bookkeeping about the current (possible) state of the environment.

Section 3.2.1 presents the state estimation functions needed to compute and update a set of system states possibly occupied by a closed system and how to compute relevant inputs when the system state is known. Section 3.2.2 shows how to combine the state estimation functions and to achieve an abstract algorithm for online test. Section 3.2.3 elaborates on the soundness and completeness of an online test algorithm.

3.2.1 State Set Estimation and Input Choice

This section defines the necessary functions to be used in online test algorithm. Let $\mathcal{S}||\mathcal{E}$ be a specification system and S be a set of current system states. First, we define the state estimation function S after σ which capture the set of possible states a system may occupy after a given observable action sequence σ assuming that it starts with of the states from S. Then, we define the sets of possible actions for test continuation after the given current state estimate.

Definition. 3.3 *State set update function after observable action transition or delay* $\sigma \in (A \cup \mathbb{R}_{\geq 0})^*$*:*

$$S \text{ after } \sigma \stackrel{def}{=} \{ \langle s', e' \rangle \mid \langle s, e \rangle \in S. \langle s, e \rangle \stackrel{\sigma}{\Rightarrow} \langle s', e' \rangle \} \tag{3.3}$$

Definition. 3.4 *Possible input actions (stimuli from environment), delays and allowed output actions (possible responses from IUT):*

$$\mathsf{EnvOutput}(S) \stackrel{def}{=} \{ a \in A_{inp} \mid \langle s, e \rangle \in S. e \stackrel{a}{\rightarrow} \} \tag{3.4}$$

$$\mathsf{Delays}(S) \stackrel{def}{=} \{ d \mid \langle s, e \rangle \in S. e \stackrel{d}{\Rightarrow} \} \tag{3.5}$$

$$\mathsf{ImpOutput}(S) \stackrel{def}{=} \{ a \in A_{out} \mid \langle s, e \rangle \in S. s \stackrel{a}{\rightarrow} \} \tag{3.6}$$

3.2.2 Online Test Algorithm

Algorithm 2 outlines an online test procedure which performs test generation, execution and IUT monitoring at the same time by operating on concrete states.

Algorithm 2: Test generation and execution, $\mathit{OnlineTest}(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T)$.

```
1  S := {⟨s0, e0⟩};                    // let the set contain an initial state
2  while S ≠ ∅ ∧ ♯iterations ≤ T do
3      switch Random({action, delay, restart}) do
4          case action                                   // offer an input
5              if EnvOutput(S) ≠ ∅ then
6                  randomly choose i ∈ EnvOutput(S);
7                  send i to IUT, S := S after i;
8          case delay                                    // wait for an output
9              randomly choose d ∈ Delays(S);
10             sleep for d time units or wake up on output o at d' ≤ d;
11             if o occurs then
12                 S := S after d';
13                 if o ∉ ImpOutput(S) then return fail;
14                 else S := S after o
15             else S := S after d;                      // no output within d delay
16         case restart                                  // reset and restart
17             S := {⟨s0, e0⟩};
18             reset IUT
19 if S = ∅ then return fail else return pass
```

3.2.3 Soundness and Completeness

This section provide expanded version of the result published in [42].

First, we briefly revisit the concept of digitization from [59]. Consider an event time-stamped trace $\rho = (e_0, t_0), (e_1, t_1), (e_2, t_2) \ldots$, where $e_i \in A$, $t_i \in \mathbb{R}_{\geq 0}$ and $t_i \leq t_{i+1}$ for all $i \in \mathbb{N}$. We obtain the observation sequence $[\rho]_\epsilon = (e_0, [t_0]_\epsilon), (e_1, [t_1]_\epsilon), (e_2, [t_2]_\epsilon) \ldots$, where $[t]_\epsilon$ is a rounding with respect to ϵ: $[t]_\epsilon = \lfloor t \rfloor$ if $t \leq \lfloor t \rfloor + \epsilon$, otherwise $[t]_\epsilon = \lceil t \rceil$. Then the *digitization* of traces Π is a set of integral traces containing all digitizations:

$$[\Pi] = \{[\rho]_\epsilon \mid \rho \in \Pi,\ 0 \leq \epsilon < 1\}$$

The timed traces Π are said to be *closed under digitization* if $\rho \in \Pi$ implies $[\rho] \subseteq \Pi$. The timed traces Π are said to be *closed under inverse digitization* if $[\rho] \subseteq \Pi$ implies $\rho \in \Pi$. The set of traces Π is said to be *digitizable* when

$$\rho \in \Pi \quad \text{iff} \quad [\rho] \in \Pi$$

Algorithm 2 depicts our randomized algorithm for providing stimuli to (in terms of input and delays) and observing the resulting reactions (in terms of output) from a given IUT. Assuming that the behavior of the IUT is a non-blocking, input enabled, deterministic TIOTS with isolated outputs the reaction to any given timed input trace $\sigma = d_1 i_1 \ldots d_k i_k d_{i+1}$ is completely deterministic. More precisely, given the stimuli σ there is a unique $\rho \in \mathsf{TTr}(\mathsf{IUT})$ such that $\rho \uparrow A_{inp} = \sigma$, where $\rho \uparrow A_{inp}$ is the natural projection of the timed trace ρ to the set of input actions.

Under a certain (theoretically necessary) testing hypothesis about the behavior of IUT and given that the TIOTSs $\mathcal{S}$ and $\mathcal{E}$ satisfy certain assumptions, the randomization used in Algorithm 2 may be chosen such that the algorithm is both complete and sound in the sense that it (eventually with probability one) gives the verdict "fail" in all cases of non-conformance and the verdict "pass" in cases of conformance. The hypothesis and assumptions are based on the results on digitization techniques in [59] which allow the dense-time trace inclusion problem between two sets of timed traces to be reduced to discrete time. In particular it suffices to choose unit delays in Algorithm 2 (assuming that the models and IUT share the same magnitude of a time unit).

Theorem. 3.2 *Assume that the behavior of IUT may be modelled as an input enabled, non-blocking, deterministic TIOTS with isolated outputs, $\mathsf{TTr}(\mathsf{IUT})$ and $\mathsf{TTr}(\mathcal{E})$ are* closed under digitization *and that $\mathsf{TTr}(\mathcal{S})$ is* closed under inverse digitization. *Algorithm 2 is then sound with only unit delays and complete in the following senses:*

1. *Whenever $OnlineTest(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T) = fail$ then IUT $\not\mathsf{rtioco}_{\mathcal{E}}$ $\mathcal{S}$.*
2. *Whenever IUT $\not\mathsf{rtioco}_{\mathcal{E}}$ $\mathcal{S}$ then:*

$$\mathsf{Prob}\big(OnlineTest(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T) = fail\big) \xrightarrow{T \to \infty} 1$$

where T is the maximum number of iterations of the while-loop before exiting.

Proof. *(Sketch) Soundness follows from an easy induction on $|\rho|$ that when starting each iteration of the while-loop the timed trace ρ observed since the last restart satisfies $\rho \in \mathsf{TTr}(\mathit{IUT})$, $\rho \in \mathsf{TTr}(\mathcal{E})$ and $\rho \in \mathsf{TTr}(\mathcal{S})$ and that any chosen extension $\rho\alpha$ still lies in $\mathsf{TTr}(\mathit{IUT}) \cap \mathsf{TTr}(\mathcal{E})$.*

As for completeness, assume that the IUT does not conform to $\mathcal{S}$ relative to $\mathcal{E}$. Then $\mathsf{TTr}(\mathit{IUT}) \cap \mathsf{TTr}(\mathcal{E}) \not\subseteq \mathsf{TTr}(\mathcal{S})$. However due to the assumed properties of closure with respect to digitization respectively inverse digitization this failing timed trace inclusion is equivalent to the existence of a timed trace $\rho = d_1 a_1 d_2 a_2 \ldots d_k a_k d_{k+1}$ with all delays being integral such that $\rho \in \mathsf{TTr}(\mathit{IUT}) \cap \mathsf{TTr}(\mathcal{E})$ but $\rho \notin \mathsf{TTr}(\mathcal{S})$. Now let $\sigma = \rho \uparrow A_{inp}$; that is σ is the input-delay stimuli allowed by $\mathcal{E}$ which when given to IUT will result in the timed trace ρ. Now assume that the random choice of input action, unit delay and restart is made using a fixed discrete and finite probability distribution (with p being the smallest probability used) it is clear that:

$$\mathsf{Prob}(\sigma \text{ is generated between two given consecutive restarts }) \geq p^{K+D}$$

where K respectively D is the number of input actions respectively accumulated delay in σ. Now let $\epsilon = p^{K+D}$ it follows that

$$\mathsf{Prob}(\sigma \text{ is generated before k'th restart }) \geq 1 - (1-\epsilon)^{k-1}$$

Obviously there will in general be several input stimuli that will reveal the lack of conformance. Hence the above probability just provides a lower bound for Algorithm 2 yielding the verdict "fail" before the k'th restart. The number of restarts diverges as $T \to \infty$ and hence we see that $\mathsf{Prob}(\sigma \text{ is generated}) = 1$. *Q.E.D.*

Theorem 3.2 assumes that the IUT can be modelled by a formal object in a class of TIOTS. The assumption is commonly referred to as the *test hypothesis*. In this case, only its theoretical existence is assumed, and a precise instance can be unknown. In particular, it may be extremely large and detailed, and most importantly it can be structurally totally unrelated to the specification.

From [35, 59] it follows that the closure properties required in Theorem 3.2 are satisfied if the behavior of IUT and $\mathcal{E}$ are TIOTSs induced by timed automata with closed constraints (i.e. where all guards and invariants are non-strict) and $\mathcal{S}$ is a TIOTS induced by an open timed automaton (i.e. with guards and invariants being strict). In practice these requirements are not restrictive, e.g. for strict guards one can always scale the clock constants to obtain arbitrary high precision.

Note that, the assumptions about determinism and IUT structure are important for theoretical completeness (exhaustive testing). Exhaustive testing for real-time systems means exercising all possible timings with high granularity which often is impractical, thus the completeness result just shows the theoretical rigor of the method.

3.3 Symbolic Techniques for Online Testing

In this section we show how to use symbolic techniques to implement Algorithm 2. We consider timed automata network as closed system containing implementation requirements and environment assumptions. Measuring the exact time instant of an event is unrealistic due to practical and theoretical reasons. Thus we prefer to describe the timing of a real world (I/O) event by an interval of time. We introduce interval time-stamps in event traces and then interval delay operations for symbolic zones and adopt new rules for symbolic transitions. The result is an implementable algorithm operating on digitized time-stamps using intervals and symbolic states encoding the concrete state estimate. The concrete real-valued timed trace from Algorithm 2 can be seen as special case where the lower bound and upper bound of interval time-stamp coincide, except that the new algorithm applies over-approximation by using most narrow integer interval to describe each instant.

3.3.1 Event Time-Stamping

Definition 3.5 assumes that it is possible to describe a test event by an input/output action and absolute time interval when the action actually happened. The events are then grouped into sequences forming event traces capturing the observable history of an online test.

Definition. 3.5 *Test events and test event traces:*

- *A test event is an observable action with associated time interval denoting the absolute time reference when the event (could have) happened, denoted by* $e = (\!|t, t'|\!)a$ *where* $a \in A$, $t, t' \in \mathbb{N}$ *and* $t \leq t'$. *Set of events is denoted by* $\mathsf{Events} \subset \mathbb{N} \times \mathbb{N} \times A$.

- *Test event trace* $\omega = e_1 e_2 \dots e_n$ *is a sequence of events with monotonically increasing intervals:* $\forall i \in [1, n]$: $e_i = (\!|t_i, t'_i|\!)a_{j_i}$, $a_{j_i} \in A$ *and* $t'_{i-1} \leq t_i$.

Here we stick to using only positive integers including zero symbolic traces. This restricts the precision with which events can be recorded. It can be shown that it is possible to achieve any rational number precision using the constraint scaling techniques from [3]. However the precision has to be fixed in advance before starting the online test. Hence we use positive integers for simplicity.

3.3.2 State Estimation

The symbolic transition relation for UPPAAL timed automata (described in Section 2.4) are designed for reachability Algorithm 1 and perform any and all delays possible within constraints of a model. In our testing framework the goal is to map the actual events and concrete delays into the model state space. Therefore a slightly different transition relation is needed, which has a better control over them without resorting to a complete discretization of time, but instead take the advantage of the symbolic model-checker engine.

We propose a new operation for delays over clock valuation zones that allows us to match concrete delays with absolute time reference and with arbitrary (interval) precision on the symbolic zone. The delay is referenced by absolute

time values between t and t' boundaries ($t \leq t'$). The interval boundary strictness (openness) is specified in parenthesis and then reflected in corresponding constraints:

$$z^{\uparrow(\!|t,t'|\!)} \stackrel{\text{def}}{=} z^{\uparrow} \wedge (t \prec_1 \chi) \wedge (\chi \prec_2 t') \quad \left\{ \begin{array}{lcl} z^{\uparrow(t,t')} & \stackrel{\text{def}}{=} & z^{\uparrow} \wedge (t < \chi) \wedge (\chi < t') \\ z^{\uparrow[t,t')} & \stackrel{\text{def}}{=} & z^{\uparrow} \wedge (t \leq \chi) \wedge (\chi < t') \\ z^{\uparrow(t,t']} & \stackrel{\text{def}}{=} & z^{\uparrow} \wedge (t < \chi) \wedge (\chi \leq t') \\ z^{\uparrow[t,t']} & \stackrel{\text{def}}{=} & z^{\uparrow} \wedge (t \leq \chi) \wedge (\chi \leq t') \end{array} \right. \tag{3.7}$$

where the inequality sign $\prec_1$ matches the left parenthesis and inequality sign $\prec_2$ matches the right parenthesis. It is assumed that the zone contains an external clock χ for global time and thus is never reset.

The symbolic transition over symbolic states from Definition 2.15 is modified to handle action and delay transitions separately. The result is the two rules outlined in Definition 3.6.

Definition. 3.6 *Symbolic transitions for testing:*

- *Action γ transition:*

$$\frac{\gamma \in (A \cup \{\tau\}) \quad \langle \overline{\ell}, g, \gamma, r, \overline{\ell}' \rangle \in E \quad z \wedge g \neq \varnothing \quad z' = (z \wedge g)_r \wedge I(\overline{\ell}') \neq \varnothing}{\langle \overline{\ell}, z \rangle \stackrel{\gamma}{\rightarrowtail} \langle \overline{\ell}', z' \rangle}$$

- *Delay transition by a non-empty interval $(\!|t,t'|\!)$:*

$$\frac{t, t' \in \mathbb{N} \quad t \leq t' \quad z' = z^{\uparrow(\!|t,t'|\!)} \wedge I(\overline{\ell}) \neq \varnothing}{\langle \overline{\ell}, z \rangle \stackrel{(\!|t,t'|\!)}{\rightarrowtail} \langle \overline{\ell}, z' \rangle}$$

The first rule is similar to an edge transition specified in Definition 2.15 except that we do not let the time pass by omitting the future operator. Thus the symbolic transition is taken along the γ-action edge considering all possible clock values described by zones z and z'. The second rule allows a delay to an absolute time reference from the moment t and until the moment t'.

The notation for symbolic event traces on symbolic states is and extension to notation from Definition 3.7 in such a way that ω event trace with interval time stamps is a digitization of corresponding concrete trace σ. The after operation gives an estimate of the reachable symbolic states after an event or a sequence of events observed.

Definition. 3.7 Symbolic notation. *For symbolic states $\langle \bar{l}, z \rangle$ and $\langle \bar{l}', z' \rangle$, set of symbolic states $\mathcal{Z}$, action $a \in A$, events $e = (\!|t,t'|\!)a \in$ Events, $e_{1,2\ldots n} \in$ Events, event trace $\omega = e_1 e_2 \ldots e_n$:*

$$\langle \overline{\ell}, z\rangle \overset{(\!|t,t'|\!)}{\Longrightarrow} \langle \overline{\ell}', z'\rangle \overset{def}{=} \exists \langle \overline{\ell}_1, z_1\rangle, \langle \overline{\ell}_2, z_2\rangle : \langle \overline{\ell}, z\rangle \overset{[0,t']\!)}{\rightarrowtail} \langle \overline{\ell}_1, z_1\rangle \overset{\tau}{\rightarrowtail}{}^{*} \langle \overline{\ell}_2, z_2\rangle \overset{(\!|t,t'|\!)}{\rightarrowtail} \langle \overline{\ell}', z'\rangle;$$

$$\langle \overline{\ell}, z\rangle \overset{a}{\Longrightarrow} \langle \overline{\ell}', z'\rangle \overset{def}{=} \exists \langle \overline{\ell}_1, z_1\rangle, \langle \overline{\ell}_2, z_2\rangle : \langle \overline{\ell}, z\rangle \overset{\tau}{\rightarrowtail}{}^{*} \langle \overline{\ell}_1, z_1\rangle \overset{a}{\rightarrowtail} \langle \overline{\ell}_2, z_2\rangle \overset{\tau}{\rightarrowtail}{}^{*} \langle \overline{\ell}', z'\rangle;$$

$$\langle \overline{\ell}, z\rangle \overset{e}{\Longrightarrow} \langle \overline{\ell}', z'\rangle \overset{def}{=} e = (\!|t,t'|\!)a, \exists \langle \overline{\ell}'', z''\rangle : \langle \overline{\ell}, z\rangle \overset{(\!|t,t'|\!)}{\Longrightarrow} \langle \overline{\ell}'', z''\rangle \overset{a}{\Longrightarrow} \langle \overline{\ell}', z'\rangle;$$

$$\langle \overline{\ell}, z\rangle \overset{\omega}{\Longrightarrow} \langle \overline{\ell}', z'\rangle \overset{def}{=} \exists \langle \overline{\ell}_i, z_i\rangle\ \forall i \in 0 \ldots n : \langle \overline{\ell}_0, z_0\rangle = \langle \overline{\ell}, z\rangle, \langle \overline{\ell}_n, z_n\rangle = \langle \overline{\ell}', z'\rangle \text{ and } \langle \overline{\ell}_{i-1}, z_{i-1}\rangle \overset{e_i}{\Longrightarrow} \langle \overline{\ell}_i, z_i\rangle;$$

$$\mathcal{Z} \text{ after } (\!|t,t'|\!) \overset{def}{=} \{\langle \overline{\ell}', z'\rangle \mid \exists \langle \overline{\ell}, z\rangle \in \mathcal{Z},\ \langle \overline{\ell}, z\rangle \overset{(\!|t,t'|\!)}{\Longrightarrow} \langle \overline{\ell}', z'\rangle\};$$

$$\mathcal{Z} \text{ after } a \overset{def}{=} \{\langle \overline{\ell}', z'\rangle \mid \exists \langle \overline{\ell}, z\rangle \in \mathcal{Z},\ \langle \overline{\ell}, z\rangle \overset{a}{\Longrightarrow} \langle \overline{\ell}', z'\rangle\};$$

$$\mathcal{Z} \text{ after } e \overset{def}{=} \{\langle \overline{\ell}', z'\rangle \mid \exists \langle \overline{\ell}, z\rangle \in \mathcal{Z},\ \langle \overline{\ell}, z\rangle \overset{e}{\Longrightarrow} \langle \overline{\ell}', z'\rangle\};$$

$$\mathcal{Z} \text{ after } \omega \overset{def}{=} \{\langle \overline{\ell}', z'\rangle \mid \exists \langle \overline{\ell}, z\rangle \in \mathcal{Z},\ \langle \overline{\ell}, z\rangle \overset{\omega}{\Longrightarrow} \langle \overline{\ell}', z'\rangle\};$$

Remarks. The symbolic techniques always result in a finite symbolic state estimated as follows:

- Internally, even Zeno traces are allowed in the specification:
  - events that are close in time may match the same integer interval, and events within the same integer interval are treated equivalently (as in regular UPPAAL symbolic techniques);
  - since the specification is finite, the infinite sequence of internal transitions can be modelled only in a loop structure;
  - during a bounded time interval a system can perform a unbounded number of (internal) actions by taking infinite number of loop iterations;
  - loops without progress (resulting in equal symbolic states) are detected by purging equal symbolic states giving just one finite symbolic state sequence as representative for infinite loop.
- Observably, realistic test traces are finite in length and contain finitely many test events hence result in finite number of operations on symbolic states which lead to finite number of states.

3.3.3 Mapping World Time and Model Time

This section explains the approach of obtaining symbolic event traces from concrete and discusses its correctness.

The following is the general formula for mapping the digital clock values to model time. The earliest event timestamp is at t the latest is at t', the model time unit is of duration T, and tester's clock resolution is r:

$$\mathsf{RM}(t,t') \overset{\text{def}}{=} \left(\!\left|\left\lfloor \frac{t}{T} \right\rfloor, \left\lceil \frac{t'+r}{T} \right\rceil\right|\!\right) \quad \begin{cases} \text{“}(\!|\text{” is “}[\text{” if } T|t \\ \text{and “(” otherwise,} \end{cases} \tag{3.8}$$

Here we assume that the tester's clock runs at discrete time intervals with ticks of period r and the tester can read its value just before an event (value t) and

just after the event (value t'). We add the clock resolution delay r to the second time stamp because the second time stamp is measured after a "tick" (discrete clock value update) and before the next "tick" which means it can be anywhere in between, i.e. the second measurement happens between t' and $t'+r$, perhaps exactly at t', but strictly before $t'+r$ because the clock did not show the next tick value $t'+r$ yet. Naturally, the ideal real-valued tester's clock has resolution of $r = 0$ and in ideal measuring and event triggering conditions the tester would observe $t = t'$.

The lower bound can be non-strict only if the lower time-stamp coincides with a model time integer value exactly, i.e. the event happens when the real clock tick coincides with model time tick (integer value).

Observe that the upper bound is always strict and can never be non-strict: even if the upper time-stamp (with resolution r added) coincides with a model time integer, we still know that the event happened *before* the next tick, otherwise the upper time-stamp would contain the value of that next tick.

Example. Figure 3.2 shows three time-lines: the tester's physical real-valued time, the digital clock used to sample the time and model time. The events are time-stamped in the following way:

Input is time-stamped by digital clock values $t = t_7$ and $t' = t_{10}$, hence in model time it happened at $(t_{12}, t_{13}) = (\frac{t_7}{T}, \frac{t_{10}+r}{T})$.

Output is time-stamped by digital clock value $t = t_2 = t'$, hence in model time it happened at $(t_4, t_5) = (\frac{t_2}{T}, \frac{t_2+r}{T})$.

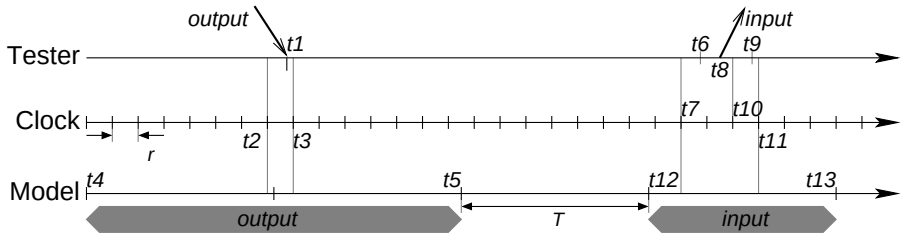


Figure 3.2: Conversion of digital clock time-stamps to model time units.

Example. Assume that tester's clock runs with a resolution $r = 10ms$, the model time unit is $T = 100ms$ and the test started at absolute time $0ms$. The tester needs to send an input action a: just before sending the input, tester looks up the clock and measures $t = 10080ms$, sends the input and immediately measures $t' = 10110ms$. This input is recorded as an event $[10080ms, 10110ms]a$ and is converted to event in the model state space as $e = (100, 102)a$. Similarly:

$$\mathsf{RM}(10000ms, 10050ms) = [100, 101), \quad \mathsf{RM}(10050ms, 10090ms) = (100, 101).$$

We forth discuss the correctness and the precision provided by our approach.

In a trivial case, consider the concrete trace σ from Definition 2.2 composed of integer delays. In this case, the conversion of concrete trace σ to symbolic event trace ω is a trivial conversion of relative delays to absolute time intervals containing just one time-stamp value for each event ($t = t'$ in the event interval timestamp $[t, t']$). Then it is easy to see that the computed symbolic state set equals the the possible reachable concrete state set: S_0 after $\sigma = \mathcal{Z}_0$ after ω.

In cases with real-valued delays, an over-approximation is used to map any real value to the nearest integer interval. Figure 3.2 shows an example how input and output events are time-stamped using digital clock and mapped on to the model time axis: the output arrives at t_1 and tester observes clock value t_2, therefore concludes that output happened between t_2 and t_3 which maps to the interval (t_4, t_5) in model time. Before sending input, the tester looks up the clock at t_6 and observes the value t_7, sends the input at t_8 and at t_9 looks up the clock again and observes value t_{10}. Thus tester concludes that input happened between t_7 and t_{11} which maps to the model time interval (t_{12}, t_{13}).

Given this mapping, we can now calculate the state estimate for any concrete trace, potentially containing real-valued delays, using this over-approximation. Such over-approximation includes the behavior that never actually happened which becomes indistinguishable from the observed behavior. In other words it leads to a loss of precision, but from correctness point of view such loss is acceptable since it can only produce false test pass verdict and never false test fail, i.e. the tool is less sensitive to faults than ideal implementation based on real-value delays. Again for practical purposes, the precision can be made arbitrarily small (if executing hardware allows) using the smaller time unit and getting a more precise tester's clock.

The method still relies on the assumption the tester's clock drift is negligible or the clock treated as an ultimate reference clock (real-time aspect is as good as this clock).

3.3.4 Test Derivation

Test derivation consists of calculating possible inputs and delays and making a choice on which input to send and how much to delay. The previous section provided us with the symbolic techniques necessary to estimate the current state and here we use this information to derive what further events are possible and when. The Events function in Definition 3.8 computes a set of actions enabled in the model from a given symbolic state set $\mathcal{Z}$. The function is parameterized with a set of actions A which can be either A_{inp} or A_{out}.

Definition. 3.8 $\mathsf{Events}(\mathcal{Z}, A)$ *computes a set of possible events with action labels* A*:*

$$\mathsf{Events}(\mathcal{Z}, A) \stackrel{def}{=} \left\{ (\!|m, M|\!)a \;\middle|\; \begin{array}{l} \forall \langle \overline{\ell}, z \rangle \in \mathcal{Z}\ \exists a \in A, \textit{ such that } \langle \overline{\ell}, z \rangle \stackrel{a}{\rightarrowtail} \langle \overline{\ell}', z' \rangle \\ (\!|m, M|\!) = (\min(z'|_{\mathcal{X}}), \max(z'|_{\mathcal{X}})) \end{array} \right\} \tag{3.9}$$

where $z|_{\mathcal{X}}$ *is a zone* z *projection to clock* $\mathcal{X}$ *giving the value solution set for clock* $\mathcal{X}$, $\min(\cdot)$ *and* $\max(\cdot)$ *are functions returning the minimum and the maximum respectively of the argument set.*

The set of enabled inputs can be computed using $\mathsf{Events}(\mathcal{Z}, A_{inp})$ and similarly possible output events are $\mathsf{Events}(\mathcal{Z}, A_{out})$. The Events function corresponds to EnvOutput and ImpOutput operator from Definition 3.4. The Events function also gives information about the possible event timings, however it is based only on the enabled transitions. Thus if there are no action transitions enabled then MaxDelay from Definition 3.9 is used to compute the maximum delay allowed by the system model. The MaxDelay function corresponds to a concrete Delays operator in Definition 3.4.

Definition. 3.9 $\mathsf{MaxDelay}(\mathcal{Z}, f)$ *computes the furthest absolute time moment less than* f *(future time horizon) from a symbolic state set* $\mathcal{Z}$ *reachable only via delay and internal transitions:*

$$\mathsf{MaxDelay}(\mathcal{Z}, f) \stackrel{def}{=} \max\{z'|_{\mathcal{X}} \mid \langle \overline{\ell}, z\rangle \in \mathcal{Z}, \langle \overline{\ell}, z\rangle \stackrel{[0,f]}{\Longrightarrow} \langle \overline{\ell}', z'\rangle\} \tag{3.10}$$

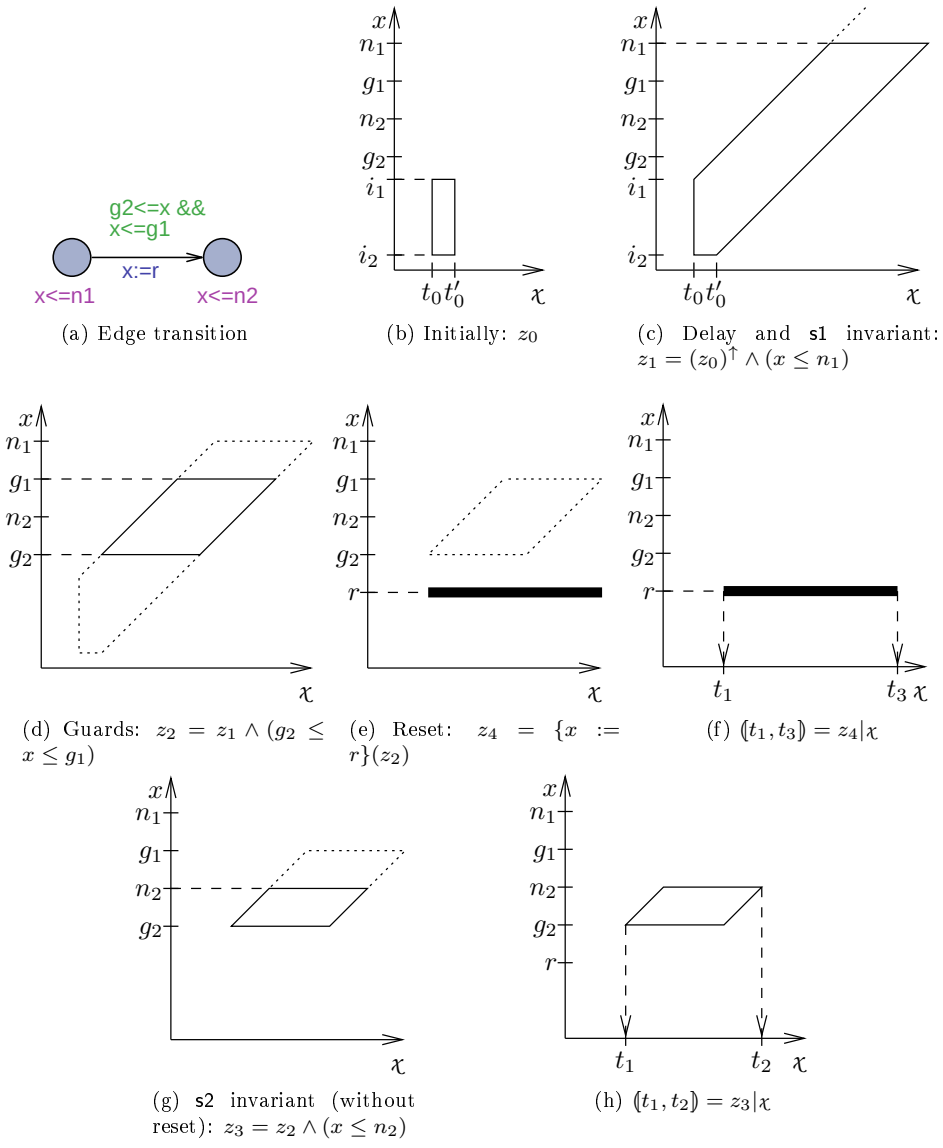
Ideally, one would always use avfuture horizon $f = \infty$, however for efficiency reasons it is beneficial to limit the horizon and minimize the number of symbolic states and reduce the number of redundant (delay closure) calculations that would be repeated when time progresses.

Example. Figure 3.3a shows simple automaton with two locations and an edge between them, automaton operates on clock x. $x \leq n_1$ and $x \leq n_2$ are invariants on locations s_1 and s_2 respectively. The edge is decorated by the guard $g_2 \leq x \leq g_1$ and a reset $x := r$. The auxiliary clock t to control and monitor the accumulated time in the model. Suppose the automaton starts at location s_1 with symbolic zone $i_2 \leq x \leq i_1$ at the moment t_0. The resulting zone z_0 is shown in Figure 3.3b. To find a symbolic transition successor we need to find out how long we can delay in current location s_1. To do this, the future operator is applied and bounded by current invariant $x \leq n_1$, resulting in the zone z_1 shown in Figure 3.3c. In order to fire a transition we need to make sure that the edge is enabled, hence we compute when the guard is satisfied by applying guard expression $g_2 \leq x \leq g_1$ on the zone z_1 and get the result shown in Figure 3.3d. If there is an assignment $x := r$ we apply a projection and get the result shown in Figure 3.3e. For a more complex case, let's assume there is no assignment and we apply invariant from target location, the result is in Figure 3.3g. From the last zone we can compute out when this transition can be fired. In this case, the time interval is between t_1 and t_2 derived from the bounds on the absolute time clock $\mathcal{X}$ in the zone z_4, Figure 3.3e.

3.3.5 The Symbolic Online Test Algorithm

We consider that an observable event is described by an action and an interval timestamp. The action is a channel synchronization that potentially has some integer variables attached to mimic value passing. The test specification then consists of a UPPAAL closed system model (system requirements and environment assumptions composed in parallel) and the test interface description. The test interface declares the set of observable input and output actions, the model time units (model time unit value in real world micro seconds) and a value for testing timeout. Once the interface is known the system model is partitioned into implementation and environment processes by a dependency analysis of the interprocess communication via channels and variables.

Algorithm 3 shows how the online test algorithm applies symbolic techniques, communicates with the IUT and computes the test verdict. The algorithm takes the following inputs: a system model partitioned into IUT requirements $\mathcal{S}$ and environment assumptions $\mathcal{E}$, a connected IUT and the time bound T for testing. The algorithm also has a few parameters: the *future* defines how much time into the future should the algorithm look ahead of time, output latency *outLatency* and input latency *inpLatency*. For simplicity assume that the input and output latencies are zero. The algorithm uses the following additional functions: GetTime() returns the global time reference with respect to the beginning of



(a) Edge transition

(b) Initially: z_0

(c) Delay and s1 invariant: $z_1 = (z_0)^{\uparrow} \wedge (x \leq n_1)$

(d) Guards: $z_2 = z_1 \wedge (g_2 \leq x \leq g_1)$

(e) Reset: $z_4 = \{x := r\}(z_2)$

(f) $(\!|t_1, t_3|\!) = z_4|_{\mathcal{X}}$

(g) s2 invariant (without reset): $z_3 = z_2 \wedge (x \leq n_2)$

(h) $(\!|t_1, t_2|\!) = z_3|_{\mathcal{X}}$

Figure 3.3: Example of symbolic edge transition.

testing, $\mathsf{Random}(A)$ is a generic random function which returns a random member of set A. The *buffer* variable is used to accumulate output actions incoming from the IUT together with their arrival time-stamp. The *action* variable contains the information about an event on specific channel (possibly with some data) at a specific moment in time estimated by an interval $(\!|from, till|\!)$.

The symbolic algorithm follows slightly different strategy than Algorithm 2: 1) the randomization between input action and delay is resolved at once by having a full set of options at once, 2) the outputs are processed as fast as they arrive (outputs may even preempt inputs) 3) the set of choices are calculated on a separate copy of a state set estimate making a reservation that outputs may

preempt inputs.

Algorithm 3: Symbolic online test, $OnlineTestImp(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T)$.

Input: $future := 1\text{mtu}$, $outLatency := 0$, $inpLatency := 0$

```
1  Z := {⟨s0, e0⟩)};                    // let the set contain an initial state
2  while Z ≠ ∅ ∧ GetTime() ≤ T do
3      while not buffer.isEmpty do       // consume the output buffer
4          e:=buffer.poll(); ;                          // dequeue first event
5          Z := Z after ⟦e.from, e.till⟧e.channel;      // apply it to the
           state set
6          if Z = ∅ then return fail;                   // check if it's OK
7      now := GetTime();
8      Z := Z after ⟦now − outLatency, now + future⟧;
9      if Z = ∅ then return fail;                       // is it OK to delay?
10     C := Z after ⟦now + inpLatency, now + future⟧; // copy for choices
11     c := Random(Events(C, A_inp) ∪ {[0, MaxDelay(C, now + future)⟧τ});
12     if buffer.isEmpty then
13         t = Random(⟦c.from, c.till⟧);
14         sleep until t or wakeup on output at t' ≤ t;
15         if buffer.isEmpty and e.channel ≠ τ then
16             from := GetTime();
17             res=send_input c.action to IUT;
18             if res==success then
19                 till := GetTime();
20                 Z := Z after ⟦from, till⟧c.action;
21 if Z = ∅ then return fail else return pass
```

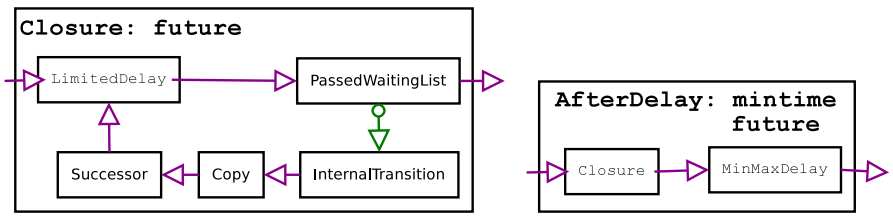
3.4 Online Test Implementation

The online test algorithm is implemented in the tool UPPAAL TRON and demonstrated in [46]. The UPPAAL TRON instructions manual is in the Appendix A.

This section describes how symbolic techniques using pipeline design pattern to process the symbolic states. We reuse as many components from UPPAAL architecture [5] as possible and describe only the new ones. The components are called symbolic state *filters*. A filter accept a symbolic state, computes the assigned operation and send the resulting symbolic state to the next connected filter. The online test algorithm is implemented by designing a set of filters for computing the after delay and the after action transitions, and also a list of available input actions. The components are described in a bottom-up way: starting with the basic filters and from there building the more complex ones. The online test code uses the filter operations and follows the Algorithm 3. At the end of testing the verdict and conclusion is decided by comparing a list allowed transitions from a last good state set with what actually happened at the very end of test.

3.4.1 Internal and Delay Transition

Figure 3.4a shows the pipeline algorithm for the `Closure` filter which computes a closure of internal and delay transitions over the current system state set. The general structure of the filter is similar to filter for reachability analysis from [5]. Closure computation starts with the `LimitedDelay` filter which applies future (delay) operation and constrains symbolic zone with $\chi \leq$`now+future` (equivalent to after d, where $d \in \mathbb{R}_{\geq 0}$): `now` is a current time representation in model time units and `future` is a parameter for `Closure` filter. The resulting states are accumulated in the `PassedWaitingList` filter: it checks if the state is new (not included in `passed` list), puts it into the `passed` list, adds the new states to the `waiting` list and finally sends to the output of `Closure` filter. When the whole state set is has been processed, the loop marked by arrow with circle is triggered. Which pulls states from the `waiting` list in the `PassedWaitingList` and sends them to the `InternalTransitionFilter`. The `InternalTransitionFilter` is based on the `TransitionFilter` which computes a list of enabled edges (checks integer guard expressions and synchronizations). In addition, `InternalTransitionFilter` passes only edges that are not decorated with observable channel synchronizations (i.e. it executed potential internal transitions). The pair of a state and a list of edges is then sent to `Copy` filter which creates a separate copy of a state for each transition (preparing a separate successor state). Then the `Successor` filter receives the pair and computes the successor state by completing the symbolic transition (applies clock guard constraints and assignments). The resulting successor state is pushed further to `LimitedDelay`, later `PassedWaitingList` and the loop continues until no new symbolic states are produced (`waiting` list becomes empty).



(a) Closure filter. (b) After delay filter.

Figure 3.4: Filters for state set update after delay.

Figure 3.4b shows how the after delay operation is computed within the `AfterDelay` filter using `Closure`. The `AfterDelay` filter is parameterized with `mintime` and `future` bounds which controls the lower and upper bounds of a delay performed. At first, the entire state set is fed into `Closure` filter, then resulting states are pushed through `MinMaxDelay` and the result is sent out. The `MinMaxDelay` is similar to `DelayFilter` except it applies two constrains: `mintime`$\leq \chi \leq$ `maxtime` where `maxtime` is set to `now+future`.

3.4.2 Observable Action Transition

Figure 3.5 shows how the after action operation is performed by the `AfterAction` filter. The `AfterAction` filter has `action` and `future` parameters. `action` contains information on channel synchronization together with the lower and

upper bounds $[l, u]$ capturing when the synchronization happened, and with the variable values passed. The `future` parameter come from `-F` command line argument and tells how much time to precompute into the future after the action is executed. At first, a `Closure` is performed with `future`$= u$, followed by `MinMaxDelay` with `mintime`$=l$ and `maxtime`$=u$, i.e. it prepares the states for action channel synchronization. The `ActionTransition` is based on `TransitionFilter` except that it selects only edges that are decorated with the given `action` channel synchronization. The `Copy` and `Successor` filters compute the resulting states after the action transition is fired. The `Data` filters the states and leaves only those that match the variable values specified in the `action` parameter. If `future` is positive then additional `Closure` with `future` is computed and the resulting states are sent to output.

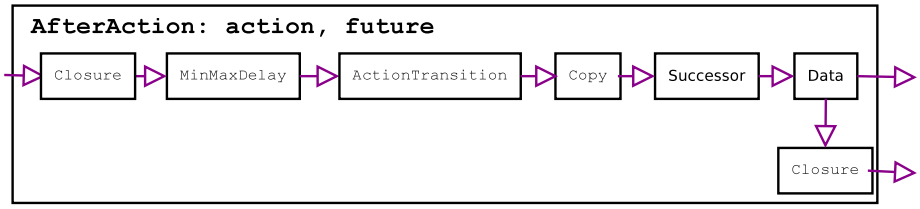


Figure 3.5: Filter for state set update after action.

3.4.3 Computing Allowed Actions

Figure 3.6 shows the symbolic state filter pipeline for computing the possible inputs and delays. The `Choice` filter computes all observable input/output events from a given state set. The resulting action choice options can then be used to decide what input is allowed when, to predict the allowed outputs and allowed delays. The first instance of the `Copy` filter ensures that the `Choice`

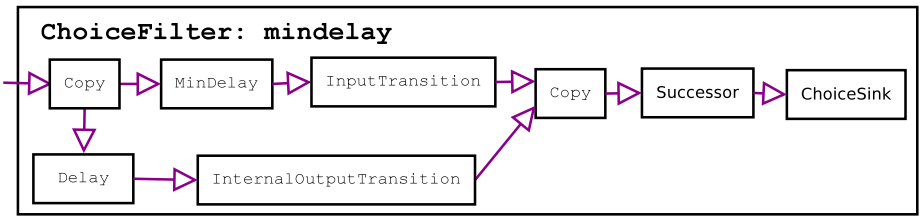


Figure 3.6: Filter for possible event estimation.

filter operates on its own copy of states and does not alter the original ones.

The `Choice` filter works by computing successor states for each transition: input transitions pass through `MinDelay` and `InputTransition`. Output and internal transitions pass through `Delay` and `InternalOutputTransition`. `MinDelay` filter constrains the input transitions by `now+mindelay`$\leq \chi$ so that only input transitions with realistic input latency are processed and inputs with strictly faster response time than `mindelay` are dropped. It is important to remark that `MinDelay` is a special filter that applies only the invariants that are specified on environment model and skips IUT invariants to avoid imposing IUT restrictions on environment behavior.

The internal and output transitions go through a regular `Delay` operation with invariants for the whole system and without any special restrictions. The resulting states are accumulated at `ChoiceSink` where states are sorted into input, output and internal choice lists. The choice options are decorated with channel synchronization and timing (like Definition 3.8), and the maximum system delay is computed by extracting largest upper bound from all successor states (see Definition 3.9).

The resulting choice lists are used for choosing the input stimuli and finally for giving diagnostic information when the test fails. The maximum system delay is used by the online test algorithm if the input choice list is empty. Note that internal and output transitions need to be processed too in order to compute a correct maximum system delay estimate.

3.4.4 Test Verdict and Basic Diagnostics

The online test algorithm terminates if current state set of the system becomes empty. Normally this would happen only if TRON observes that IUT failed to conform to the specification, however in practice it is possible that state set becomes empty due to test execution platform being too slow to satisfy the assumptions specified in the environment model. Moreover, developers need to identify the cause of a failure too. Thus an elaborate procedure is needed to determine what (could have) went wrong.

Currently TRON provides the following verdicts:

passed – no non-confirmance has been observed,

failed – non-conformance has been observed,

inconclusive – some assumption about online test failed and test can no longer continue.

A simple diagnostic informations is provided based on last good state set in case of failed or inconclusive verdict. This diagnostics is naive in sense that it assumes that the fault happened at the very last step of online test. On the other hand the procedure automates the tedious process of inspecting the last good state set which may easily contain several hundreds of symbolic statesand thus cumbersome to inspect manually. Algorithm 4 shows the pseudo-code for calculating failed or inconclusive verdict and drawing the conclusion. The `Action` is a class containing the following data about actual input/output observed: channel identifier, values for associated data, the interval of expected occurrence time (*lowerBound* and *upperBound*). The type `Choice` contains data about possible choice for input stimuli: channel identifier, values for associated variables and the interval of enabled time (*minBound* and *maxBound*). The `Choice` objects are generated by the `ChoiceFilter` filter inside UPPAAL engine, while *Action* objects are created, decoded and time-stamped by the test driver connected to IUT adapter.

Initially, the possible input and output choices are computed from the last good state set (stored in `backup`). Then the algorithm is split into two parts depending on the immediate cause of test termination: upon an observable I/O (lines 3-21) or a silent delay (lines 23-33). The observable I/O part is split into an analysis of inputs (lines 4-9) and of outputs (lines 11-21) depending on what

Algorithm 4: Verdict based on a last good state set.

```
   Input: StateSet backup, Event e, Choice c
   Output: verdict: Passed, Failed or Inconclusive
 1 A_inp =EnvOutput(backup); A_out =ImpOutput(backup);
 2 if e then                          // state set empty upon observable I/O
 3     if e.isInput then    // if e is input, then there was a choice
 4         "Decided to input c, but executed as
           e.channel@[e.lowerBound,e.upperBound)";
 5         "The target state was: c.targetState";
 6         if c.maxBound < e.lowerBound then
 7             return Inconc(Input executed too late);
 8         else if e.upperBound < c.minBound then
 9             return Inconc(Input executed too early);
10     else                                          // e is an output
11         "Got unacceptable output
           e.channel@[e.lowerBound,e.upperBound)";
12         "Expected outputs: A_out";
13         boolean tooLate=false, tooEarly=false;
14         forall c_o ∈ A_out s.t. e.channel==c_o.channel do  // see outputs
15             if e.upperBound < c_o.minBound then  tooEarly=true;
16             if e.lowerBound > c_o.maxBound then  tooLate=true;
17         if tooLate ∧¬ tooEarly then
18             return Failed(Output produced too late);
19         else if ¬tooLate ∧ tooEarly then
20             return Failed(Output produced too early);
21         else  return Failed(Observed unacceptable output);
22 else            // there was no observable I/O, only time delay
23     "Last time-window is beyond maximum allowed delay";
24     if t_S < t_O then
25         return Inconc(Bug: output deadline behind allowed delay);
26     else if t_O < t_S then
27         return Inconc(Model contains time lock)
28     else if t_S < t_T then
29         return Failed(IUT failed to send output in time)
30     else if t_I < t_O then
31         return Failed(IUT failed to send output in time)
32     else  return Inconc(Model contains deadlock)
33 return Inconc(Empty stateset. Bug, please report it.);
```

kind of I/O was observed. The text in quotation marks is printed by TRON into a log explaining the flow of the analysis.

If the test terminates by offering an input, then the executed input event e is compared with choice c computed before input is offered: if a lower bound of the actual input e is less than an upper bound of the choice c then input must have been executed too late, otherwise the upper bound of executed input is checked against lower bound of choice for possibility of input being executed too early. The third option could be that bounds of executed input and computed

choice overlap, but then either the resulting state set would not be empty (and test would not terminate) or IUT model is not input enabled – hence a violation of online test assumption. In either case, the test is inconclusive because TRON failed to execute input according to environment model: it did not observe any fault from IUT and the test cannot continue either.

If the test terminates due to an observed output action, the algorithm tries to determine if the output arrived too early, too late, contained wrong data values or was just not acceptable. Lines 14-16 try to identify the corresponding choice, and therefore the required timings for the observed output. If the output choice is identified, TRON tries to determine whether the actual output was too early or too late, otherwise TRON complains that the output is simply not acceptable for conformance to the model.

If the last executed step in the online test was a delay, then many things may be wrong: IUT failed to produce output in time and thus test fails, or the IUT might have been expecting input at the same time as required to report output and thus the test is inconclusive, or the system model contains a deadlock. Upon the online test termination, the following timings (at absolute scale) are used from the last good state set:

t_S – the largest permissible delay for IUT without observable I/O.

t_O – the largest permissible delay for IUT output.

t_T – the largest permissible delay for the environment without inputs, i.e. this is how much tester can delay at most without issuing any input. Such delay is determined by `ChoiceFilter` which computes the system's behavior without IUT invariants.

t_I – the largest permissible delay for the input by the environment, computed by `ChoiceFilter`. If the set of input choices is empty, then t_0 is taken instead.

Soundness of Verdict Algorithm

There are two ways for the online test to terminate without "pass": either the last observed action could not be matched in the model, or the model could not delay more than the last observed silent delay.

If termination happened because of an observable event, then there are two cases: wrong input — means that the tester failed to generate the input according to environment model, hence test verdict is "inconclusive", or wrong output — means that the IUT produced an output that could not be matched at the model, hence test verdict is "failed".

If the online test terminated upon delay, then there are many possible situations: some fall under "failed" verdict, some under "inconclusive" and some can be considered as gray area depending on concrete interpretation of a test case (we still denote such situations as "inconclusive", following the principle "not guilty until proven so", because of lack of evidence).

These upper bounds t_I, t_O, t_T and t_S can be considered as points in time and we can draw a conclusion based on the relations between them. There are $4! = 24$ permutations possible, and $2^3 = 8$ equality and inequality combinations for each permutation, hence giving a total of 192 combinations. Some of the

combinations with equalities can be written in multiple ways, giving only 79 unique combinations (see Table 3.2). Most of them still contain contradictions like the following:

$t_T < t_S$**:** the tester's behavior is obtained from a system model without IUT invariants, hence the tester should be able to delay at least as much as t_S.

$t_T < t_I$**:** inputs are described by the environment model, hence the tester should be able to delay at least as much as t_I.

$t_S < t_O$**:** outputs are generated by the IUT model, it should be able to delay at least as much as output bound t_O, otherwise such output could not be computed in the first place.

Finally, when contradicting combinations are removed, we end up with 16 meaningful cases enumerated in Table 3.1. Based on the logically implied relations

No	Bounds	$t_O < t_S$	$t_S < t_T$	$t_I < t_O$	Verdict and cause
1	$t_I = t_O = t_S = t_T$	false	false	false	Inconclusive, deadlock
2	$t_I = t_O = t_S < t_T$	false	true		Failed to send output in time
3	$t_I = t_O < t_S = t_T$	true			Inconclusive, time-lock
4	$t_I = t_O < t_S < t_T$	true			Inconclusive, time-lock
5	$t_I < t_O = t_S = t_T$	false	false	true	Failed to send output in time
6	$t_I < t_O = t_S < t_T$	false	true		Failed to send output in time
7	$t_I < t_O < t_S = t_T$	true			Inconclusive, time-lock
8	$t_I < t_O < t_S < t_T$	true			Inconclusive, time-lock
9	$t_O < t_I = t_S = t_T$	true			Inconclusive, time-lock
10	$t_O < t_I = t_S < t_T$	true			Inconclusive, time-lock
11	$t_O < t_I < t_S = t_T$	true			Inconclusive, time-lock
12	$t_O < t_I < t_S < t_T$	true			Inconclusive, time-lock
13	$t_O = t_S < t_I = t_T$	false	true		Failed to send output in time
14	$t_O = t_S < t_I < t_T$	false	true		Failed to send output in time
15	$t_O < t_S < t_I = t_T$	true			Inconclusive, time-lock
16	$t_O < t_S < t_I < t_T$	true			Inconclusive, time-lock

Table 3.1: Unique and meaningful cases of bound permutations leading to a failed or inconclusive verdict.

between t_I, t_O, t_S and t_T instances, we characterize the cause behind the verdict. We distinguish a property of time-lock ($t_O < t_S$), where IUT is able to delay until t_S but is not able to produce an output after t_O. Such property implies that the model contains deadlock and hence not suitable for testing. Therefore all entries (# 3,4,7,8,9,10,11,12,15,16) with $t_O < S =$ true are marked with verdict "inconclusive". Another property $t_S < t_T$ means that the environment model may progress further than IUT, i.e. tester had a legal choice to delay, therefore the deadlock at the end of online test is caused by the IUT and cases # 2,6,13,14 are assigned verdict failed due to missed output deadline. Now for the remaining (#1 and #5) we can use the evidence of whether $t_I < t_O$ is true, meaning that the tester from time point t_I does not have any other choice but delay, therefore the deadlock is caused by IUT again and therefore the verdict is "failed" in case #5. The remaining case #1 does not present any more evidence

(at least from the analyzed bounds), except perhaps a global deadlock, hence it is safe to declare verdict "inconclusive".

Note that almost all "inconclusive" verdicts indicate a time-lock, meaning that our assumption that the model is deadlock-free is wrong, and hence online test should not be applied on such a model. The other "inconclusive" verdict, without time-lock, is a very special case where model of IUT and model of environment synchronize and cause a deadlock together at the same time $t_S = t_T$ (deadlock-free assumption broken again), which is a sign of bad invariant, most probably at the environment model (this can be determined by inspecting the last good state set dumped by TRON). If the bad invariant is only at the IUT model, then it is very likely that a second case will be hit instead.

We conclude that the verdict algorithm either declares the non-conformance for sure, or shows the symptoms that the model is not suitable for online test.

3.5 Discussion

In addition to timed delays in conformance testing we considered the environment of the IUT. We conclude that the assumptions about environment play important role in the system: loosely specified environments are more discriminating towards implementation and may expose more faults than concrete ones, but at the same time they are more expensive to test. In the extreme cases, environment may allow most exhaustive tests and become passive monitoring if restricted from issuing inputs at all. There is also a tradeoff on how realistic the environment model should be: more realistic models tend to be very detailed and constrained, whereas more abstract model are simpler to describe but may expose faults that are not observable in the real environment. Moreover, explicit environment model can have many engineering interpretations: most permissive environment can be used for load/stress testing, realistic models provide IUT-in-the-loop simulations, specific use-case scenarios are like human created test cases, and concrete test execution traces can be re-imported for debugging purposes.

In a spirit similar to [62] we proposed an abstract online test algorithm with support for real-time. We conclude that the algorithm is sound (the failed verdicts show that IUT does not conform) and under certain conditions (input enabledness, IUT determinism and time digitization) the online test is complete (exhaustive) given sufficient time. The assumptions for exhaustiveness are impractical but we have shown that non-conforming implementation can be detected in principle. Moreover, explicit modeling of environment allows to optimize online tests even more toward realistic environment where faults are less likely to manifest.

Further, we conclude that it is possible to implement a real-time test algorithm reusing basic building blocks of a model-checker. We show that the same basic symbolic operations can be applied for state estimation purposes and that those operations can be grouped into new UPPAAL pipeline components reducing software engineering and maintenance efforts. However we had to add several non-trivial operators to track absolute time and distinguish observable transitions.

The symbolic online test algorithm is refined one more step further by not relying on infinitely precise time measurements as abstract algorithm assumed.

Instead, the interval time-stamping technique is used where the time measurements are mapped to symbolic representation. Remarkably the time mapping is very similar to digitization method proposed by [59], except that our approach is more practice-oriented by combining the resolutions of both physical clock and model time units and by proposing interval time-stamp traces which essentially serve as a compact representation of uncountably large set of real-valued trace set.

In addition we propose heuristic algorithm to provide basic diagnostics, thus it is possible to locate the offended parts of the model if the test fails. The heuristics is based on a systematic and comprehensive analysis of the last good state set estimate. The implementation of diagnostic algorithm reuses the same UPPAAL symbolic analysis components, thus the diagnostic analysis is consistent with the rest of test generation and evaluation. Ideally we would want to be able to identify the exact location of a violated model element, however it may be turn out to be ambiguous given the non-deterministic specifications, thus it remains a challenge for future research.

Bounds	Verdict
$t_I = t_O = t_S = t_T$	Inconclusive, deadlock
$t_I = t_O = t_S < t_T$	Failed
$t_I = t_O < t_S = t_T$	Inconclusive, TL
$t_I = t_O < t_S < t_T$	Inconclusive, TL
$t_I < t_O = t_S = t_T$	Failed
$t_I < t_O = t_S < t_T$	Failed
$t_I < t_O < t_S = t_T$	Inconclusive TL
$t_I < t_O < t_S < t_T$	Inconclusive, TL
$t_I = t_O = t_T < t_S$	Contradiction ($t_T < t_S$)
$t_I = t_O < t_T < t_S$	Contradiction ($t_T < t_S$)
$t_I < t_O = t_T < t_S$	Contradiction ($t_T < t_S$)
$t_I < t_O < t_T < t_S$	Contradiction ($t_T < t_S$)
$t_I = t_T < t_O = t_S$	Contradiction ($t_T < t_S$)
$t_I = t_T < t_O < t_S$	Contradiction ($t_T < t_S$)
$t_I < t_T < t_O = t_S$	Contradiction ($t_T < t_S$)
$t_I < t_T < t_O < t_S$	Contradiction ($t_T < t_S$)
$t_I = t_T = t_S < t_O$	Contradiction ($t_S < t_O$)
$t_I = t_T < t_S < t_O$	Contradiction ($t_T < t_S$)
$t_I < t_T = t_S < t_O$	Contradiction ($t_S < t_O$)
$t_I < t_T < t_S < t_O$	Contradiction ($t_T < t_S$)
$t_I = t_S < t_O = t_T$	Contradiction ($t_S < t_O$)
$t_I = t_S < t_O < t_T$	Contradiction ($t_S < t_O$)
$t_I < t_S < t_O = t_T$	Contradiction ($t_S < t_O$)
$t_I < t_S < t_O < t_T$	Contradiction ($t_S < t_O$)
$t_I = t_S = t_T < t_O$	Contradiction ($t_S < t_O$)
$t_I = t_S < t_T < t_O$	Contradiction ($t_S < t_O$)
$t_I < t_S = t_T < t_O$	Contradiction ($t_S < t_O$)
$t_I < t_S < t_T < t_O$	Contradiction ($t_S < t_O$)
$t_O = t_I = t_S = t_T$	* Inconclusive, deadlock
$t_O = t_I = t_S < t_T$	* Failed
$t_O = t_I < t_S = t_T$	* Inconclusive, TL
$t_O = t_I < t_S < t_T$	* Failed, TL
$t_O < t_I = t_S = t_T$	Inconclusive, TL
$t_O < t_I = t_S < t_T$	Inconclusive, TL
$t_O < t_I < t_S = t_T$	Inconclusive, TL
$t_O < t_I < t_S < t_T$	Inconclusive, TL
$t_O < t_I = t_T < t_S$	Contradiction ($t_T < t_S$)
$t_O < t_I < t_T < t_S$	Contradiction ($t_T < t_S$)
$t_O = t_T < t_I = t_S$	Contradiction ($t_T < t_I$)
$t_O = t_T < t_I < t_S$	Contradiction ($t_T < t_I$)
$t_O < t_T < t_I = t_S$	Contradiction ($t_T < t_I$)
$t_O < t_T < t_I < t_S$	Contradiction ($t_T < t_I$)
$t_O = t_T = t_S < t_I$	Contradiction ($t_T < t_I$)
$t_O = t_T < t_S < t_I$	Contradiction ($t_T < t_S$)
$t_O < t_T = t_S < t_I$	Contradiction ($t_T < t_I$)
$t_O < t_T < t_S < t_I$	Contradiction ($t_T < t_S$)

Bounds	Verdict
$t_O = t_S = t_I = t_T$	* Inconclusive, deadlock
$t_O = t_S = t_I < t_T$	* Inconclusive, TL
$t_O = t_S < t_I = t_T$	Failed
$t_O = t_S < t_I < t_T$	Failed
$t_O < t_S = t_I = t_T$	* Inconclusive, TL
$t_O < t_S = t_I < t_T$	* Inconclusive, TL
$t_O < t_S < t_I = t_T$	Inconclusive, TL
$t_O < t_S < t_I < t_T$	Inconclusive, TL
$t_O = t_S = t_T < t_I$	Contradiction ($t_T < t_I$)
$t_O = t_S < t_T < t_I$	Contradiction ($t_T < t_I$)
$t_O < t_S = t_T < t_I$	Contradiction ($t_T < t_I$)
$t_O < t_S < t_T < t_I$	Contradiction ($t_T < t_I$)
$t_S < t_O = t_I = t_T$	Contradiction ($t_S < t_O$)
$t_S < t_O = t_I < t_T$	Contradiction ($t_S < t_O$)
$t_S < t_O < t_I = t_T$	Contradiction ($t_S < t_O$)
$t_S < t_O < t_I < t_T$	Contradiction ($t_S < t_O$)
$t_S < t_O = t_T < t_I$	Contradiction ($t_S < t_O$)
$t_S < t_O < t_T < t_I$	Contradiction ($t_S < t_O$)
$t_S < t_I < t_O = t_T$	Contradiction ($t_S < t_O$)
$t_S < t_I < t_O < t_T$	Contradiction ($t_S < t_O$)
$t_S < t_I = t_T < t_O$	Contradiction ($t_S < t_O$)
$t_S < t_I < t_T < t_O$	Contradiction ($t_S < t_O$)
$t_S = t_T < t_I = t_O$	Contradiction ($t_T < t_I$)
$t_S = t_T < t_I < t_O$	Contradiction ($t_T < t_I$)
$t_S < t_T < t_I = t_O$	Contradiction ($t_S < t_O$)
$t_S < t_T < t_I < t_O$	Contradiction ($t_T < t_I$)
$t_S = t_T < t_O < t_I$	Contradiction ($t_T < t_I$)
$t_S < t_T < t_O < t_I$	Contradiction ($t_S < t_O$)
$t_T < t_I = t_O = t_S$	Contradiction ($t_T < t_I$)
$t_T < t_I = t_O < t_S$	Contradiction ($t_T < t_I$)
$t_T < t_I < t_O = t_S$	Contradiction ($t_T < t_I$)
$t_T < t_I < t_O < t_S$	Contradiction ($t_T < t_I$)
$t_T < t_I = t_S < t_O$	Contradiction ($t_T < t_I$)
$t_T < t_I < t_S < t_O$	Contradiction ($t_T < t_I$)
$t_T < t_S < t_I = t_O$	Contradiction ($t_T < t_I$)
$t_T < t_S < t_I < t_O$	Contradiction ($t_T < t_I$)
$t_T < t_S = t_O < t_I$	Contradiction ($t_T < t_I$)
$t_T < t_S < t_O < t_I$	Contradiction ($t_S < t_O, t_T < t_I$)
$t_T < t_O < t_I = t_S$	Contradiction ($t_T < t_I$)
$t_T < t_O < t_I < t_S$	Contradiction ($t_T < t_I$)
$t_T < t_O < t_S < t_I$	Contradiction ($t_T < t_I$)

Table 3.2: Test verdict based on bound permutations, where t_I - upper bound for inputs, t_O - upper bound for outputs, t_S - upper bound for system (IUT) delay and t_T is an upper bound for tester (environment) delay where IUT invariants are removed, TF - tester failed, TL - time-lock in the model.

Chapter 4

Adaptation Framework

In this chapter we show how the adapter is integrated into testing framework and may help resolving concurrency of input and output events. The problem is that in a realistic setup, a tester and an IUT are two separate entities which exist potentially at two different locations, they control inputs and outputs independently of each other. Moreover, it takes time for input and output signals to reach the other side through the test adapter, and as a consequence both tester and IUT may disagree on the order and timing of the observed signals because transmission of an I/O signal is a different event than a reception of the same signal. A classical approach to resolve the event ordering and timing is to develop some kind of time synchronization protocol, like [41]. However in a generic testing framework we cannot assume or impose a particular design decision on a given black-box IUT. Interestingly other testing frameworks ([11, 39]) seem to implicitly rely on a shared global reference clock to time-stamp and resolve the order of I/O signals.

We take a different approach and propose to model adapter explicitly in the specification model and consequently tester may use only one clock for time-stamping events and safely assume that it is local at the tester and not shared. This gives an advantage of decoupling the tester and the IUT and leaves a burden of time-stamping and ordering consistency to a single physical clock which is local to the tester and the IUT is free to use any other means to measure the time.

The goal of this chapter is to document the conceptual design of our test adapter and provide a proof that such adapter satisfies the required properties:

1. Input and output signals can not block each other and the protocol should not deadlock even if input and output interleave in the adapter. The requirement is essential for a protocol to be working at all.

2. Both the IUT and the tester should be input enabled. This requirement comes from our theoretical framework and from practical considerations where communication is implemented through some kind of media and the messages cannot be revoked nor stopped once issued without extra functionality in the communication protocol and our goal is to keep the protocol as simple and fast as possible.

3. The protocol should be non-intrusive or should not pose additional con-

straints over input/output messages. This requirement comes from desire for the IUT test instrumentation to be as close to deployment as possible and without putting too much (potentially faulty) additional functionality just for testing.

4. The protocol must allow arbitrary input and output interleaving as controlled by a tester and an IUT. Usually, black-box implementation is located outside tester's area of control, thus input/output events travel through channels independent from each other causing a natural interleaving. Early prototypes of TRON adapter were based on mechanism of locking all channels to resolve the consistent ordering and time-stamping at both IUT and tester sides. Effectively this mechanism caused additional delays due to blocking and serialization of both inputs and outputs which reduced possible interleaving orders of input/output events. Such setup makes testing simpler, but it also restricts and reduces the stress-load on IUT (the reported outputs may lock the channels and thus prevent inputs from stressing IUT).

5. The tester's actions should not interfere with the IUT functionality that it is not in the model. For example, if the protocol is synchronous then TRON should acknowledge the reception of output as fast as possible without causing any unnecessary delay to IUT. If such a delay is required to be tested, then the acknowledgment functionality should be part of the model explicitly.

Chapter starts with explaining how the specification model is adopted for testing using UPPAAL TRON, describes the virtual time framework which can be used to avoid communication latency, presents a verification of SocketAdapter implementation with and without virtual time and explains the consequences of adapter modeling and possible further development of more optimized and even distributed adapter.

4.1 Model Partitioning

UPPAAL TRON assumes that the specification model is partitioned into three parts like shown in Figure 4.1:

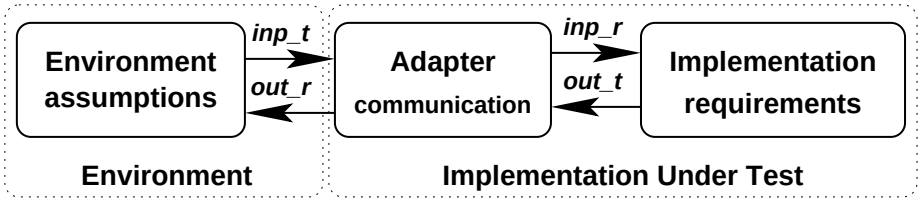


Figure 4.1: Partitioning of the specification model.

- Environment assumptions – processes that describe how the IUT environment behaves. It receives outputs on a set of channels out_r (should be ready to receive at any time) and transmits inputs on a set of channels inp_t.

- Adapter communication is modelled by a set of processes specifying the queueing and delay of the signals: *inp_t* (*out_t*) are queued, delayed and later emitted on *inp_r* (*out_r* respectively). The exact queueing algorithm and a bounded delay modeled in the adapter should reflect on how the adapter is implemented. Ideal adapter model would have space for infinite queue with infinitely many clocks, however only a bounded queue with bounded number of clocks are possible in timed automata, thus one has to measure the adapter in advance or analyze the implementation requirements and environment assumptions to determine the upper bound on the number of events arriving in short intervals.

- Implementation requirements specify a set of processes capturing actual requirements for the IUT: they receive inputs on a set of channels *inp_r* and transmit outputs on channels *out_t*.

Uppaal Tron then expects that channels *inp_t* and *out_r* are declared as observable. The inclusion of the adapter model as part of IUT requirements, makes sure that Uppaal Tron will consider all possible (and realistic) interleaving scenarios between simultaneous inputs and outputs while time-stamping events only on *inp_t* and *out_r* channels, thus effectively allowing IUT to have a different perception of input and output interleaving than the tester does.

The manual in Appendix A documents several adapter APIs to configure the observable inputs and outputs. The manual also documents the set of rules that Uppaal Tron uses to automatically deduce the partitioning of the model from observable channel declaration. The rules ensure that environment processes communicate with IUT processes only through observable channels (no side channel communication) and processes are partitioned consistently (each process is assigned either to environment or IUT side). The partitioning is then automatically used to treat environment and IUT processes accordingly (IUT invariants are discarded when computing a set of possible inputs).

4.2 Virtual Time Framework

Our virtual time (VT) framework provides a controlled accurate environment for running online real-time tests on a soft-real-time operating system where the effects of scheduling latencies and communication latencies are removed. The motivation is to verify the online testing paradigm in controlled, "lab" conditions, ability to replay online test traces, provide playground for education, and even to accelerate online tests on some real-time software in fast pace where time-related system calls can be diverted to a global shared clock, see e.g. smart lamp example described in Appendix A.

The VT framework thus assumes that all time delays are expressed in timed system calls, and that algorithmic computation time is virtually zero.

The idea is to replace all such timed system calls with calls to a virtual clock object which negotiates the time delay across all threads in the IUT-Tron system and advances the value of global time with the commonly agreed delay. The framework assumes that all participating threads are registered with virtual clock and thus it may safely advance the global time when all threads are waiting for time to elapse.

In order to ensure the consistency of the timed system calls, we use a monitor pattern with mutex and condition variables where each delay request is associated with a condition variable and all the calls related to this condition variable are guarded by locking the associated mutex.

The easiest way to override the timed system calls (e.g. POSIX family or Java monitor code) is to compile with the analogous functions supplied by Uppaal Tron binary. A remote IUT can override the calls in similar way and redirect requests to a virtual clock via TCP/IP socket protocol where each remote thread is represented by a local proxy thread (the virtual clock API is documented in Appendix A).

The adapter for a remote IUT in using the VT framework has the additional challenge to control the communication latency, thus it requires additional communication and blocking of the virtual time while the input/output signal is being transfered.

4.3 Adapter Protocol Verification

Uppaal Tron provides a number of APIs for test adapter to connect to the testing tool. The APIs specify a concrete transport layer and format of messages documented Appendix A, but the basic principles of input/output signal handling are the same across all APIs. The adapter protocol without virtual time is a simple asynchronous communication through mutex-guarded message queues at IUT and Tron sides. Since the protocol is asynchronous it is easy to ensure the correctness of the protocol just by following monitor paradigm and protecting the critical sections which access the input/output queues. Basically Tron offers two methods to connect a test adapter:

1. Local, via shared library API by sharing the same process address space. The communication is done via simple function calls which put the message into the receivers queue and immediately returns.

2. Remote, via standard input/output streams or TCP/IP socket streams. Here processes do not share the address space, and thus no function calls are possible. Instead, processes communicate through additional proxy threads which wait for incoming messages and put them into the receivers queue. It is easy to see that conceptually there is nothing new here and function calls are just replaced by stream communication.

As noted above, the VT framework relies on synchronous communication to prevent virtual clock from progressing while signals are traveling. In a local setting, this communication is completely transparent because messages reach the recipient queues immediately via one function call. In fact it can be switched even without recompiling a dynamic library. However, the VT framework with remote IUT requires synchronous communication over asynchronous streams and thus we need to accommodate extra synchronization messages into our protocol which make it much more complicated.

For our purposes we take a `SocketAdapter` as an example, which is the general enough and includes all features, in particular handling of virtual time with remote IUT as demonstrated with smart lamp example in Appendix A.

We model the `SocketAdapter` protocol in Uppaal and check the properties using the model-checker. Figure 4.2 shows a signal flow diagram of processes

involved in a test adapter. The protocol consists of two symmetric sides: tester

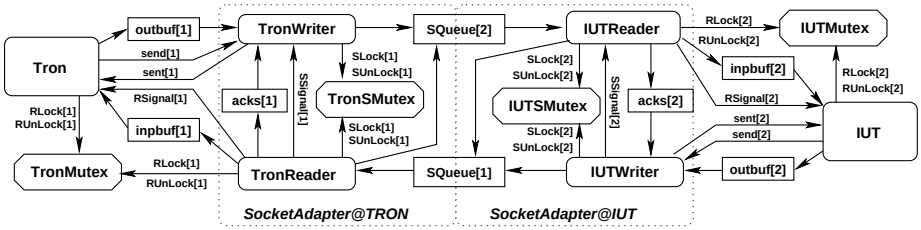


Figure 4.2: Signal flow of `SocketAdapter` model: rectangles are queues, octagons are mutexes, rounded rectangles are threads and arrows show direction of data flow.

(TRON) and IUT which are connected with two stream queues (`SQueue`). The TRON algorithm thread is represented by an abstract process `Tron` and IUT is represented by `IUT`. In the VT framework each side has two queues (`inpbuf` and `outbuf` for storing incoming and outgoing signals respectively) representing the socket, StreamReader and StreamWriter processes (instantiated as TronReader and TronWriter; IUTReader and IUTWriter respectively), mutex for incoming queue (TronMutex and IUTMutex), stream queue (`SQueue[2]` and `SQueue[1]`) and stream mutex (`TronSMutex` and `IUTSMutex`) guarding the respective stream queues.

Figure 4.3 shows a scenario where `Tron` sends an input to `IUT`: `TronWriter` locks the `TronMutex`, puts the input signal thread for writing to socket, and waits for acknowledgement effectively blocking the virtual clock from progressing; `IUTReader` picks up the incoming message from a socket, puts it into `inpbuf[2]` queue used by `IUT` and sends an acknowledgment which is picked up by `TronReader`; `TronReader` delivers the acknowledgement to `TronWriter` which returns to the `Tron` process; the `IUT` process then picks up the input from its queue and consumes it. Notice that while the input is on the way, the `IUT` is capable of sending the output to `Tron` at any time in parallel.

Figure 4.4 shows UPPAAL TA templates of all the processes that are instantiated in Listing 4.1 based on scheme in Figure 4.2:

- Figure 4.4a shows a template for mutex parameterized by UPPAAL channels `lock` and `unlock`. The template implements a simple locking mechanism where the requesting process is blocked if the mutex is locked and at most one process can lock/own the mutex.

- Figure 4.4b shows a template representing the tester and IUT: the process may choose to send a message by putting it into `outbuf` queue and return when the message is sent, or check the `inpbuf` queue for incoming messages which is guarded by a mutex and is thus surrounded by a `RLock` and `RUnLock` sequence.

- Figure 4.4c shows a `StreamReader` template which reads a command from an incoming stream queue if the queue is not empty. In the VT framework the command can be interpreted as: a) an acknowledgement for reception of previously sent signal, thus the acknowledgement is transfered to stream writer though a condition variable `SSignal` guarded by

`SLock` and `SUnLock`, b) an I/O signal which is put into the `inpbuf` queue guarded by `RLock` and `RUnLock`; an acknowledgment is sent to outgoing stream queue `SQueue` which is protected by `SLock` and `SUnLock` to avoid conflicting writes into the shared stream queue. The implementation of UPPAAL TRON time-stamps the output signal before it is put into `inpbuf`. Similarly the time of input signal is estimated by two time-stamps: before the message is sent and when thread returns after the message is sent. Notice that `StreamReader` acts as a proxy for `StreamWriter` on an opposite side.

- Figure 4.4d shows a `StreamWriter` template which is responsible for delivering the signal from `outbuf` to outgoing stream queue `SQueue` surronded by `SLock` and `SUnLock`. Then in virtual time case, `StreamWriter` waits for an acknowledgment notification on condition variable `SSignal`.

In the case of VT, the acknowledgement makes the communication between the tester and the IUT synchronous. It blocks the virtual clock when the signal is being transfered over stream queue. In order to make the adapter consistent with virtual time, the `StreamReader` threads are not registered with the virtual clock, because this thread acts as a proxy of already registered thread (it waits on incoming stream queue most of the time rather than condition) and we don't want to block the time when there are no messages.

In case of real world clock time the acknowledgement is not sent and is not waited for. This is obtained by by omitting the `outbuf`, `StreamWriter`, `acks` and stream mutex `SMutex` altogether, which makes the protocol simple, asynchronous and non blocking.

```
1  /** Instantiate Tron side: */
2  TronMutex = Mutex(RLock[1], RUnLock[1]);
3  TronSMutex = Mutex(SLock[1], SUnLock[1]);
4  Tron = Process(1);
5  TronReader = StreamReader(1);
6  TronWriter = StreamWriter(1);
7  /** Instantiate IUT side: */
8  IUTMutex = Mutex(RLock[2], RUnLock[2]); // input mutex
9  IUTSMutex = Mutex(SLock[2], SUnLock[2]); // socket mutex
10 IUT = Process(2); // IUT receiving and sending actions
11 IUTReader = StreamReader(2); // socket reader
12 IUTWriter = StreamWriter(2); // socket writer
13 system Tron, TronReader, TronWriter, IUTReader, IUTWriter, IUT, TronMutex, TronSMutex,
       IUTMutex, IUTSMutex;
```

Listing 4.1: Process instantiations in `SocketAdapter` model.

Listing 4.2 shows the rest of declarations structure supporting the adapter model.

```
1  const bool VirtualTime = true;
2  const int ACK = 0; // constant for ack message
3  /** queue implementation in OO style: */
4  const int MAXQ = 5; // maximum length
5  typedef struct {
6      int elem[MAXQ];
7      int [0,MAXQ-1] size;
8  } queue_t;
9  bool isEmpty(const queue_t& q) { return (q.size==0); }
10 bool isFull (const queue_t& q) { return (q.size==MAXQ-1); }
11 void add(queue_t& q, int elem) { q.elem[q.size++] = elem; }
12 int rem(queue_t& q) {
13     int e = q.elem[0], i;
14     for (i=0; i<q.size; ++i) q.elem[i] = q.elem[i+1];
```

```
15      q.elem[q.size--]=0;
16      return e;
17  }
18  /** there are two copies of identical "sides": 1=TRON, 2=IUT */
19  typedef int [1,2] side_t;
20  /** input buffer is protected by RLock and signalled through RSignal: */
21  int inpbuf[side_t];
22  chan RLock[side_t], RUnLock[side_t];
23  broadcast chan RSignal[side_t];
24  /** output buffer is transfered via send channel: */
25  int outbuf[side_t];
26  chan send[side_t], sent[side_t];
27  /** socket input stream queues, read is performed *only* by reader: */
28  queue_t SQueue[side_t];
29  /** write to socket is performed by both reader and writer, protected by SLock: */
30  chan SLock[side_t], SUnLock[side_t];
31  /** acks are protected by SLock too, changes are signaled by SSignal: */
32  int acks[side_t];
33  broadcast chan SSignal[side_t];
```

Listing 4.2: Global declarations of `SocketAdapter` model.

The following is a list of queries we checked to ensure that the protocol works are expected:

- Can TronReader and TronWriter write to the same socket at the same time? [No]
  E◇ TronReader.WriteAck ∧(TronWriter.SocketWrite ∨ TronWriter.CheckForAck)
- Is it possible for Tron to be waiting and be notified about incoming output? [Yes]
  E◇ Tron.Alert
- Is the socket stream queue always bound by the size of 2? [Yes for VT]
  A□ SQueue[1].size≤ 2
- Is it possible that there will be more than one acknowledgement needed at a time? [No]
  E◇ acks[1]>1∨acks[2]>1
- Can there be more than two messages in the input buffer when Tron is consuming them? [Yes]
  E◇ Tron.Consume∧ inpbuf[1]>2
- Is the protocol deadlock free? [Yes for VT]
  A□ ¬deadlock
- Is the protocol deadlock free while queues are not full? [Yes for RT]
  A□ (not isFull(SQueue[1])∧ not isFull(SQueue[2])) ⇒ (not deadlock)

4.4 Discussion

In this chapter we showed how the system model is partitioned into assumptions about the environment and requirements for IUT. The rules are used based algorithm to enforce the consistent model partitioning. We conclude that the partitioning is consistent with composition of environment and IUT requirement model and it is possible to enforce assumption automatically for many UPPAAL specifications except those that require runtime execution for interpretation (e.g. channel arrays).

We have made a formal model of test adapter protocol and conclude that the protocol symmetric in the sense that neither the tester nor the IUT has priority over issuing I/O events (fair and fully distributed control). We show that it is correct with respect to absence of deadlocks, order preservation of input (output resp.) events using model-checker. Moreover, the protocol can be deployed with very minor modifications in virtual time framework.

We provide a virtual time framework for testing systems in which it is possible to override time-related functions with calls to virtual clock. The interface uses a subset of the POSIX [33] interface, hence should be applicable for many software systems.

For practical purposes we offer additional method to describe I/O latencies in the adapter and I/O scheduling in general.

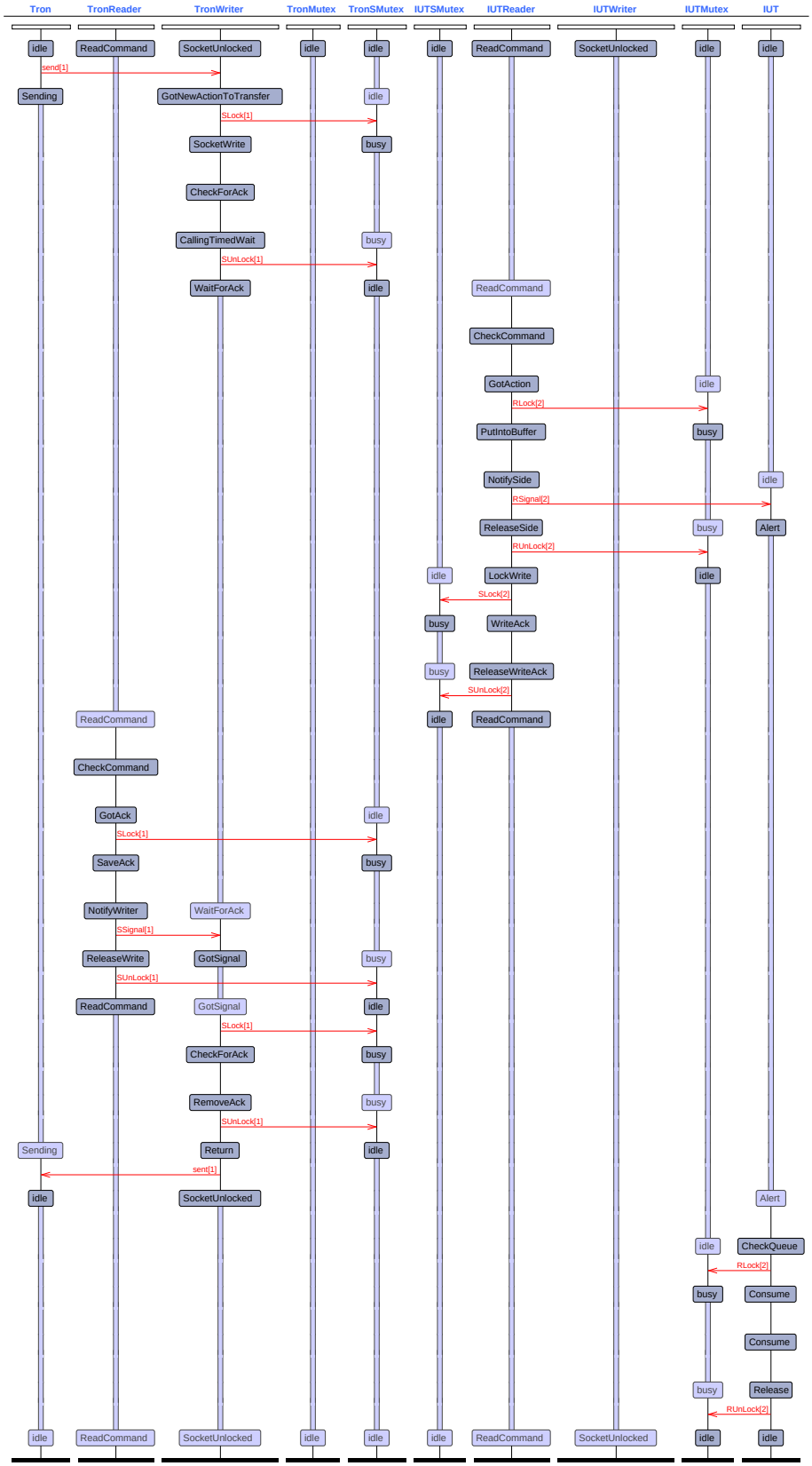
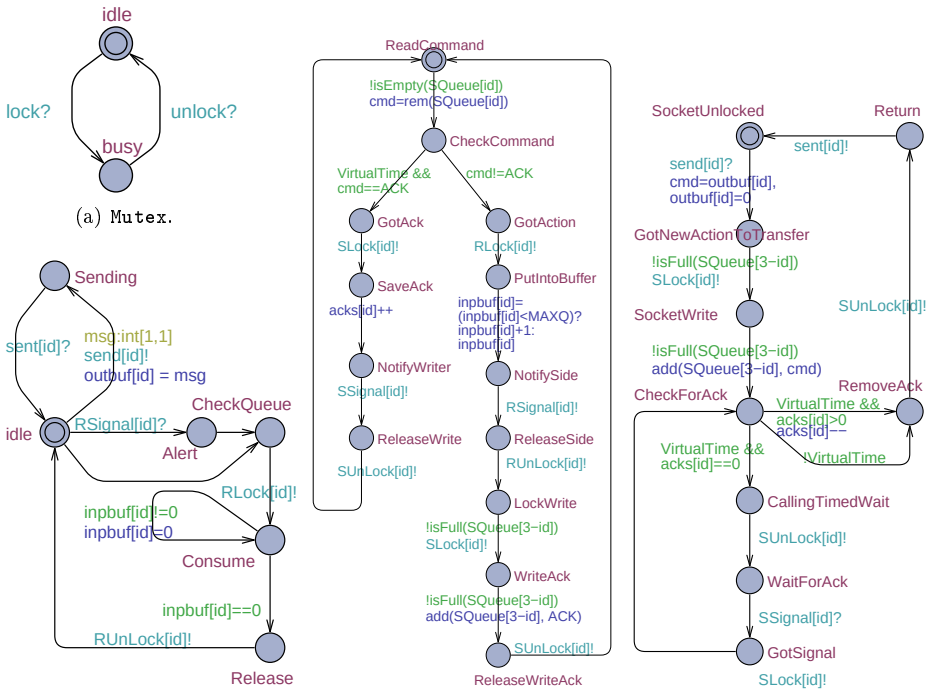


Figure 4.3: `Tron` sends input over `SocketAdapter` in virtual time framework.



(a) `Mutex`. (b) `Process`. (c) `StreamReader`. (d) `StreamWriter`.

Figure 4.4: UPPAAL TA templates for socket adapter model.

Chapter 5

Experiments

In this chapter we conduct empirical tests against online test implementation in tool TRON, we measure and evaluate the tool by the following aspects:

Correctness. We examine whether TRON implements UPPAAL features faithfully both in emulation of environment and monitoring of IUT requirements. The experiment is a non-exhaustive feature test which also demonstrates simple ways of interacting with UPPAAL models without coding effort. The experiment is described in Section 5.1.

Precision. We evaluate how TRON performs on a concrete execution platform by measuring timed behavior, scalability and performance of individual operations. The experiment is a continuation of a correctness evaluation in a quantitative sense where we try to obtain statistical measures on real-time performance. The details are described in Section 5.2.

Relevance. We conduct an experiment close to real world conditions where we examine what parts of IUT is stressed by online tests conducted by TRON. Online test algorithm contains a lot of randomization thus it is important to evaluate if the tool is capable to generate tests exercising relevant parts of IUT. The IUT code coverage experiment is described in Section 5.3.

Effectiveness. We look at whether TRON is able to detect faults in IUT in a similar setting as in the relevance experiment. The faults are automatically seeded by Java code mutation tool developed for evaluating JUnit test suites, thus we believe it provides a fair setup to evaluate effectiveness of our tests too. The automated mutant study is described in Section 5.4.

A similar study of measuring performance and effectiveness has been conducted by us in [42] on slightly different setting with different models and mutants have been created manually. In this chapter we provide experimental data on a larger scale containing more statistical evidence.

5.1 Basic Feature Test

The purpose of this experiment is to check that UPPAAL modeling features are correctly handled by TRON. We create a model as a test suite and connect TRON

to `TraceAdapter` which reads and emulates behavior of the given timed trace. The timed traces are exercising various parts of the model, thus all testing is driven by IUT implemented by trace script.

We distinguish two classes of tests:

Positive tests that forces TRON to emulate specific features of an environment model and `TraceAdapter` checks whether the response is described in the given trace. Such tests should always conclude with `test pass` verdict or test can be terminated prematurely by `TraceAdapter` after unexpected behavior of TRON is detected.

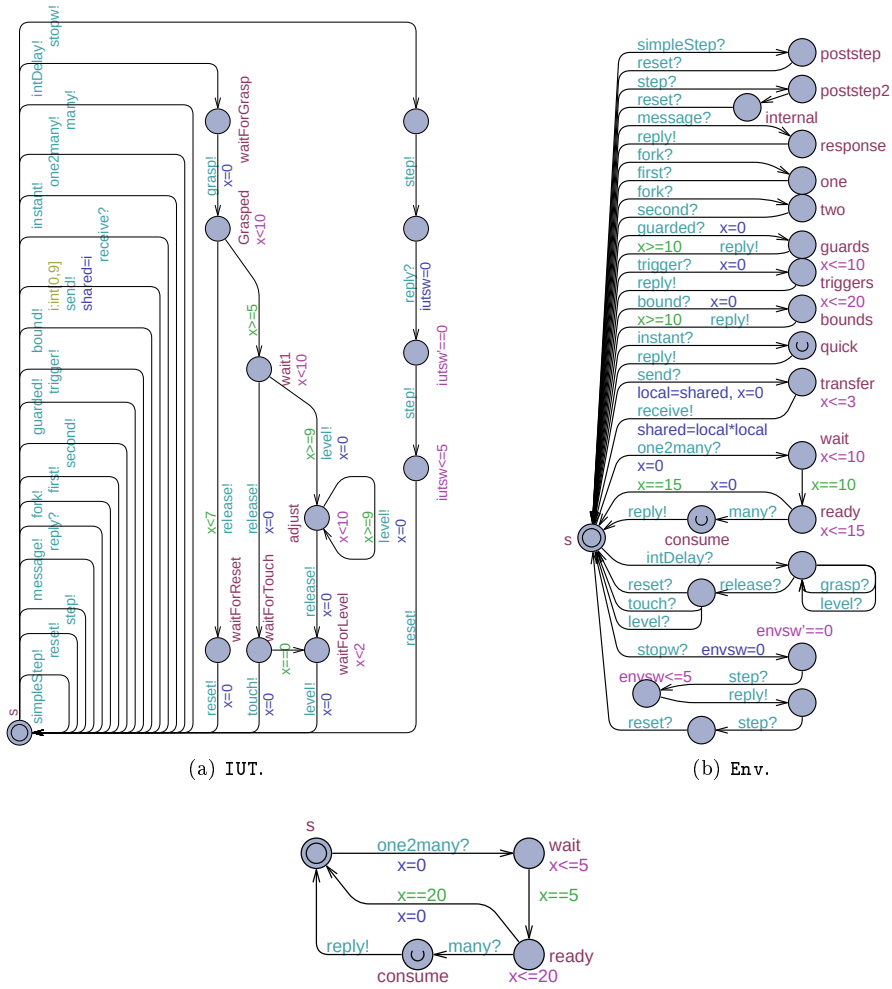
Negative tests that force TRON to exercise particular part of the testsuite model and detect misbehavior of the trace when some model feature is violated by the behavior of `TraceAdapter`. Such tests should always finish with `failed` verdict.

First we describe the test suite model, then show how test traces are created and conclude with results.

5.1.1 Model

In order to check modeling features we create one model containing most of UPPAAL features: simple output sequence, simple output and reply, non-deterministic behavior in time and action, clock guards and invariants, urgent locations, broadcast channels, stopwatches. The list is not exhaustive, in particular we do not aim to cover all possible combinations of features—only basic modeling elements. We put features of interests in both sides of the model: IUT requirements and environment assumptions. Figure 5.1 shows timed automata as requirements for IUT (Figure 5.1a) and two processes for environment (Figures 5.1b and 5.1c). `IUT` and `Env` with `Env2` are run in almost perfect synchrony, thus we only make sure that environment and IUT are input enabled only locally. All tests start with location vector $\langle IUT.s, Env.s, Env2.s \rangle$. First, `IUT` starts by selecting an output action which is received by `Env` and/or `Env2` process and thus environment ends up in particular location and further behavior depends on what other actions are enabled.

For example, the test may start with `IUT` selecting `simpleStep` output action, then `Env` is driven to location `poststep` and thus tester should expect `reset` output without time constraints. After output `reset` is observed, `Env` is brought back to location `s` and testing may continue further. For example, next test could start with `step` and test whether tester can perform internal transition non-deterministically. The test prefix `message` would test if tester can generate an input `reply` without time constraints (timing will mainly be determined by `-P` command line option). `fork` tests whether tester can arrive at two different locations and then be able to consume either `first` or `second` message controlled by IUT. Outputs `guarded`, `trigger` and `bound` test implementation of clock bounds: a guard and an invariant. Output `instant` would examine the implementation of urgent location. `send` tests the integer variable value transfer and simple computation. `one2many` tests broadcast channels which also engage `Env2` process. Notice that broadcast channels synchronization is non-blocking, thus, based on concrete timing the next event `many` may trigger either or both of `Env` and `Env2`.



(a) `IUT`. (b) `Env`.

(c) `Env2`.

Figure 5.1: Uppaal model for testing Tron.

Next we examine whether Tron can handle non-determinism in the model of IUT requirements with `intDelay` message. Depending on concrete timing of events, `IUT` can be in a several locations at the same time. Eventually Tron should be able to figure out whether concluding events `reset`, `touch` and `level` are appropriate based on the model and observed timings.

Finally we add test for stopwatches: `stopw` triggers test on `envsw` stopwatch in the environment, then the sides are changed and `iutsw` is stopped. In both cases the correctness is examined through subsequent handling of invariant: if the clock is properly stopped then there is time leak just before message `step`, thus IUT would be able to choose how much time should be leaked by delaying `step`.

Listing 5.1 shows global declarations and system instantiation of a test suite model.

```
1 chan reset, step, simpleStep, fork, first, second, message, reply, guarded,
2     trigger, bound, send, receive, instant, intDelay, grasp, release, touch,
3     level, stopw;
4 broadcast chan one2many, many;
5 int shared;
6 clock iutsw, envsw;
7 system IUT, Env, Env2;
```

Listing 5.1: Declarations and instantiation of test suite model.

5.1.2 Test Traces

After the test suite model is created, we could just implement one or a few complicated IUT which would drive TRON through the test suite model, but TRON comes with a `TraceAdapter` which can emulate any IUT by interpreting timed traces in a textual format. Originally `TraceAdapter` was implemented to replay the exact same sequences recorded by the driver during previous test runs, but the format is able handle one-step timed and action non-determinism, thus flexible enough for our basic testing purposes. The trace consists of two parts: preamble and a timed sequence of I/O events. Figure 5.2 shows the trace preamble: the declaration of test input/output interface and timing setup.

```
1 input reply(), receive(shared);
2 output reset(), simpleStep(), step(), fork(), first(), second(),
3       message(), guarded(), trigger(), bound(), send(shared),
4       instant(), one2many(), many(), intDelay(), release(),
5       grasp(), level(), touch(), stopw();
6 precision 10000;
7 timeout 100000;
```

Listing 5.2: Observable input/output and timing declaration for `TraceAdapter`.

Figures 5.2 and 5.3 show samples of test traces that `TraceAdapter` can interpret. The trace consists of commands terminated by semicolon. There are three types of commands: `delay`, `input` and `output` which tell `TraceAdapter` what has to be performed (output reported now, or delay) and what and when can be expected (by comma separated alternatives in all commands). The timing can be expressed in model time units, microseconds, in absolute and relative time scale. `TraceAdapter` terminates with an error if the expected action and/or its timing do not match. Please consult TRON manual for full details. For example, the trace in Figure 5.2a says that `TraceAdapter` should perform a relative delay with random duration between 0 and 1 model time units (line 1) and expect no inputs (no actions enumerated with comma). Then output `step` should be reported (line 2), and no input should be observed at this time. Then another randomized delay follows (line 3) and output `reset` is reported. Another delay is appended to make sure the timing offset is randomized again. The trace in Figure 5.2a triggers the test that starts with `simple` in the `Env` model.

The trace in Figure 5.2b implements `stopw` test. Note that command at line 5 specifies to wait for `reply` which should happen within 5 model time units,

```
1 delay [0.0,1.0];
2 output step();
3 delay [0.0,1.0];
4 output reset();
5 delay [0.0,1.0];
```

(a) Simple.

```
1 delay [0.0,1.0];
2 output stopw();
3 delay [0.0,10.0];
4 output step();
5 input
      reply()[0.0,5.0];
6 output step();
7 delay [0.0, 5.0];
8 output reset();
```

(b) Stopwatch.

```
1  delay [0.0, 1.0];
2  output one2many();
3  delay [0.0, 4.0];
4  output many();
5  delay 21.0;
6
7  output one2many();
8  delay [6.0, 9.0];
9  output many();
10 input reply()[0.0,
       0.0];
11 delay 15.0;
12
13 output one2many();
14 delay [11.0, 15.0];
15 output many();
16 input
       reply()[0.0,0.0];
17 input
       reply()[0.0,0.0];
18 delay 10.0;
19
20 output one2many();
21 delay [16.0, 20.0];
22 output many();
23 input
       reply()[0.0,0.0];
24 delay 5.0;
25
26 output one2many();
27 delay [21.0, 22.0];
28 output many();
29 delay 1.0;
```

(c) `one2many`.

Figure 5.2: Traces for `TraceAdapter` to exercise various parts of the testsuite model.

```
1  delay [0.0,1.0];
2
3  output intDelay();
4  delay [0.0,1.0];
5  output grasp();
6  delay [0.0,6.9];
7  output release();
8  delay [0.0,1.0];
9  output reset();
10 delay [0.0,1.0];
11
12 output intDelay();
13 delay [0.0,1.0];
14 output grasp();
15 delay [5.0,7.9];
16 output release();
17 delay [0.0,3.0];
18 output touch();
19 delay [2.0,3.0];
20
21 output intDelay();
22 delay [0.0,1.0];
23 output grasp();
24 delay [5.0,7.9];
25 output release();
26 delay [0.0,1.9];
27 output level();
28 delay [0.0,1.0];
29
30 output intDelay();
31 delay [0.0,1.0];
32 output grasp();
33 delay [9.0,9.9];
34 output level();
35 delay [9.0,9.9];
36 output level();
37 delay [9.0,9.9];
38 output level();
39 delay [0.0,9.9];
40 output release();
41 delay [0.0,1.9];
42 output level();
```

Figure 5.3: Trace for `TraceAdapter` to exercise `intDelay` part of the testsuite model.

otherwise test terminates.

The trace in Figure 5.2c examines five variations of running `one2many` test (variations are separated by an empty line). Similarly the trace in Figure 5.3 covers four variations of `intDelay` test run, except that there is only one environment process involved.

So far we showed traces for positive tests. Next, Figure 5.4 show samples for negative test. For example, the trace in Figure 5.4a on line 5 delays only up to < 9 model time units and then outputs `level` which actually violates the guard(s) that do not allow `level` outputs before 9 time units elapsed after `grasp` event, thus TRON should report it as test failure. Similarly traces in Figures 5.4b, 5.4d are not allowed in the test suite model, but the fault is

deducable only at the second to last command. The trace in Figure 5.4c is slightly different because it generates variable value 10 that is not allowed (only values from 0 to 9 are allowed) in `send` test. The last long delay is append so that `TraceAdapter` would wait for verdict and not exit prematurely.

```
1 delay [0.0,1.0];
2 output intDelay();
3 delay [0.0,1.0];
4 output grasp();
5 delay [0.0,8.9];
6 output level();
7 delay 100.0;
```

(a) Guard.

```
1 delay [0.0,1.0];
2 output intDelay();
3 delay [0.0,1.0];
4 output grasp();
5 delay 100.0;
```

(b) Invariant.

```
1 delay [0.0,1.0];
2 output send(10);
3 delay 100.0;
```

(c) Data.

```
1 delay [0.0,1.0];
2 output intDelay();
3 delay [0.0,1.0];
4 output grasp();
5 delay [0.0,6.9];
6 output release();
7 delay 3.0;
8 output level();
9 delay 100.0;
```

(d) Non-determinism.

Figure 5.4: Trace for `TraceAdapter` to exercise test failures.

5.1.3 Results

We created 12 non-deterministic trace fragments (151 lines in total) for positive tests and 5 short traces (32 lines in total) for negative tests.

The positive test traces are concatenated in a loop by a shell script and fed to `TraceAdapter`, which created lengthy test sequences running for full duration of 10000 model time units. The tests are repeated with various TRON delay choice options: `lazy`, `random` and `eager`. All three test runs passed TRON test with a lot of time randomization.

The negative test traces are very short (and executed fast), thus in order to create additional timing randomization the tests are repeated 1000 times. All test runs finished with `test failed` verdict.

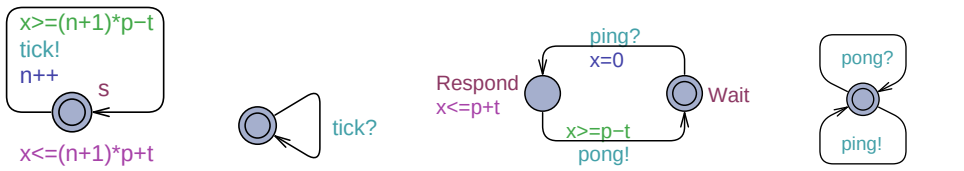
We conclude that TRON faithfully emulates and monitors most popular UPPAAL modeling features and test suite can serve as a regression test for `Uppaal` features. There are still time precision issues such as test failure may slip undetected for up to 1 model time unit due to time offset (e.g. test starting with `instant`), we examine them more closely in the next experiment.

5.2 Benchmarks

We use benchmarks experiments to examine various aspects of TRON's timed behavior. We run experiments on a regular laptop with Intel Core 2 Duo 2.2GHz CPU, Linux kernel 2.6.27.6, using round-robin scheduler with priority 21 (highest non-real-time), with default time quantum of 0.1s. It is notable that Linux had many CPU scheduler improvements since version 2.6.23 (October 2007) and has been matured to provide guarantees on CPU allocation to ready threads within $1ms$ in average and within $10ms$ in worst case. The experiments are run in normal desktop usage setting, where all auxiliary tasks are mostly idle.

5.2.1 Time Accuracy

The purpose of this benchmark is to measure the time accuracy of inputs issued by TRON. We setup an environment model shown in Figure 5.5a, a simple model for IUT shown in Figure 5.5b and run online test against implementation which records the timing of each `tick` arrival. The model uses constant values `p=250,`



(a) `Ticker`. (b) `TickObserver`. (c) Ping-pong env. (d) Ping-pong IUT.

Figure 5.5: Models for measuring time accuracy and responsiveness of TRON.

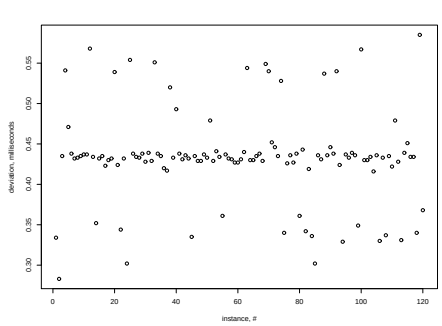
`t=50` and test is setup for $1000\mu s$ duration of one model time unit. Thus the ticks should arrive with a period of $250 \cdot 1000\mu s = 250ms$ and may appear within $\pm 50 \cdot 1000\mu s = \pm 50ms$ offset. TRON has four different options for controlling the choice of input timings: `eager`— send input as soon as possible, `lazy`—as late as possible, `random`—randomly within model bounds and bounded interval which is the same as `random` but with explicit upper bound (to avoid choosing arbitrary large delays).

TRON is run with the options `-F 400 -l 1000`, with three different variants of `-P` option: `eager`, `random` and `lazy`. After test run we compute the difference between actual tick arrival and earliest expected $(250 \cdot n - 50)ms$ for each input instance $n \in [0, 119]$. The results are plotted in Figure 5.6. Figure 5.6b shows that in `eager` setting TRON delivers input always within $0.6ms$. Figure 5.6f shows that TRON is delayed at least until $99.25ms$ and at most until $99.60ms$, i.e. it never exceeds 100 model time units and is slightly early by at most $0.75ms$ which is within 1 model time unit. Figure 5.6c shows that the inputs are scattered anywhere with deviation between 0 and $100ms$ as dictated by the model between 0 and 100 model time units. From above we conclude that it is possible to schedule inputs within reasonable bounds of 1 model time unit and overall timing is disturbed by at most $0.6ms$ (with eager setting) and by $0.75ms$ in worst case (lazy setting). The extra disturbance in lazy setting can be explained by delay option `-l 1000` which tells TRON that input may potentially be delayed by $1ms$ and thus it is safer to choose earlier timings to avoid violating upper bound.

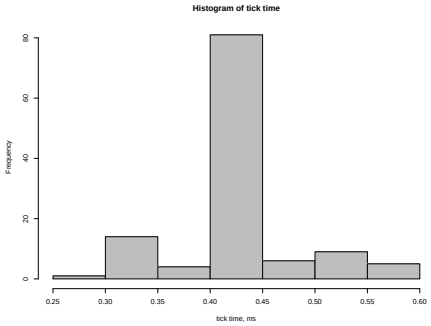
We conclude that for simple models, TRON is able to deliver inputs at specific timing dictated by the environment model, given that underlying OS has some real-time guarantees which actually even exceeded our expectations by $0.5ms$ as opposed to $1ms$ promise.

5.2.2 Impact of Time Discretization

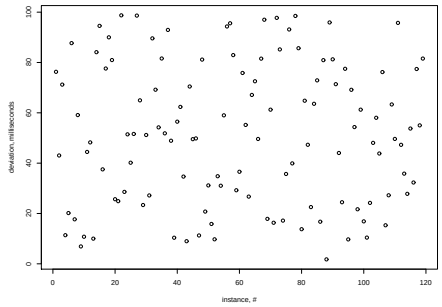
The last experiment showed that TRON is able to generate inputs at specific timing controlled by the model, however we know that TRON uses model clock as a reference to global time. The model clock has integer precision and hence all timings may be based on integer offset. In this experiment we measure TRON



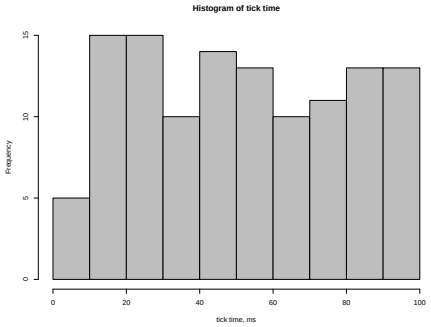
(a) `eager` instances.



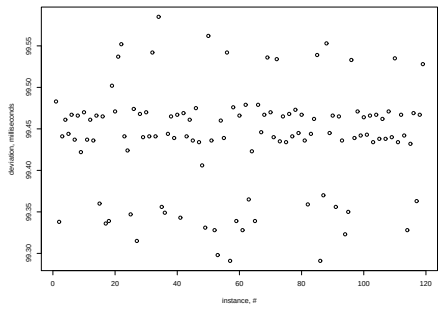
(b) `eager` distribution.



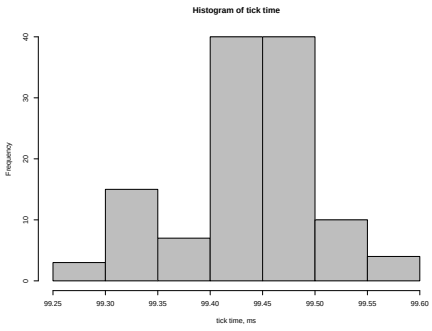
(c) `random` instances.



(d) `random` distribution.



(e) `lazy` instances.



(f) `lazy` distribution.

Figure 5.6: Accuracy of input generation: `tick` timing deviation from period offset.

response to events that are not based on integer offset. We use special IUT to generate `ping` outputs at periods of 410ms and randomized within 100ms time instances (with 48bit nano second randomization). IUT expects `pong` input as response from TRON after 200ms within 100ms and records the timing of `ping` and `pong`. TRON is instrumented to use the system model shown in Figure 5.5c and 5.5d as test specification with constants `p=250, t=50` and time unit of 1000μs.

We use `-P eager` option to force TRON to choose input timing as early as possible, i.e. at around $200ms$. Each ping-pong timing pair is treated as an independent measurement (where time of `ping` is randomized). Then we measure the time difference between each individual `ping` and `pong`.

Figure 5.9 displays `ping` and `pong` timings and their differences. The timing of each event instance n is normalized by subtracting $n \cdot 410ms$, hence each dot appears as a separate measurement aligned with others: all `ping`s are within first 100ms, `pong`s are between 200 and 300ms (approximately by 200ms later than a corresponding `ping`) and the computed timing difference between each corresponding `pong` and `ping` is within 199.8 and 200.4ms.

Figures 5.7b and 5.8b show that timing of `ping` and `pong` is distributed approximately uniformly and the time differences in Figure 5.9b are similar to a normal distribution with many instances lying around mean value of 200.15ms. Student's t-Test (produced by [53]) reveals that 95% confidence interval for difference is $[200.133; 200.153]ms$ and all differences are within $[199.5; 200.3]ms$. Moreover, linear model analysis [17] (summary in Table 5.1) says that linear dependency coefficients of `pong` timings on `ping` timings are $1.000 \pm 1.8 \cdot 10^{-4}$

```
Residuals:
      Min        1Q    Median        3Q       Max
-0.191449 -0.037229  0.001785  0.032446  0.153486

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.001e+02  9.666e-03   20705   <2e-16 ***
ping        1.000e+00  1.789e-04    5590   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.0713 on 192 degrees of freedom
Multiple R-squared:     1,Adjusted R-squared:     1
F-statistic: 3.125e+07 on 1 and 192 DF,  p-value: < 2.2e-16
```

Table 5.1: Linear model analysis produced by R [53].

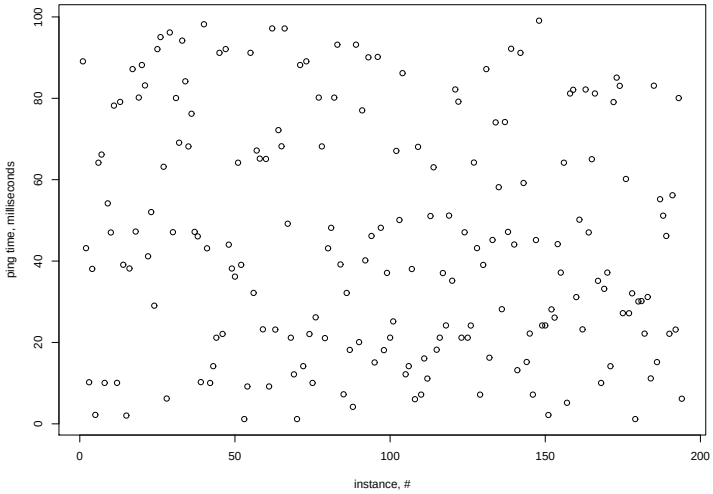
and $200.137 \pm 9.7 \cdot 10^{-3}$, i.e. the relation between `pong` and `ping` timings can be expressed as $t_{pong} = 1.0 \cdot t_{ping} + 200.137ms$ with standard error $\pm 0.0713ms$. Figure 5.10a shows the linear dependency between `ping` and `pong` times and Figure 5.10b shows residual distribution against fitted values of `pong` times. There is no structure in residual distribution, hence the `ping` timings are well randomized and results from linear model analysis are valid.

We conclude that inputs are only slightly delayed in most cases (within $0.3ms$ in worst case), but response times are not influenced by model clock integer discretization and TRON is able to provide input independently from timing offset.

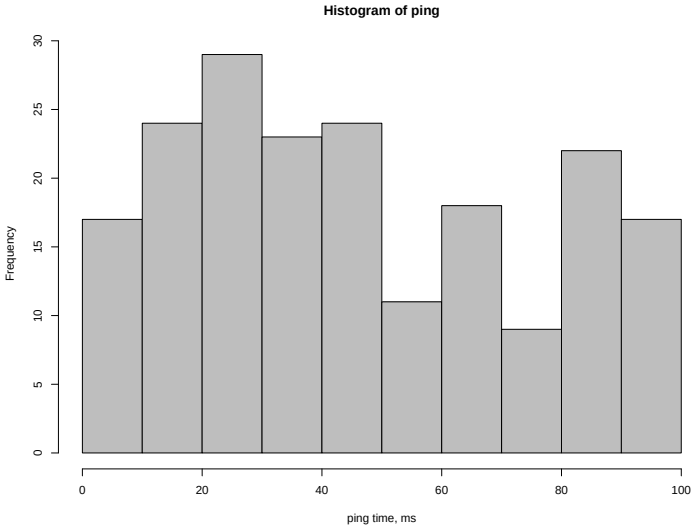
5.2.3 Minimal Reaction Time

In this experiment we measure the minimal TRON reaction time from output detection to issuing immediate input. The test stresses the CPU scheduler as well as computations in UPPAAL engine and gives the most optimistic estimate of TRON reaction on a common computer.

We reuse the test setting from previous experiment in Section 5.2.2, except that the environment model in Figure 5.5c has urgent location `Respond` instead



(a) Instances.



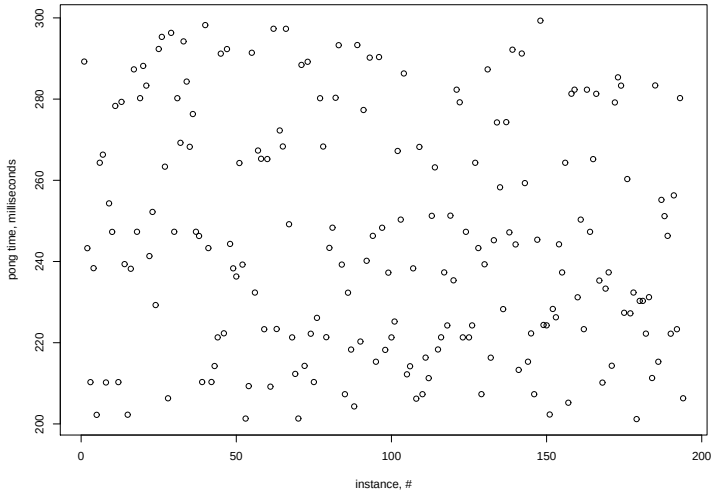
(b) Distribution.

Figure 5.7: Ping times.

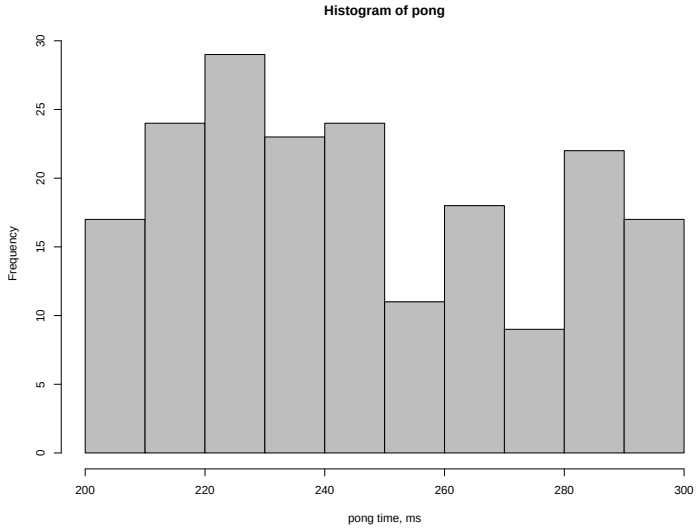
of invariant, meaning that the tester should issue `pong` input immediately after it senses `ping` output.

Figure 5.11 shows the distribution of time differences between individual `ping` and `pong` events. The reaction time is between $0.1ms$ and $0.5ms$, the average is $0.366ms$ and the 95% confidence interval from Student's t-Test is $[0.358; 0.373]ms$.

We conclude that TRON can be used to schedule inputs with up to $0.5ms$ reaction time at best.



(a) Instances.

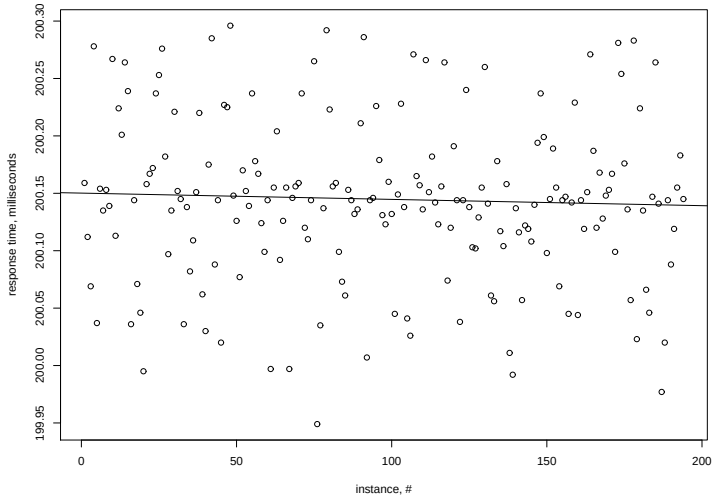


(b) Distribution.

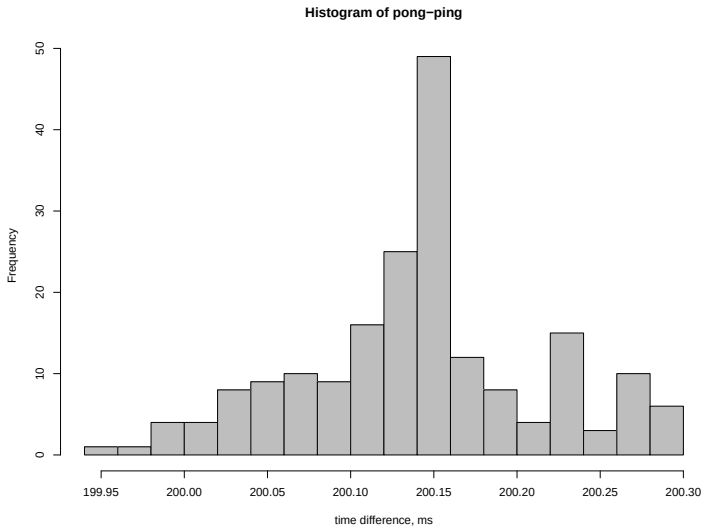
Figure 5.8: Pong times.

5.2.4 Scalability

The goal of this experiment is to determine how online test performance scales based on the size of a system model. We use a train gate model from UPPAAL demo examples, originally published in [64]. We used a variation of this example before in [42] for mutant study and performance benchmarks. In this experiment the model is adapted to completely asynchronous setting where the outputs from gate controller are separated from inputs arriving from trains. The original



(a) Pong-ping time difference.

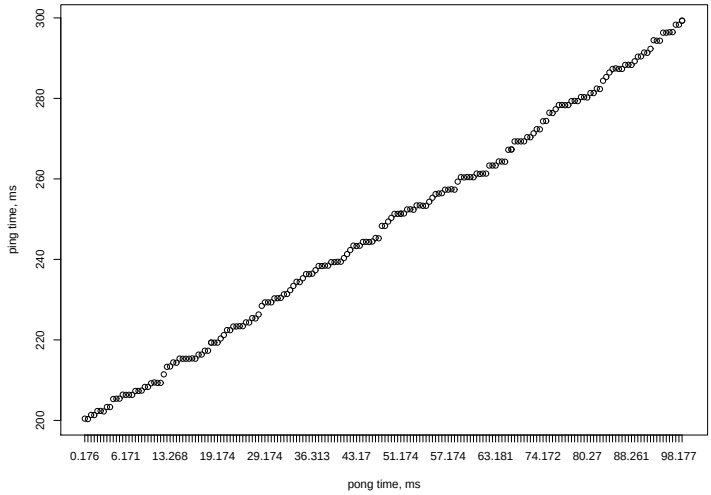


(b) Pong-ping time difference distribution.

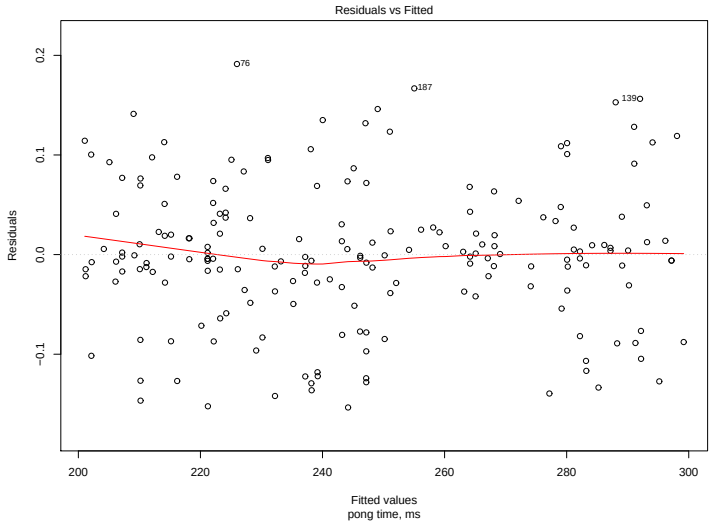
Figure 5.9: Ping-pong times with `-P eager` option.

model assumes that the inputs are sensed by gate controller immediately hence trains and gates are in perfect synchrony. We can no longer assume this in latest TRON test setup where inputs and outputs travel independently and may interleave in any order. Moreover we have to make sure that gate controller is always input enabled. The resulting model is shown in Figure 5.12.

Train model shown in Figure 5.12a (same as in [64]). The model specifies that train may approach the crossing by moving from location `Safe` to `Appr` and after 10 time units may enter the crossing by moving to location `Cross`. While



(a) `ping` vs. `pong` times.

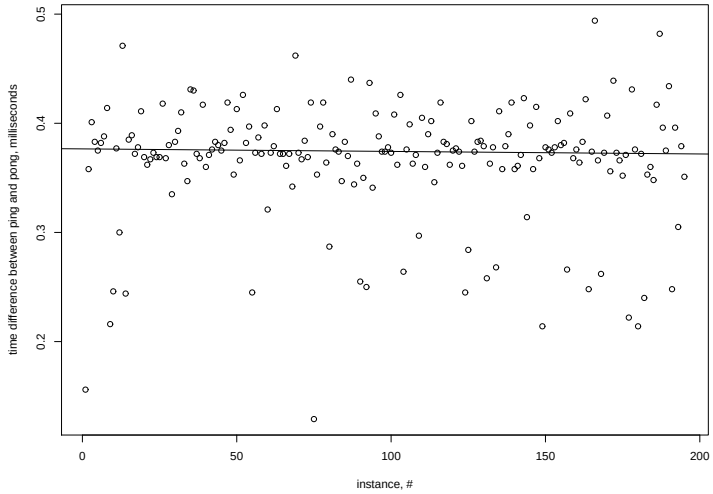


(b) Residuals vs. fitted `pong` times.

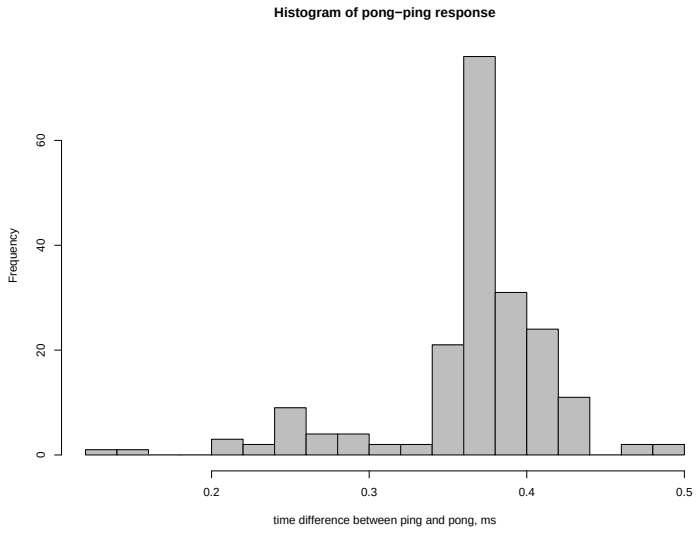
Figure 5.10: Visualization of linear model analysis.

train is approaching, it may receive a signal `stop` from the gate controller within 20 time units and hence it would move to `Stop` location. Once the train is in `Stop` it need explicit signal `go` to move to `Start`. Finally, train may leave and free the crossing from location `Cross` by issuing `leave`. The are N instances of trains created in the system model.

Gate model shown in Figure 5.12b (changed radically). The model still maintains FIFO queue of trains. The controller may be in a location `Opened` where trains are silently allowed to go through, `Closed` where trains are stopped and



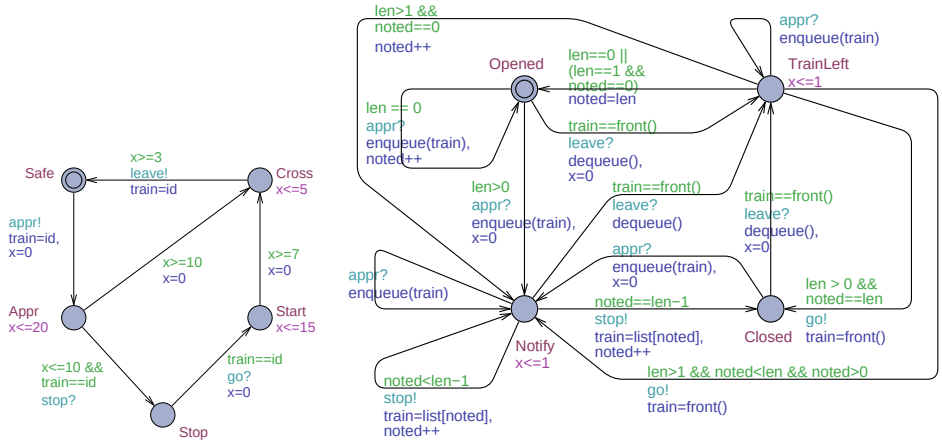
(a) Pong-ping time difference.



(b) Pong-ping time difference distribution.

Figure 5.11: TRON reaction: time difference between `ping` and immediate `pong`.

gate is idly awaiting for one of the trains to leave the crossing. In location `Notify` gate controller is required to issue `stop` signals to additional trains within 1 time unit and when train leaves the controller goes to location `TrainLeft` where it should let go the first train in the queue by sending signal `go`. If the queue becomes empty (length of the queue is encoded by variable `len`), the gate controller comes back to location `Opened`. Since trains can arrive in any order at any time (controller is input enabled), the gate controller also maintains information which trains have already been issued a signal `stop` by maintaining additional



(a) `Train`. (b) `Gate`.

(c) `InpAdapter`. (d) `OutAdapter`.

Figure 5.12: UPPAAL Timed automata models of train-gate system.

index to the queue `noted`. Listing 5.3 shows local gate controller declarations containing functions used in the model.

We also add adapter processes to model the input signal delay shown in Figure 5.12c. There are N instances of `InpAdapter` for `appr` and N instances for `leave` signals to ensure all trains can arrive at the same time and signals may interleave on the way. The adapter instances are ordered by global variable `order` to ensure that only one instance at a time will be used (partial order reduction), because all instances are equivalent and we would like to avoid unnecessary non-determinism. We use `OutAdapter` for outputs just to transfer the train ID from gate controller to individual train. In real test setting outputs should also delay the signal like `InpAdapter`, but delaying inputs is enough, and we use a simple abstraction of outputs.

The system model is instantiated by declarations in Listing 5.4.

TRON is instructed that inputs are `appr(envTrain)` and `leave(envTrain)`, outputs are `stop(envTrain)` and `go(envTrain)`, hence the system model is partitioned into model of environment consisting of `Train`s and requirements for IUT consisting of `Gate`, `InpAdapter` and `OutAdapter`.

The resulting model also satisfies all the properties examined by [64]. The most notable for us is that the model is deadlock free, moreover `Gate` is input enabled with respect to assumptions on `Train`s.

We run online test for 100000 model time units in virtual time (model time unit is set to $10ms$) for each instance of $N \in [1, 24]$ trains and measure the amount of resources consumed by a complete online test run. The measurements are displayed in Figure 5.13. The figures show that the CPU time usage and

```
1  clock x;
2  id_t list [N+2];
3  int [0, N+1] len, noted;
4  /** Put an element at the end of the queue */
5  void enqueue(id_t element) { list [len++] = element; }
6  /** Remove the front element of the queue: */
7  void dequeue() {
8      int i = 0;
9      len--;
10     while (i < len) list [i++] = list[i + 1];
11     list [i] = 0;
12     noted--;
13 }
14 /** Returns the front element of the queue: */
15 id_t front() { return list [0]; }
```

Listing 5.3: Gate model declarations.

memory consumption grow exponentially when the number of trains is increased. It can be explained by the fact that the complexity of a model also increases rapidly and there are many more states to keep track of. We can compare state space sizes by UPPAAL verification: it takes 18.1s and 39MiB to verify deadlock freeness for $N = 3$ instance and far more than 20min and 1.85GiB for $N = 4$ instance (verification did not complete). On the other hand, only the environment model complexity is increased, which means that TRON may choose to maintain only particular environment choices, whereas current TRON implementation tracks all of them.

We conclude that the online test performance degrades exponentially in the number of parallel processes in the model, but slow down is not as extreme as in case of UPPAAL verification of entire state space. There is also room for optimizations in computing state set estimates when environment model transitions are executed. Next, we examine how individual state set estimation functions perform.

5.2.5 Performance

The goal is to measure the performance of symbolic operations in UPPAAL engine during online test. We use the same model as in previous experiment only with a single instance of a model with 24 trains (`const int N = 24`). There are mainly two operations performed by UPPAAL engine: `AfterDelay` and `AfterAction`. `AfterDelay` computes the state set estimate when a time delay is observed. `AfterAction` computes the state set estimate when an input or output action is observed. Usually the operations are applied in alternating fashion, except for a few instances of subsequent `AfterDelay` operations when TRON decides to wait repetitively (which is minimized by large argument to `-F` parameter). We take wall-clock time stamp before and after operation and record the time difference the operation takes and the state set size before operation (as a measure of input complexity for the algorithm).

Figures 5.14a and 5.14b show the distributions of state set sizes during online

```
1  const int N = 3;          // # trains
2  typedef int [0, N-1] id_t;
3  meta id_t envTrain, iutTrain;
4  chan appr, stop, leave;
5  chan go;
6  const int INPMAX = N;
7  typedef int [0, INPMAX-1] input_t;
8  chan apprIUT, stopIUT, leaveIUT;
9  chan goIUT;
10 input_t apprOrder, leaveOrder;
11
12 train(const id_t id) = Train(id, envTrain);
13 gate = Gate(iutTrain, apprIUT, leaveIUT, goIUT, stopIUT);
14 ApprAdapter(const input_t id) = InpAdapter(id, 1, appr, envTrain, apprIUT,
       iutTrain, apprOrder);
15 LeaveAdapter(const input_t id) = InpAdapter(id, 1, leave, envTrain,
       leaveIUT, iutTrain, leaveOrder);
16 GoAdapter = OutAdapter(goIUT, iutTrain, go, envTrain);
17 StopAdapter = OutAdapter(stopIUT, iutTrain, stop, envTrain);
18
19 system train, gate, ApprAdapter, LeaveAdapter, GoAdapter, StopAdapter;
```

Listing 5.4: Global declarations and instantiation of train-gate model.

test, these are the inputs to `AfterDelay` and `AfterAction` algorithms. Note that the vast majority of state sets are small and there are larger state sets for `AfterAction` than for `AfterDelay` since they are a result of `AfterDelay` computations where uncertainty about current system state increases, while `AfterAction` has an opposite affect that TRON determines the state more precisely due to additional information from observed I/O. Figures 5.14c and 5.14d show individual instances of CPU time measurements for each state set size. Figures 5.14e and 5.14f show the computed means of the same measurements for each state set size. Note that there is a linear CPU usage tendency towards the line computed by linear model analysis by R [53], and performance is hardly predictable at all when state set sizes are large (very few measurements available). On the other hand, the worst case CPU time consumption on 400 states is about $0.5s$ which is acceptable for many interactive systems.

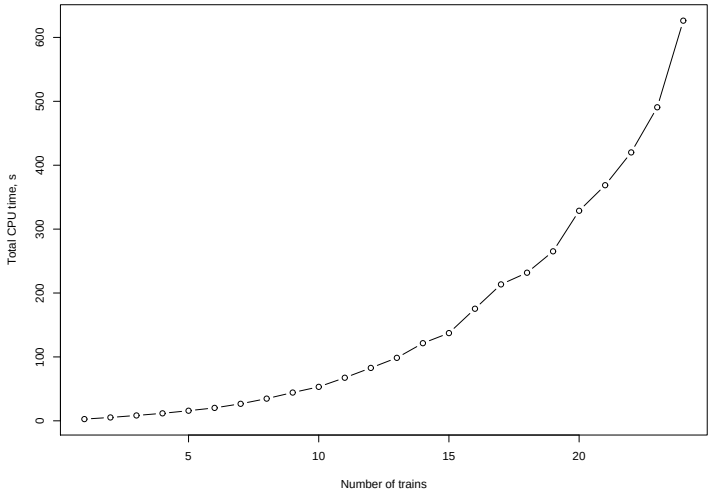
5.3 Code Coverage Experiment

The goal is to examine how much of the implementation code is exercised when stimulated by online test.

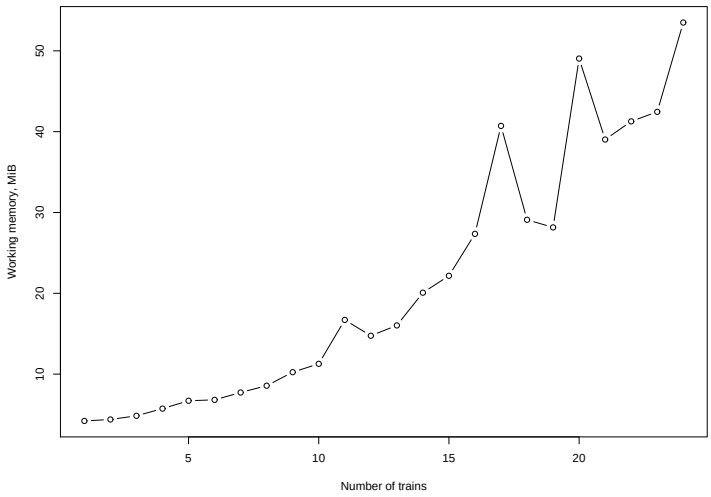
We use smart lamp controller example to experiment with TRON tests against Java implementation.

5.3.1 Smart Lamp Model

The test specification consists of smart lamp system model shown in Figure 5.15:



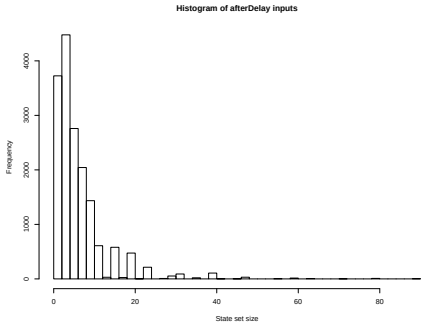
(a) User CPU time.



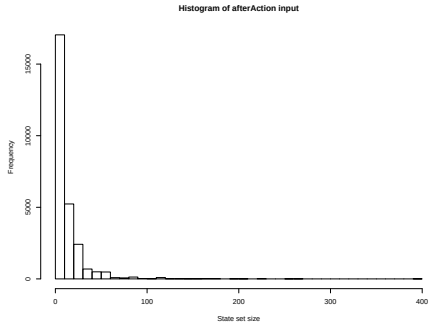
(b) Maximum working (resident) memory usage.

Figure 5.13: Resources used by online tests with various model sizes.

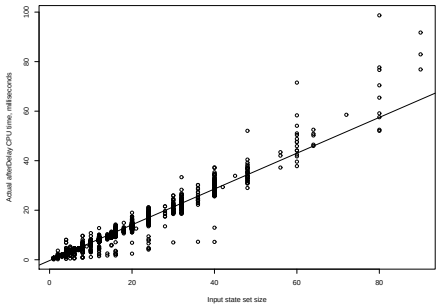
`Interface` accepts sequences of `grasp` and `release` inputs and translates them into `touch`, `startHold` and `endHold` signals based on timing relation between subsequent `grasp` and `release`. If `grasp` and `release` happen within short `epsilon` time then it is ignored. If the time difference is between `epsilon` and `delta` then a `touch` is registered. If time difference is longer than `delta` then `startHold` is issued and `endHold` is issued upon `release`. The timing is relaxed by a constant `tolerance` which makes timing requirements more realistic by allowing some behavior non-



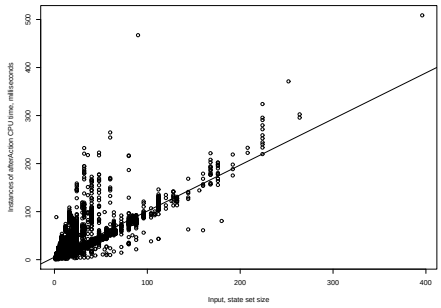
(a) State set sizes for `AfterDelay`.



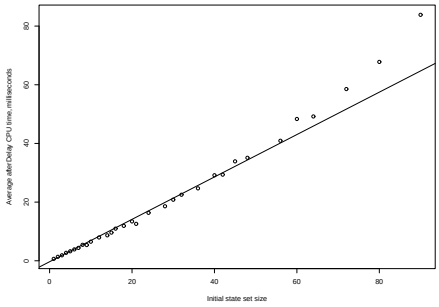
(b) State set sizes for `AfterAction`.



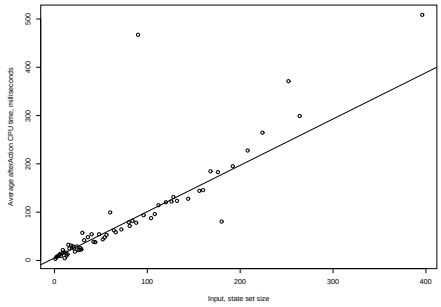
(c) CPU usage by `AfterDelay`.



(d) CPU usage by `AfterAction`.



(e) Average CPU usage by `AfterDelay`.



(f) Average CPU usage by `AfterAction`.

Figure 5.14: State set sizes and CPU usage during online test with 24 trains.

determinism.

`Switch` consumes `touch` signals and switches the light on and off. The light level is remembered in variable `OL` so that it is restored when the light is turned on again.

`Dimmer` reacts to `startHold` and `endHold` and moves between locations: `PassiveUp` idly waits for `startHold` and then moves to `Up` where the light level `L` is increased with `delay` time steps until `endHold` is received. `PassiceDn`

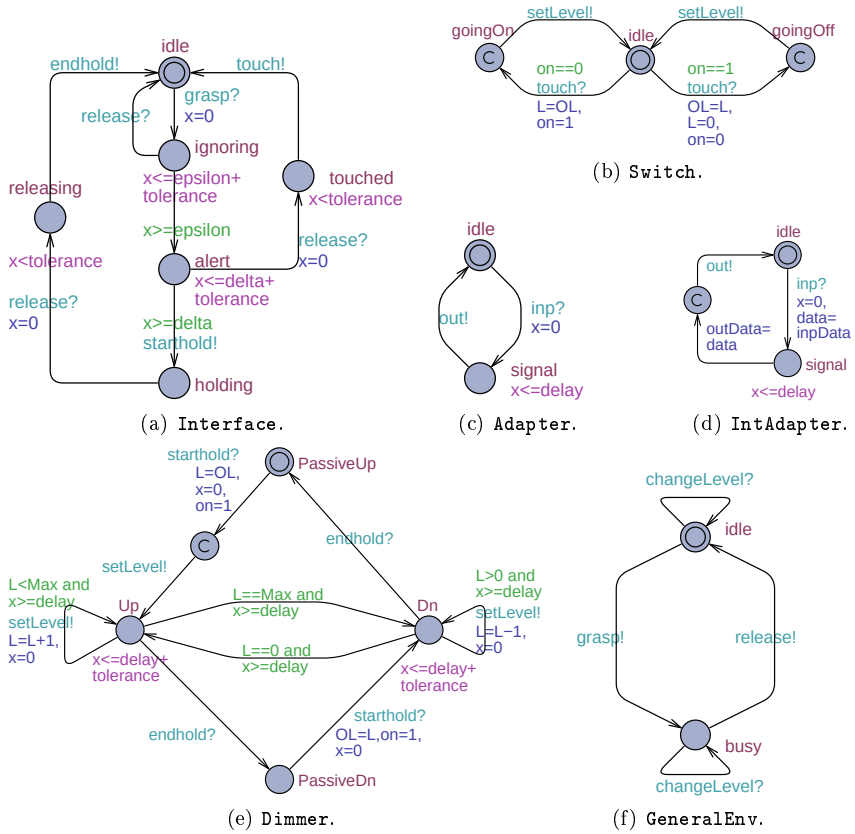


Figure 5.15: Smart lamp timed automata model.

and `Dn` are equivalent to `PassiveUp` and `Up` except that the light level is decreased instead of increased. `Dimmer` may also move between `Up` and `Dn` when extreme light level values are reached.

`GeneralEnv` is a model of a user which can produce alternating sequences of `grasp` and `release` and observe the changes in light level via `changeLevel`.

`Adapter` is a model for test adapter delaying the input signals by at most `delay` time units (it is a different parameter than `delay` in `Dimmer`).

`IntAdapter` is a model for test adapter delaying the outputs signals one integer data by at most `delay` time units.

The system model is instantiated by declarations shown in Listing 5.5.

5.3.2 Code Coverage Tool

We use EMMA tool which instruments Java byte code on-the-fly (upon Java class loading) with coverage counters. EMMA gives statistics on *basic block* coverage. *Basic block* is a sequence of bytecode instructions without any jumps or jump targets, i.e. basic block is executed as one atomic unit (if exceptions are not thrown). Several Java source lines can be within the same basic block. The

```
1  const int Max = 10; // max level of light
2  const int tolerance = 5; // max timing tolerance
3  const int epsilon = 20; // timeout when grasp cannot be ignored
4  const int delta = 50; // longest duration for registering touch
5  const int delay = 100; // dimmer increment/decrement delay
6  const int latency = 5; // adapter communication latency
7  const int Wait = 2000; // used by environment
8  const int T_react = 1; // used by environment
9  // IUT internal:
10 chan touch, starthold, endhold;
11 int [0,1] on;
12 int iutLevel, OL;
13 // IUT interface to adapter:
14 chan setGrasp, setRelease; // inputs
15 chan setLevel; // outputs
16 // Observable
17 chan grasp, release, level;
18 int envLevel;
19
20 // IUT part:
21 interface = Interface(epsilon, delta, setGrasp, setRelease);
22 dimmer = Dimmer(delay, setLevel, iutLevel);
23 switcher = Switch(setLevel, iutLevel);
24 // Env part:
25 user = GeneralEnv(level, envLevel);
26 // communication latency adapters:
27 graspAdapter = Adapter(latency, grasp, setGrasp);
28 releaseAdapter = Adapter(latency, release, setRelease);
29 levelAdapter = IntAdapter(latency, setLevel, iutLevel, level, envLevel);
30
31 system interface, switcher, dimmer, user, graspAdapter, releaseAdapter,
       levelAdapter;
```

Listing 5.5: Global declarations and instantiation of smart lamp model.

basic block is treated as covered when the last instruction is executed. EMMA developers claim that basic block coverage is more reasonable than sheer line coverage as it disregards comments and it is finer grained in a sense that 100% basic block coverage implies 100% executable line coverage.

5.3.3 Results

Table 5.2 shows the coverage statistics on smart lamp source code produced by EMMA after TRON test.

Visual inspection of coverage-highlight source code revealed that thread interrupt exception handling, some of thread startup code and some `break` statements are not covered. It is normal that exception handling is not exercised as it is never used nor tested (e.g. there is no special test input to trigger application termination). Thread startup code depends on thread scheduling during initialization, thus it is also normal that not all possible initialization cases are exercised after just on test run. The coverage of `break` statements are

Method	Basic block Coverage	
`SmartLamp.java` (`Interface` functionality)		
SmartLamp	100%	(25/25)
handleGrasp	69%	(47/68)
handleRelease	75%	(63/84)
run	60%	(99/164)
Total:	69%	(234/341)
`DimmerM0.java` (`Dimmer` and `Switch` functionality)		
DimmerM0	100%	(18/18)
handleStartHold	76%	(65/86)
handleEndHold	66%	(40/61)
handleTouch	79%	(38/48)
run	77%	(150/194)
setLevel	65%	(26/40)
Total:	75%	(337/447)
Total:	72%	(571/788)

Table 5.2: Smart lamp code coverage after online test.

somewhat mysterious as they are exit points of (covered!) `switch` branches and some `break` statements are not considered as coverable at all.

We conclude that a fairly large portion of source code is exercised and no important functionality is left out, however this does not imply anything about the correctness of the code, hence we devise next experiment in the following section.

5.4 Mutation Experiment

Mutant is a (slightly) modified (mutated) object under test. The purpose of mutation testing is to evaluate the quality of test suite by examining whether test suite is capable of detecting the mutation change(s) in the object.

In our setting we evaluate TRON's online test ability to identify mutants by issuing different test verdicts. We pick Jester [49] as a mutant generation tool. The advantage of using external tool over the mutant study described in [42] is that mutants are generated automatically in vast quantities and mutations are independent of developer's (our) bias. We reuse the smart lamp model and Java implementation described in Section 5.3.

5.4.1 Jester

Originally Jester [49] was created as a testing tool for JUnit tests working on Java source code, but its setup is flexible enough to run any test tool, including TRON. Jester is instructed to modify a set of Java source files, compile and run test on each of them. Jester has a set of mutation rules similar to find-and-replace functionality of text editor. It searches a source code for rule match and applies the rule by replacing the found string producing a source file mutant. The mutation procedure is applied only once per one mutant and changes of previous mutations are discarded. Once the source mutant is produced, Jester

tries to run a test script which attempts to compile the modified sources and run the test suite. If compilation or some test fails then Jester treats the mutant as being detected by the test suite. Alternatively, if all tests pass, the test suite prints a string "TEST PASSED" which is recognized by Jester. Jester then records the result and the applied change and moves on to a next mutation.

The mutation rules are in the form of `%string1%string` which means that `string1` is to be replaced by `string2`. Listing 5.16 shows the rules Jester uses to create mutations. Rules 1-10 are provided by default and rules 11-24 are added

```
1  %true%false
2  %false%true
3  %if(%if(true␣||
4  %if␣(%if␣(true␣||
5  %if(%if(false␣&&
6  %if␣(%if␣(false␣&&
7  %==%!=
8  %!=%==
9  %++%--
10 %--%++
11 %+=%-=
12 %-=%+=
13 %-%+
14 %+%-
15 %*%/
16 %/%*
17 %<%<=
18 %<%>
19 %>%>=
20 %>%<
21 %<=%<
22 %<=%>=
23 %>=%>
24 %>=%<=
```

Figure 5.16: Jester mutation rules.

by us. In addition to rules, Jester implements "modifying literal numbers", the result is that the first digit of a number is incremented. Table 5.3 shows example mutations.

Original code	Rule	Mutated code
`if (a==b) a++;`	4	`if (true || a==b) a++;`
`if(a==b) a++;`	5	`if(false && a==b) a++;`
`if (a==b) a++;`	7	`if (a!=b) a++;`
`int delay = 500;`	incr.	`int delay = 600;`

Table 5.3: Example rule applications in Jester mutant generation.

From the rules above, it can be seen that Jester mutations are simple and naive text replacements. This is an advantage to create many mutants cheaply, however apart from compiler errors, it may also lead to deadlocks and even infinite loops in the implementation. Thus we created a script that checks the test progress and it would terminate IUT if the test is still running after 40 seconds assuming that it has locked up in busy loop or deadlock. Normally one test run takes up to 20s at most so no good behavior is terminated. Jester records such termination as a test failure, i.e. as if test suite has detected mutant.

5.4.2 Results

Jester is applied on the smart lamp example, namely the two files responsible for `Interface`, `Dimmer` and `Switch` functionality. The online test is run in virtual time to reduce risks of spurious test failures due to soft-real-time OS scheduling. The results are summarized in Table 5.4.

5.4.3 Discussion

There are 32 mutants that passed the online test. 19 of them are rtioco - conforming and hence are not detected by TRON. We describe them below:

Rules	Mutations detected by			Mutations Passed	Total
	Compiler	Lockup	TRON		
1-10	0 (0%)	9 (15.0%)	27 (45.0%)	24 (40%)	60
11-24	26 (61.9%)	0 (0%)	8 (19.0%)	8 (19.0%)	42
1-24	26 (25.5%)	9 (8.8%)	35 (34.3%)	32 (31.4%)	102

Table 5.4: Mutant detection results.

Timing mutations are changes in the value of timing constants, that made `Dimmer` to report light level changes by 100ms later than original values. Such delays are not detected because the model allows 5 model time units (mtu) for input and output communication latency (optimized for world-time tests), and 5mtu more for timing tolerances, thus allowing implementation potentially to be late by 150ms in total. We made additional online test runs with smaller values of adapter delay and tolerance in the model, and all tests failed on such mutants, thus it can be considered as a flexibility of non-determinism in the test specification.

Debug mutations are within code that dealt with debug messages. Some parts of the code is turning on or off the debug messages depending on the environment variables, some parts are issuing messages depending on whether the debug mode is turned on, and other parts print derived timing information. Obviously such code has no influence on the behavior observed by TRON and hence no difference detected.

Superfluous code mutations are within additional conditions that are always true and Jester reported that rule 4 mutants are not detected. Such dead code is not obvious at local inspection of the code and was added for education exercises.

Redundant assignment initializes the light level which apparently is always overwritten with a value of old light level upon first interaction, and hence Jester's change of initialization value is not detected. The initialization code mutation came as a (pleasant) surprise, but nevertheless such code should be present in case the implementation is changed in the future and the initial value is not overwritten.

Leftover code are remnants from an older virtual thread API which required that timeouts in timed-wait functions were absolute. The API has been changed to be consistent with Java interface, but expressions calculating the absolute time value were still left, and some of them are used in debug messages. The original rules (1-10) do not mutate this code, but our additional rules do, and naturally TRON does not detect the change.

Jester also revealed 13 mutations that do change the behavior but are not detected by online test, we review all of them below:

Premature startup. Jester noted that mutations concerned with variable `alive` are not detected. `alive` reflected whether a thread has already been started and is still running. During startup, it is possible that operating system schedules IUT threads in such a way that it establishes the

connection and there is already an incoming input from TRON, but the `Dimmer` thread has not been scheduled yet. Such scenario would result in lost inputs and normally fail the test (the actual failures were reproducible on a rare occasion). Hence the implementation was instrumented to delay the test start until all threads had a chance to initialize by signalling `alive==true`. Naturally, such thread scheduling is very unlikely on multi-core architectures, it is independent from the tester and TRON was given only one test run with low chance of triggering it.

Abrupt termination code was added to gracefully terminate the application in case a thread or a program received request for interruption. In particular the presence of `try-catch` clauses for `wait` calls are required by Java compiler even though the thread interruption feature is not used. Mutations in such code are not detected as TRON is not instructed to terminate the application (e.g. inputs did not include "terminate" and adapter was not created accordingly), hence such code is never tested.

The distribution of undetected changes are summarized in Table 5.5.

Rules	Time	Debug	Cond.	Redundant	Left.	Start	Term.	Total
1-10	2	5	2	2	0	3	10	24
11-24	0	4	0	0	4	0	0	8
1-24	2	9	2	2	4	3	10	32

Table 5.5: Distribution of mutants which passed online test.

In addition, we also found a non-trivial mutant that comes from sloppy thread-condition programming, whose behavior depends on OS thread scheduling and is not detected reliably, however in such case we were able to create an environment model that stresses and eventually triggers the code at fault at will. The faulty code did not check the returned value of conditional-timed-wait method that was signalling if timeout was reached. The problem was that another thread could have changed the `Dimmer` state several times (by issuing sequence `grasp release grasp` without delays) before `Dimmer` thread is awaken in between and thus such state change would get lost leading to a test failure. Such mutant has been found in the early version of supposedly correct implementation of smart lamp application but due to its hideous nature it has been mostly undetected.

5.4.4 Conclusion

It is important to note that mutations are selected by external tool:

- Changes are independent from tester, thus no data spoofing is possible and there is a lot of possibilities for placebo effects.
- Some of the changes (in debug code, comments, various compiler errors) have little to no meaning and distorts greatly the statistics. Therefore the mutant generator could be more sensitive to program semantics.
- Changes are trivial text substitutions. The default Jester settings seems to aim to branch coverage (rules 3-6), and provides bare minimum not

to trigger compiler errors and avoid infinite loops. In effect, it is difficult to control the mutation process to achieve arbitrary coverage (e.g. `while` loop condition, invert value of a boolean variable). It is not possible to generate more complex mutants that would consider additional program state information, such as the return value of conditional timed-wait call and other concurrency aspects.

TRON identified a number (34.3%) of mutants directly and other (34.3%) were detected by compiler or lockup. The experiment confirms (partially) that TRON tests are sound by not issuing "failed" verdicts to conforming mutations, however there are 13 (12.7%) non-conforming mutations that are not detected. A close inspection of code revealed that 3 (2.9%) of passed mutations concern thread scheduling during startup and 10 (9.8%) are due to abrupt termination, which are reasonable findings given that TRON had little to no chance to detect them.

The mutant experiment does not reveal any faults or surprising behavior of TRON tests, and it does show new insights on smart lamp source code, reveals dead code and provides hints on potential timing errors, stresses the features of timing non-determinism in the model. Overall it has been a very positive experience.

5.5 Discussion

Most modeling features are implemented faithfully and the test suite is available to any new features to be implemented in future UPPAAL. Timing precision is limited by model time units as well as the guarantees of the execution platform. On a standard PC UPPAAL engine performance allows to schedule inputs for simple systems within $0.5ms$ and within $0.5s$ for as complex systems as described by 400 symbolic states at a time. IUT stimuli and fault detection capability proved to be very successful, only very rare thread scheduler-dependent bugs could have slipped through, but in given black-box system-level testing assumptions we could not expect better.

Chapter 6

Danfoss EKC Case Study

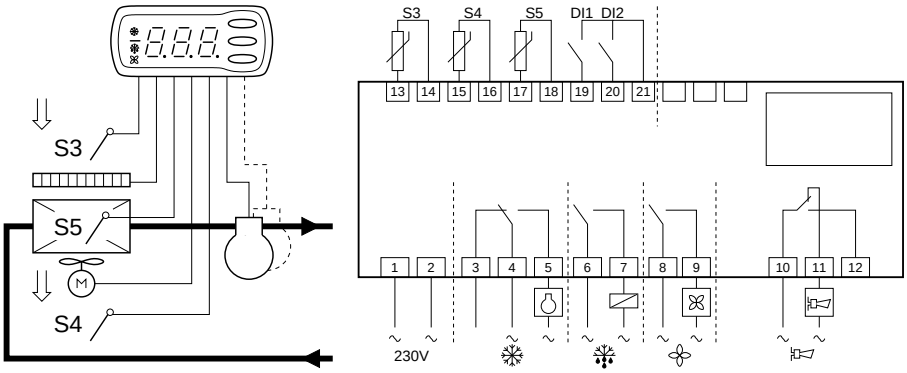
In this chapter we evaluate the applicability of TRON on an industrial product EKC (electronic cooling controller) from Danfoss A/S company in Denmark. This is a second iteration on a EKC product line since the old case study reported in [43]. In the first iteration we had difficulties with modelling the displayed temperature timely calculation. The resulting model contained too much non-determistic behavior due to allowed temperature deviation eventually resulting in more than 4000 states in a current state set, which bogged down the performance.

In this iteration we have a next generation controller which has higher precision temperature sensors, slightly different temperature calculation algorithm due to improved precision and improved test temperature injection mechanism which allows fairer testing conditions. In this study we provide a different approximation to temperature calculation which does not pose severe performance penalty. The resulting temperature modeling pattern can be generalized for piece-wise monotonic functions. In addition we managed to test fan relay and also interactions among defrost, compressor, fan and high temperature alarm.

Section 6.1 provides a brief summary of the product description from user manual. In Section 6.2 we provide motivation for repeating the case study on a new generation of devices. In Section 6.4 we show how to express the system-level requirements from user manual into UPPAAL timed automata network. The requirements are selectively extracted from the user manual under conditions where the source code and even product design documents were not available. Section 6.6 describes technical solutions used in order to connect to the implementation under test. Danfoss product engineers were available for comments and technical help during system modelling and adaption for testing. Section 6.8 summarizes the lessons learned in this case study.

6.1 The Refrigeration Control

Figure 6.1a shows typical setup how the devices are typically placed during the operation, where `S3`, `S4` and `S5` are temperature sensors. `S5` is placed on evaporator, `S3` is placed before defrost heater on evaporator and `S4` is placed after the air-flow from evaporator. The air is moved through evaporator with fan motor `M`. The refrigeration fluid is pumped to the evaporator by a compressor



(a) Typical placement of devices. (b) Connections for sensors and relays.

Figure 6.1: Schemes from EKC204A temperature controller manual.

Parameter		Values			
Function	**Code**	**min**	**max**	**factory**	**actual**
Temperature (set point)	SP	$-50°C$	$50°C$	$2°C$	$2°C$
Differential	r01	$0°$	$20°$	$2°$	$2°$
Manual service, stop regulation, start regulation	r12	-1	1	0	0, 1
Delay of temperature alarm	A03	$0min$	$240min$	$30min$	$16min$
Delay of temperature alarm after defrost	A12	$0min$	$240min$	$90min$	$20min$
High temperature alarm limit	A13	$-50°C$	$50°C$	$8°C$	$7°C$
Low temperature alarm limit	A14	$-50°C$	$50°C$	$-30°C$	$-2°C$
Thermostat signal for alarm (0%=S3, 100%=S4)	A36	0%	100%	100%	100%
Compressor minimum ON-time	c01	$0min$	$30min$	$0min$	$5min$
Compressor minimum OFF-time	c02	$0min$	$30min$	$0min$	$3min$
Defrost method (none/El/Gas/Brine)	d01	*none*	*brine*	*el*	*el*
Interval between defrost starts	d03	$0h$	$48h$	$8h$	$1h$
Maximum defrost duration	d04	$0min$	$180min$	$45min$	$8min$
Drip off time	d06	$0min$	$60min$	$0min$	$1min$
Delay for fan start after defrost	d07	$0min$	$60min$	$0min$	$2min$
Fan cutin during defrost	d09	*no*	*yes*	*yes*	*yes*
Defrost sensor (0=time, 1=S5, 2=S4)	d10	0	2	0	0
Fan stop at cutout compressor	F01	*no*	*yes*	*no*	*yes*
Delay of fan stop	F02	$0min$	$30min$	$0min$	$4min$

Table 6.1: A few selected controller parameters from EKC204A manual.

or two. The EKC is measuring the temperatures by reading the sensors and controls fan and compressor by switching their relays.

Figure 6.1b shows one way of connecting devices to the EKC unit: compressor relay is on 4-5 contacts, defrost heater on 6-7, fan motor on 8-9, alarm device on 10-11, sensors on 13-18, door sensors ("digital input") on 19-21. Depending on a particular application another compressor can be attached instead of fan, light installation instead of alarm and so on.

The EKC can be programmed to operate the devices with respect to the device configuration and individual refrigeration demands. The EKC logic parameterization is done via setting a number of register variables by using three buttons on a unit or via network. The register database consists of more than 70 variables, the most important ones are displayed in Table 6.1.

Figure 6.2 demonstrates the main EKC operation principle. The goal of temperature regulation is to keep the temperature at a designed *set point* (see SP in Table 6.1) with a small deviation defined by *differential* (r01), i.e. nor-

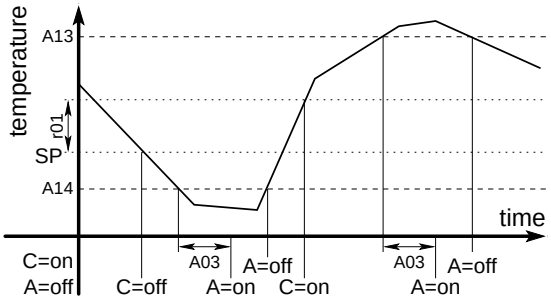


Figure 6.2: Controller actions during temperature regulation where $c01 = c02 = 0$, A is alarm relay and C is compressor relay.

mally the temperature should be between SP and SP+r01. In order to achieve this, the compressor is turned OFF whenever temperature drops below SP (allowing the room to warm up) and is turned ON whenever temperature exceeds SP+r01 (the running refrigeration fluid evaporates within evaporator and cools down the room). The compressor switching can be stressful for the motor and power supply, hence designers provided c01 and c02 parameters to postpone the switching by enforcing minimum ON-time and minimum OFF-time. The device is equipped with an alarm function which can be triggered whenever the door is left open or temperature is too extreme for too long time. Figure 6.2 also shows alarm relay switching ON whenever the temperature drops below *low temperature alarm limit* A14 or warms up above *high temperature alarm limit* A13. Note that the alarm is not raised until the *delay of temperature alarm* A03 has elapsed. The EKC unit is also responsible for controlling the defrost cycles in order to get rid of accumulated ice on the evaporator. The defrost can be triggered based on temperature readings or based on timing (d10) with specific intervals specified by d03. The defrost period can be limited by *maximum defrost duration* d04. The defrost cycle may also interact with other features: the compressor should be turned OFF whenever defrost is in progress, *delay of temperature alarm after defrost* A12 can be different from A03, the compressor start can be delayed after defrost to allow the water to drip off (d06) and the fan start can be delayed after defrost is over (d07). The EKC also controls the fan motor and can use it to distribute the temperature quicker whenever the compressor is ON and turn it OFF whenever the door sensor is open or compressor is OFF (F01). The fan switching OFF can be also delayed by F02. In order to ensure reliable and timely defrost cycles, engineers designed the software in such a way that defrost timers can never be reset even after factory default reset is issued or the power is disrupted.

Note that some parameter settings may result in inconsistent requirements. For example defrost interval (d03) can be set to 0 hours which imply continuous defrost (re-)start. A non-obvious inconsistencies may arise in more complicated configurations, consider the following setting where the fan should be turned ON and OFF at the same time: d09=yes and F01=yes, then the compressor should be turned OFF when the defrost starts (general requirement) and the fan should be turned OFF (F01 requirement, since the compressor is OFF) while at the same time as the fan relay should be cut-in, i.e. turned ON (d09 requirement since the defrost has started). It is not clear from the manual how

such situations should be resolved and it can get even more intricate when the timing requirements are added on top.

6.2 New Generation of Controllers

The new generation EKC controllers are equipped Pt sensors which measure the temperature with increased 0.1° accuracy. The Pt sensors are also more reliable and do not degrade over time.

The displayed temperature calculation procedure has been changed and now is processed gradually in small steps following the PID (proportional-integral-derivative) controller algorithm when the temperature change is less than 1°. Interestingly the temperature display is updated almost immediately to exact value when the temperature change is greater than 1°. Due to numerical methods used, the temperature display may exhibit instability by fluctuating between neighboring temperature values, e.g. display may switch back and forth between $16.7°C$ and $16.8°C$. Neither internal precision nor frequency, nor exact PID constants of internal temperature calculations are specified.

The new controllers also come with improved interface for test input (temperature) injection which allows testing the device behavior under more realistic conditions than before where we had to modify the temperature *setpoint* in order to trigger.

The output sampling period in the driver software has been reduced to $0.3s$ (although it still may take up to $1.35s$ depending on the load) and we have Windows port of TRON which may use the Windows drivers directly.

In the previous work [43] we experienced state set explosion of up to 3000 symbolic states which prevented us from testing features which required interaction with long defrost periods.

During the second iteration the methodological part has been improved, the new model is more abstract and sustains the state set to up to 250 symbolic states at a time, improved failure diagnostics, better input scheduling, trace replay possibility, coverage highlight in UPPAAL GUI enabled easier incremental model development and creation of test purposes.

6.3 The Modeling Methodology

Our goal is to test the timed features of EKC product, which means monitoring the displayed temperature, status of relays and determine if their behavior fits the description in the manual. First, we group the requirements and features in order to keep the model as simple as possible. One way of grouping is to create a separate process per each output aspect, i.e. one process responsible for calculating display temperature `tempMonitor` and one for each relay: `lowTempAlarm`, `highTempAlarm`, `compressor`, `fan` and `defrost`. Second, we need a flexible structure of environment in order to generate sensible inputs, therefore we have two processes: `tempGen` generates temperature changes while listening to `test` commands. Third, the test adapter layer inevitably introduces delays between signal transmission and reception, hence we add adapter processes for buffering and delaying the input and output signals: `relay` caries update on status of relays, `tempObserve` caries the displayed temperature value

and `tempInject` caries the value to be injected into temperature sensors. Figure 6.3 shows an overview how processes (entities in ellipses) communicate with

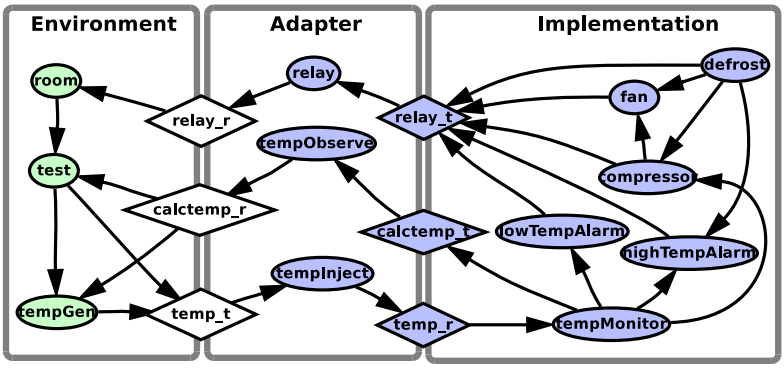


Figure 6.3: Communication flow diagram of EKC aspects.

each other: e.g. the `compressor` process controls relay (arrow from `compressor` to `relay_t`), its behavior depends on the currently displayed temperature calculated by `tempMonitor` (arrow from `tempMonitor` to `compressor`) and current defrost mode (arrow from `defrost` to `compressor`).

The processes in Figure 6.3 are partitioned into environment, adapter and implementation. The diamonds correspond to signal events on the adapter boundaries with environment and implementation. The events at the adapter-environment boundary are observable, while events on adapter-implementation boundary are not and are treated as all other IUT-internal ones.

In addition, we have to maintain that the online test assumptions are true: implementation should be input enabled and `tempMonitor` should be prepared to accept temperature injection at any time, environment should be input enabled hence `room` has been added to consume any relay change and the system should be free of time-locks and deadlocks in general.

We propose to use two testing modes:

- Online testing within environment as general as possible. It has an advantage of cheaply generating random but unexpected tests and disadvantages: environment is highly non-deterministic (makes it very hard to ensure the online test assumptions), such tests are highly unstructured, can be very long before hitting a fault, hard to reproduce and diagnose the location of a fault. Such environment is useful when developer has high confidence that implementation conforms to the model and does not expect elaborate fault diagnostics.

- Online testing within a guiding environment with a purpose of exercising specific parts of the IUT model. Such environment models are harder to create, but they are easier to analyze within model-checker, they give short traces leading to a fault, easier to reproduce and locate the fault. Such environment is useful when the model is not complete (e.g. some aspects are missing), but developer needs to gain the confidence that some specific aspect is implemented/modeled correctly. We also use this method in order to reproduce faulty cases discovered by general environment and to deduce the fault location and find a possible fix.

We used the following algorithm based on reverse engineering in order to refine our model when unspecified or unexpected (but still sound) behavior is discovered:

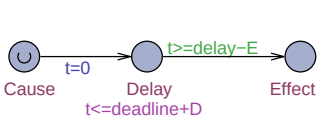
1. Formulate hypothesis model for one IUT component.
2. Create/update environment model with a purpose of testing the newly added IUT model features.
3. Validate the IUT and environment model composition against TRON test assumptions. If the IUT component model is already mature enough, then the purpose can be optimized to be shorter, allow broadest timing ranges and at the same time cover the target functionality for sure (independently from what IUT legal responses can be).
4. Run online test with the specified purpose environment.
5. If test fails, refine the IUT component model, replay the trace until the trace is accepted. Shortcoming: the model may require substantial editions so that the test purpose and the trace are no longer valid, then we have to go back to step 2.
6. If test passes, add a model for another component.

When the IUT model is complete, run the online test with most liberal but still realistic environment as long as possible.

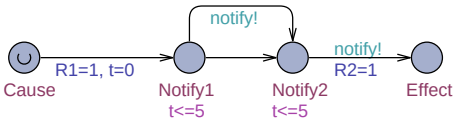
Further we show a few modeling patterns which make online testing of EKC feasible in practice.

6.3.1 Timing and Concurrency Tolerances

Suppose we need to model a delay between two events which is bounded from by `deadline` from above and by `delay` from below, which effectively means a non-deterministic delay of [`delay`, `deadline`] time units. Figure 6.4a shows a typical modelling pattern for delays between *Cause* and *Effect* using constraints in invariant and guard. Sometimes the values of `deadline` and `delay` are very



(a) Timing: t is a clock, *delay, deadline, E, D* – integers.



(b) Concurrent change: t is a clock, *R1, R2* – integers, *notify* – a channel.

Figure 6.4: Patterns for tolerance modelling.

close to each other or even equal, which means that such behavior is hardly realistic or implementable. In fact, depending on the nature of a delay, the cheap and non-critical implementations usually follow one of the two approaches: 1) schedule the second event a bit earlier than a deadline potentially violating the lower bound, or 2) delay the second event exactly to deadline and risk violating the upper bound by a small delay. We propose to enhance the delay boundaries with D and/or E integers in order to accommodate such timing tolerances. In

this case study one boundary extension at a time is enough and experimental test runs are used to determine which scheduling approach is used.

Suppose that two events happen at very close instances of time, and from tester's perspective they sometimes coincide and appear as one event. Usually such events are a result of chain-reaction of dependent events and there is a causality relation between them, although the causality span may be very short in time. In Figure 6.4b we propose to model the change of *R1* and *R2* variables where the notification about the change may happen just once (automaton takes τ transition instead of *notify!* and both *R1* and *R2* appear to change at the same time as `Notify2→Effect` transition) or twice (first *R1* and then *R2* change at `Notify1→Notify2` and `Notify2→Effect` instances). The maximum distance between events is constrained by 5 time units by invariants. Such behavior is observed in alarm handling where the main alarm relay value change depends on temperature relay changes, see Figures 6.11 and 6.10.

6.3.2 Observable I/O in Adapter

The main motivation for adapter modeling is to reflect the fact that it takes time to transfer observable input and output signals. Such timing is often abstracted away in model-checking, however it is crucial for determining the correctness in testing as precisely as possible. For example, if tester observes an output too late according to specification then any of the following can be true: 1) the device failed to comply with deadlines, 2) the output signal was delayed too long and/or 3) the output was a response to a delayed input signal to begin with. Hence it is important to have a model of input and output signals.

The main functionality of an adapter is queueing of input and output signals. In abstract terms the output is transmitted from IUT, saved in the adapter process and then received at the environment or tester's side. The same queueing principle applies to inputs. The signals usually travel through the same channels which allows to assume that signals are serialized in first-in-first-out (FIFO) order. Depending on the adapter architecture, the signal delivery involves scheduling and communication latencies hence the timing and concurrency tolerance patterns are used to an extreme degree.

Figure 6.13a shows *TempSignal* template used for modeling temperature input injection (another instance of such template is used to transfer the displayed temperature output). The *TempSignal* waits for signal to be `transmited` from shared variable `vfrom` and then it passes on this signal to be `received` at shared variable `vto` within `delay` time units which corresponds to a worst case communication latency. In this case study, the temperature is always injected one signal at a time, hence such single-signal `buffer` is enough to guarantee the input enableness. However this is not the case for relay output signals which have tight dependencies and tend to come at similar times, therefore we have a bit more complicated queueing with multiple instances of *RelaySignal* template from Figure 6.13b. Multiple instances correspond to multiple places in the signal queue. Such design however comes with a potential state space explosion due to many concurrent buffer processes. We employ partial order reduction by using an assumption that all signals are serialized (travel in FIFO order) to get rid of redundant interleavings: each instance of *RelaySignal* has its own `id` and shared variables `startturn` and `finishturn` determine which instance should be used in order to ensure FIFO order.

6.3.3 Temperature Estimation

By experimenting with the new test temperature injection mechanism, we found out that the temperature setting gets displayed almost immediately if a new temperature differs from the old one by more than one degree. If the change is less than one degree, then controller employs PID-like equation to remove sensor noise by executing it approximately once per second:

$$T_{n+1} = \frac{4 \cdot T_n + T_s}{5}$$

where T_s is a temperature sensor reading and T_n is n^{th} estimate of a temperature.

The controller operates on fixed-point numbers and thus depending on concrete temperature setting (positive or negative in Celsius scale), positive or negative change and the size of the change, the temperature is updated gradually and reaching the requested temperature within 7.0-14.5 seconds.

Figure 6.5 shows a dotted line of PID-like temperature estimation and a solid step-line of displayed temperature values between setting and observing the `new` temperature value. The temperature update steps do not happen at regular in-

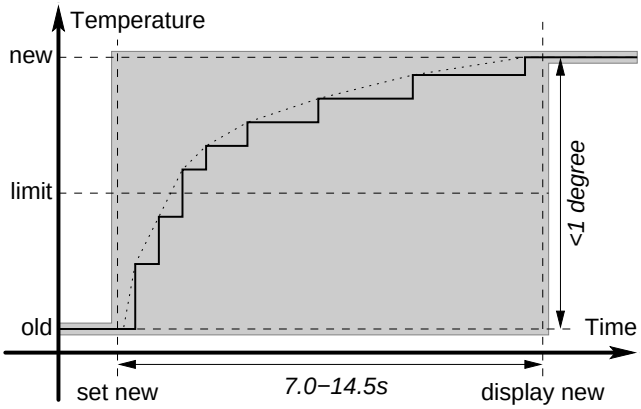


Figure 6.5: Displayed temperature calculation in the EKC.

tervals and we do not know if EKC uses the more precise PID-calculated value or the displayed value, moreover the temperature calculator often undershoots (does not reach the temperature value set by $0.1°C$). On one hand, previous study showed that it would be an overkill to model such non-deterministic temperature changes at such a detailed level. On the other hand we still need an estimate when some limit (e.g. high temperature limit) has been stepped over.

Similarly to an idea of piece-wise monotonic function modeling in timed automata from [28] and interval arithmetic [32] and propose to use a temperature over-approximation with two integer variables to represent the temperature estimate internally inside EKC: calculated temperature lower bound `CalcTL` and calculated temperature upper bound `CalcTU`. For example in Figure 6.5 we start with `CalcTL`=`CalcTU`=`old`, then upon `new` temperature injection we set `CalcTU` to `new` immediately and leave `CalcTL` unchanged until after $15s$ has passed. After $15s$ we set `CalcTL` to the `new` value. This way our temperature estimate

is always within the interval [`CalcTL`; `CalcTU`] showed in gray area. Figure 6.9 shows the model of such temperature calculation with two internal `tempChange` events: the first `tempChange` happens at a non-deterministic time within the first 150 time units, where any component has a chance to check if their limit has been stepped-over, and the second `tempChange` where the temperature settles down to one value.

With such model we do not know precise temperature between the "set `new`" and "display `new`" events, and we cannot check it at that period (at least not with current TRON implementation), therefore we modify the adapter to report the temperature changes only when the temperature actually reaches the value we injected. We also modify the temperature injection in order to get rid of spurious undershoots: we assume that PID-like calculations never overshoot (which seems to the case) and safely add 0.049° to the injected temperature change which attempts to overshoot the `new` temperature value, however 0.049° is too small and will be rounded down to the nearest 0.1° step which effectively hides our attempt to overshoot and does not allow PID to undershoot.

6.3.4 Test Purpose Construction

In Figure 6.3 the environment model consists of three parts:

- The `room` model consumes any output IUT might produce at any moment. The `room` component makes sure that the environment is able to observe any behavior and ensures that testing is not stopped due to environment model. Figure 6.14a shows a model for `room` process with coverage monitoring capabilities.

- The `tempGen` generates temperature changes according to testing commands either by incrementing or decrementing the temperature in timely fashion. The `tempGen` also consumes the displayed temperature updates in order to prevent generating temperature injections too often. Figure 6.14b shows the model for `tempGen`.

- The `test` drives the testing process by reading the environment variables and sending commands to `tempGen`. `test` may circumvent `tempGen` and feed the temperature value directly if a specific temperature value is needed. Figures 6.15a and 6.15b are examples for guided tests.

Ensure that model element is covered.

Find exact timing ranges for the most liberal test purpose which still achieves the coverage.

6.4 The Model

The model consists of a set of constants representing the parameter database, several processes representing different controller aspects, environment model and adapter processes modeling the signal transfer to and from the IUT. Figure 6.6 shows a signal diagram as an overview of entire system model. The blue items belong to IUTand green items to environment. We grouped the requirements into aspects denoted by underlined entities and modeled each aspect by

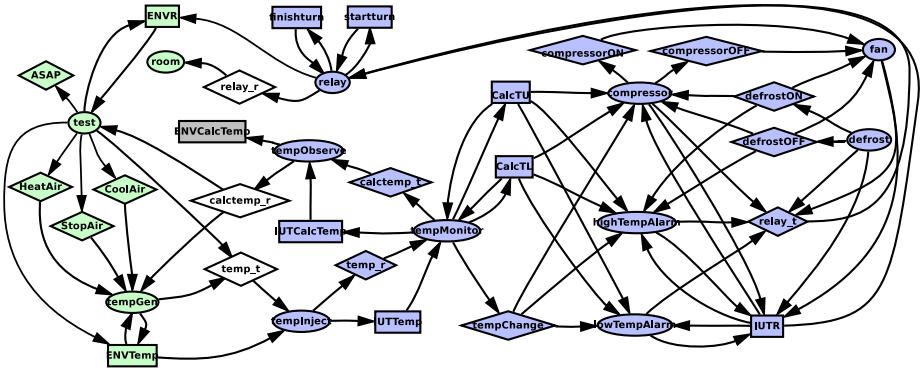


Figure 6.6: Signal flow diagram generated from the EKC UPPAAL model.

a separate process in the usual parallel composition, e.g. compressor process represents all requirements regarding the compressor relay control.

Listing 6.1 shows all global and shared declarations: list of integer constants modeling the fixed parameter values, relay state snapshot structure, relays state copies, channels for internal feature interaction inside EKC program, timing uncertainty constants, adapter channels and shared variables, constants and channels for the environment processes and system instantiation declaration.

```
1  // conventions:
2  typedef int[-5000,5000] TempT; // temperature type in 0.01 Celsius degrees
3  typedef int [0, 48*60*60*10] TimeT; // time in 0.1 seconds
4
5  // EKC register/parameter "database" (only relevant parameters)
6  const TempT Setpoint=200; // --- (#0), +2.0C
7  const TempT Diff=200;   // r01 (#1), differential, +2.0K
8  const TimeT TempAlarmDelay=8*60*10; // A03 (#24), delay before alarm
9  const TimeT PulldownDelay = 16*60*10; // A12, before alarm during defrost and
       startup
10 const TempT HighTempLimit= 700; // A13 (#22), 7.0C, high temp. alarm limit
11 const TempT LowTempLimit=-200; // A14 (#23), -2.0C, low temp. alarm limit
12 const TimeT MinOnTime=5*60*10; // c01 (#7), compr. min. time in "ON" state
13 const TimeT MinOffTime=3*60*10; // c02 (#8), compr. min. time in "OFF" state
14 const TimeT DefrostInterval=1*61*60*10; // d03 (#13), 1hour+1min(!)
15 const TimeT DefrostDuration=20*60*10; // d04 (#14), max. defrost duration
16 const TimeT DripOffTime=1*60*10; // d06, 1 minutes, wait for water to drip
17 const TimeT FanStartDelay=2*60*10; // d07, after defrost: start after compr on 2min
18 const bool FanDuringDefrost = 1; // d09, use fan during defrost
19 const bool FanStopComprOff = 1; // F01, stop when compressor turns off
20 const TimeT FanStopDelay=4*60*10; // F02, stop delay after compressor is off, 4min
21
22 // structure for storing state of all relays (snapshot)
23 typedef struct {
24     bool Compr;// compressor relay
25     bool Defr; // defrost cycle relay
26     bool Fan;  // fan relay
27     bool Alarm;// general (any) alarm relay
28     bool HAlarm;// high temperature alarm
29     bool LAlarm;// low temperature alarm
30 } Relays;
31
32 Relays IUTR = {1, 0, 1, 0, 0, 0}; // IUT copy of (up-to-date) snapshot
33 Relays ENVR = {1, 0, 1, 0, 0, 0}; // ENV copy of (last) snapshot
34 TempT ENVTemp=1600; // generated room temperature, initially +16.0C
35 TempT IUTTemp=1600; // temperature sensed by IUT, initially +16.0C
36 TempT CalcTL=1600, CalcTU=1600;// calculated lower and upper bounds of temp
37 TempT ENVCalcTemp=1600, IUTCalcTemp=1600;// calculated temp display (ENV and
       IUT copies)
38
39 // internal EKC notifications about temp, defrost and compressor status change:
```

```
broadcast chan tempChange;
broadcast chan defrostON, defrostOFF, compressorON, compressorOFF;

// internal EKC timing uncertainties:
const TimeT E=20; // allow change to be made 2s too early (in defrost)
const TimeT CRD = 20; // allow max 2s relay delay (in compressor)
const TimeT DRD = 20; // allow max 2s relay delay (in defrost)

// timing uncertainties in adapter:
const TimeT IOD = 14; // I/O delay: it takes at most 1.4s to get snapshot

// channel events in the adapter: input transmit and output receive are observable
chan temp_t, temp_r; // temp input (transmit and receive)
chan calctemp_t, calctemp_r; // calculcated temp output (transmit and receive)
chan relay_t, relay_r; // relay state output (transmit and receive)

// partial order reduction on signal buffering assuming that
// signals travel in serialized order:
const int relay_signals = 10;
typedef int [0,relay_signals-1] relay_signal_t;
relay_signal_t startturn=0, finishturn=0;

// tempGen properties:
const bool slow=1, medium=0, fast=0; // the speed of temp changes in tempGen
urgent chan HeatAir, CoolAir, StopAir; // "commands"
urgent broadcast chan ASAP; // "label" for urgent transitions
// interesting temperature limits:
const TempT limits[6]={-5000, LowTempLimit, Setpoint, Setpoint+Diff,
      HighTempLimit, 5000};
const TempT middle[5]={(limits[0]+limits[1])/2, (limits[1]+limits[2])/2,
         (limits[2]+limits[3])/2, (limits[3]+limits[4])/2,
         (limits[4]+limits[5])/2};
const TempT d = 0030; // threshold: 0.3 degrees
const TempT bounds[10]={limits[0]+d, limits[1]-d, limits[1]+d,
         limits[2]-d, limits[2]+d, limits[3]-d, limits[3]+d,
         limits[4]-d, limits[4]+d, limits[5]-d};
/** System declarations: */
relay(const relay_signal_t id) = RelaySignalG(relay_t, relay_r, IOD, IUTR, ENVR, id);
tempInject = TempInjectG(temp_t, temp_r, IOD, ENVTemp, IUTTemp);
tempObserve = TempObserveG(calctemp_t, calctemp_r, IOD, IUTCalcTemp,
      ENVCalcTemp);
tempGen = TempGenTestG(30*600);

system tempMonitorG, compressorG, defrostG, lowTempAlarmG, highTempAlarmG,
    fanG, roomG, tempGen, tempInject, tempObserve, relay;
```

Listing 6.1: Global and system declarations for EKC system model.

We start describing the modeled processes from defrost which is the simplest aspect in EKC. Figure 6.7 shows that we start in `WaitForOn` location and wait

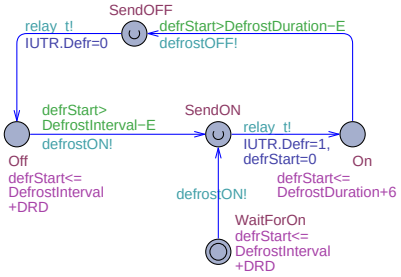
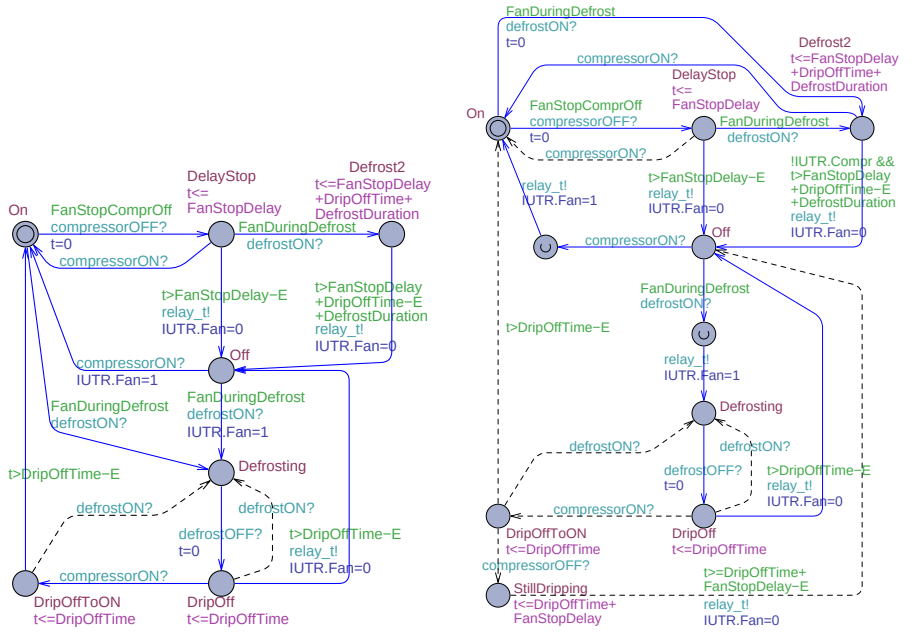


Figure 6.7: Defrost cycle: `defrStart` is a local clock.

until the first defrost starts. This is because we cannot reset the defrost timer by reseting the unit at the start of testing and we have no way of knowing when the defrost may start or finish because we do not know what happened before

the testing started. The first defrost start will become our point of reference hence we reset the clock `start`, notify other components about the defrost start by shouting on the broadcast channel `defrostON`, arrive at `SendON`, shout on `relay_t` to notify the outside world that we changed the relay `IUTR.Defr` and arrive to location `On`. Next, the process is allowed to stay in location `On` until `DefrostDuration` elapses and then we can turn the defrost relay OFF, but no earlier than `DefrostDuration-E` has passed. `E` is an uncertainty constant which allows the relay to be changes slightly earlier. It has been experimentally observed that relay may switch up to $E = 2s$ too early. After the defrost relay is switched OFF the process can stay in `Off` location until the next defrost cycle starts, i.e. until `DefrostInterval` elapses. The defrost may start up to $RD = 2s$ too late than the actual setting. We consider E and RD to be reasonably small compared to other timing constants (`DefrostDuration`$= 8min$) and hence not a fault. Note that uncertain defrost start and timing uncertainties introduce non-determinism into the model.

In a similar way we provide a model of a fan in Figure 6.8 where the process



(a) Passes TestFan but not TempGenTest. (b) Passes TestFan and TempGenTest.

Figure 6.8: Fan control: `t` is a local clock, dashed edges are not covered.

alternates between `On` and `Off` locations following the events from compressor processes with a few exceptions if defrost cycle is involved. Again, the relay change is notified by shouting on `relay_t` channel. The figure shows two versions: early model in Figure 6.8a model passes online tests with TestCompr and TestFan environment but fails a more random TempGenTest, and a more refined model in Figure 6.8b which passes TempGenTestG. We used Figure 6.8a for devising the test sequence TestFan and therefore could not foresee that this model may have additional edges, however TempGenTest revealed that Fan reacts to

Defrost and Compressor relay changes.

Figure 6.9 shows the temperature sensor monitor which is responsible for cal-

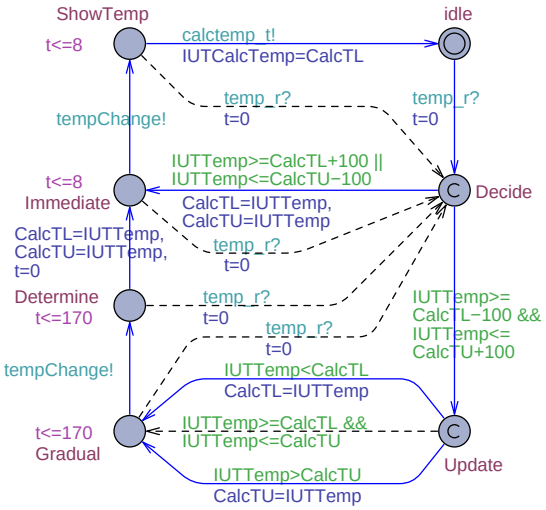
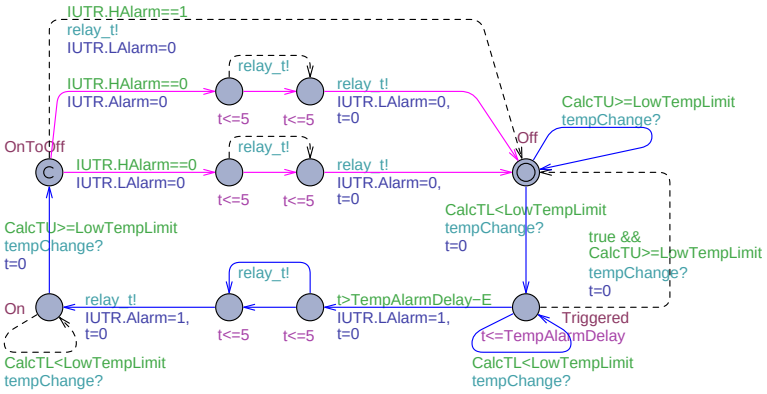


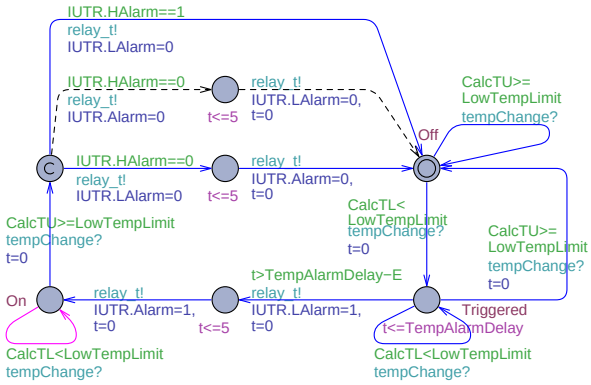
Figure 6.9: Temperature monitoring: `t` is a local clock.

culating the current displayed temperature from sensor reading `IUTTemp`. The problem here is that the temperature change is not displayed instantaneously but computed using PID numeric methods which approximate exponential nearing to the limit value set. In such case the temperature is approaching and reaching (and sometimes stabilizing just before reaching) the limit in about $7-14.5s$. The process of gradual temperature update is also non-deterministic in timing hence we cannot follow all the updates since many small updates at non-deterministic timings may raise the state set explosion. Instead we encode that currently displayed temperature by two integer variables `CalcTL` and `CalcTU` tracking the lower and upper bounds of possible temperature values. In location `Decide` the process checks whether the temperature is calculated gradually (goes to `Update`) or not (goes to `Immediate`). If the update is gradual the temperature limits are adjusted accordingly by taking one of the edges leading to `Gradual`. Now the temperature monitor notifies all IUT processes at any (and all) instance of time where $0 \leq t \leq 15s$ since we do not know neither if the potential temperature limit has been reached nor at which moment exactly the potential limit is reached. In addition it may take up to $0.8s$ to display the newly calculated temperature (the same as in an immediate update case).

Similarly to defrost and fan Figure 6.10 shows the low temperature alarm behavior which alternates between `Off` and `On` locations. The process is aware of the shared `CalcTL` and `CalcTU` variables which describe the possible temperature estimates. The alarm process is also non-deterministic due to the fact that it is not known when exactly the temperature limit is reached. For example if `CalcTL`$\leq$`LowTempLimit`$\leq$`CalcTU` then both edges `Off`$\rightarrow$`Off` and `Off`$\rightarrow$`Triggered` can be taken. In a similar way the process can return from `Triggered` to `Off` if the temperature remains around the `LowTempLimit`. Once the `TempAlarmDelay` elapses in `Triggered` location then the process should set the low temperature alarm `IUTR.LAlarm` and general alarm `IUTR.Alarm` relays to ON. Depending on



(a) Coverage by TestFan.



(b) Simplified, with coverage by TempGenTest.

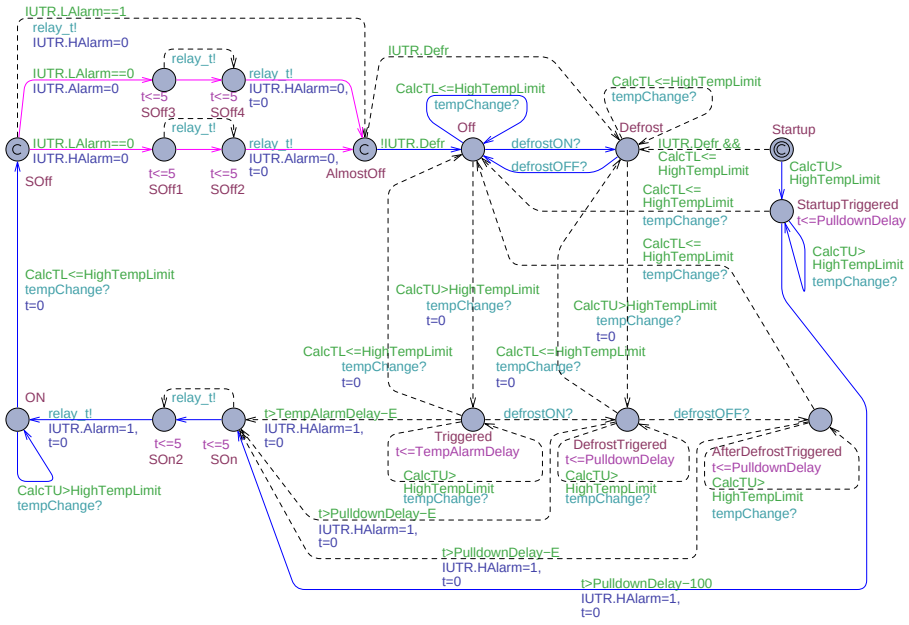
Figure 6.10: Low temperature alarm: `t` is a local clock.

snapshoting time we may observe both relays being set to ON at the same time, or sequentially: first low temperature alarm and then general alarm, hence we need to allow this in our model too. Similar observations are expected when the alarm goes OFF.

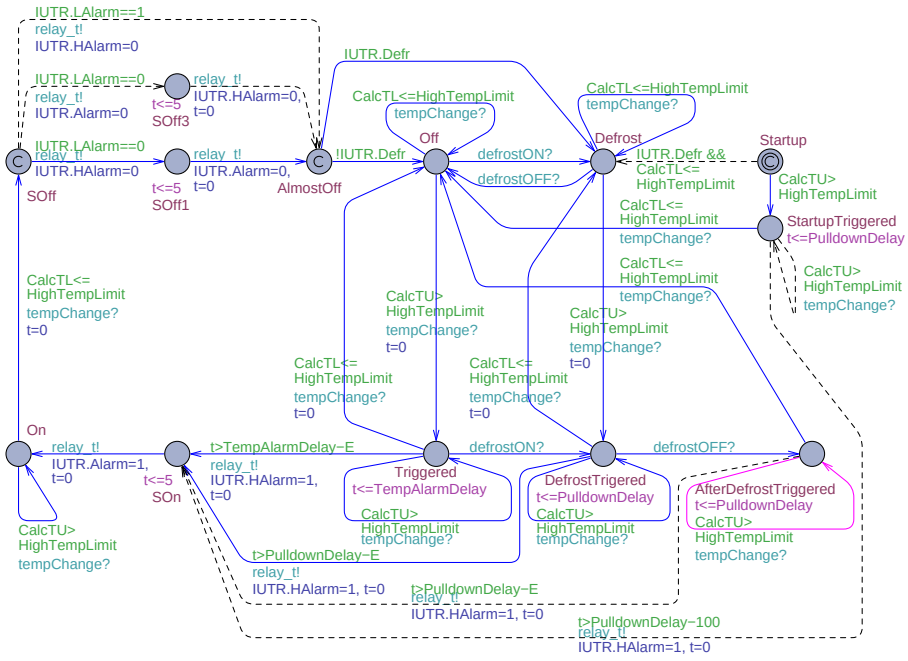
Figure 6.11 shows monitor model for high temperature alarm which is similar to low temperature alarm process except that it interacts with defrost cycle and different delays are applied if defrost has been observed.

Figure 6.12 shows the compressor relay model. The general idea is the same as with other relays: model the changes between `On` and `Off` locations while monitoring the calculated temperature through *CalcTL* and *CalcTU* variables. In this case we also have to take the minimum ON and minimum OFF time into account, hence locations `OnWait` and `OffWait` are added and local clocks *onTime* and *offTime* are reset accordingly. In addition, the compressor should be kept OFF whenever the defrost period comes and stay OFF for a `DripOff` period to allow the water to drip off after the defrost. In fact, the water drip off period is enforced so strictly that we have to track when the last defrost was over, therefore we reset the local clock d whenever `defrostOFF` is triggered (see e.g. self loops on locations `On` and `OffWait`).

Figures 6.13b and 6.13a show the adapter signal transfer models which are



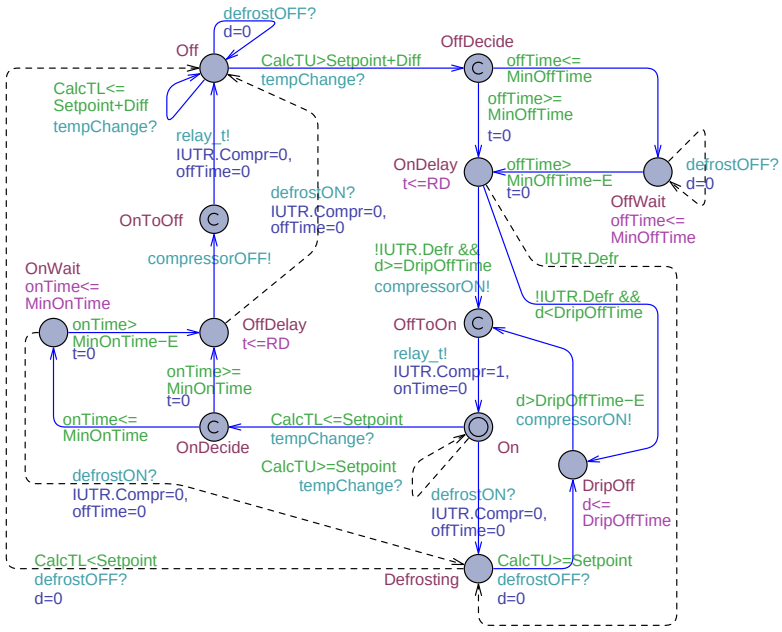
(a) Coverage by TestFan.



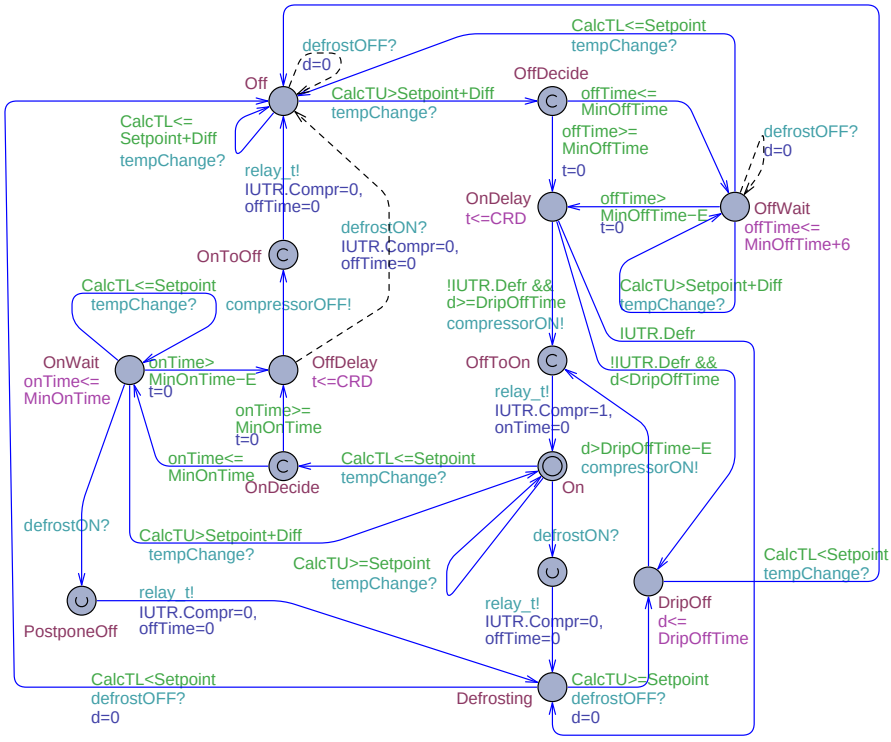
(b) Simplified, coverage by TempGenTest.

Figure 6.11: High temperature alarm: `t` is a local clock.

basically one-size buffers. We use `TempInject` for both to feed the new (input) temperature to the EKC sensors and observe the (output) calculated/displayed temperature. From Listing 6.1 (`tempInject` and `tempObserve` instantiation)

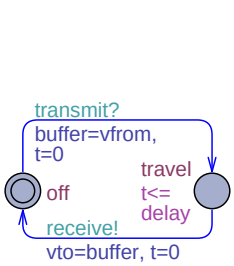


(a) Coverage by TestFan.

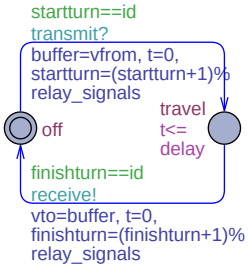


(b) Refined, coverage by TempGenTest.

Figure 6.12: Compressor: `minOnTime`, `minOffTime`, `d` and `t` are local clocks.



(a) TempSignal: `t` is a local clock, `buffer` is a local integer, `vfrom` and `vto` are shared integers.



(b) RelaySignal: `t` is a local clock, `buffer` is a local `Relays`, `relay_signals` is a total number of instances and `startturn` and `finishturn` are shared between instances.

Figure 6.13: Models for adapter signals.

one can see that `transmit` and `receive` channels are assigned to `temp_t` (temperature transmit) and `temp_r` (temperature receive) for temperature injection and `calctemp_t` and `calctemp_r` for calculated temperature observation. The temperature value is carried from `ENVTemp` to local integer `buffer` and then to `IUTTemp`, and `IUTCalcTemp` to another local integer `buffer` and then to `ENVCalcTemp`. Only events on `temp_t` and `calctemp_r` channels are observable between test driver and adapter and `temp_r` and `calctemp_t` happen between adapter and EKC. The temperature does not change very often, hence one instance per I/O channel is enough In Figure 6.13b we employ similar idea to transfer the relay state snapshot, hence `Relays` structure is being transfered via buffer. Unfortunately the relays may change independently of each other and some changes may happen at a very similar timings, hence there could be several relay signals travelling on the way in the adapter.

6.5 Coverage Estimation

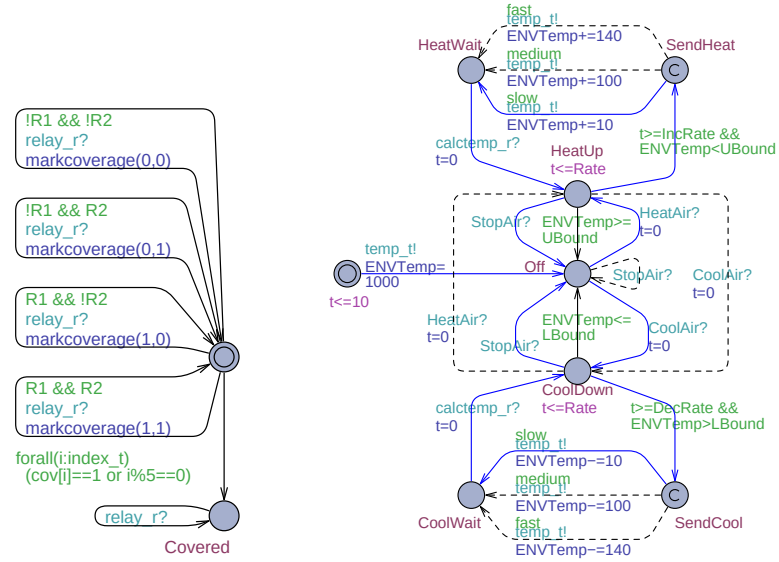
We estimate edge-coverage of by associating each edge with a boolean variable assignment to true, effectively including the coverage information into the state estimate. The coverage estimation is carried out offline in post-mortem analysis by replaying the recorded trace on a decorated model, thus this additional decoration does not hinder the performance of online test:

```
cat driver-cut.log fail.log | tron -Q log -l 11500 -P 300,300 \
    -F 500 -v 8 ekc2ecov.xml -I TraceAdapter - -m
```

where `driver-cut.log` is the driver log with ending cut off and `fail.log` is a fake faulty continuation of the trace which forces TRON to declare failure and dump the last state set containing the coverage information.

The state estimate consists of many symbolic states, thus this leads to a set of possible coverage estimates. We say that the edge is *definitely covered* if the coverage variable is set to true for all symbolic states from the final state estimate. Analogously, we say that the edge is *possibly covered* if the coverage variable is set to true only for some symbolic states from the final state estimate.

For example in Figure 6.10a TRON could not distinguish which path was



(a) Monitor for relay transition coverage.

(b) Temperature generator, where `t` is a local clock.

Figure 6.14: Environment parts for relay monitoring and temperature generation.

taken from location `OnToOff` to location `Off` and marked both paths as possibly covered. The reason is that the adapter always recorded that both relays `LAlarm` and `Alarm` were set simultaneously and the intermediate `relay_t` synchronisation was never used. Later we serialised the relay changes in the adapter by reporting one relay change at a time, this information allowed us to simplify the model into Figure 6.10b.

Similarly Figure 6.11a was simplified into Figure 6.11b.

6.6 Adaptation and Testing

The first case study published in [43] used an adapter involving a complicated chain of communication processes: TRON running on Linux machine, snapshot process running on Windows machine, gateway controller translating MODBUS messages from a particular EKC controller to and from TCP/IP streams.

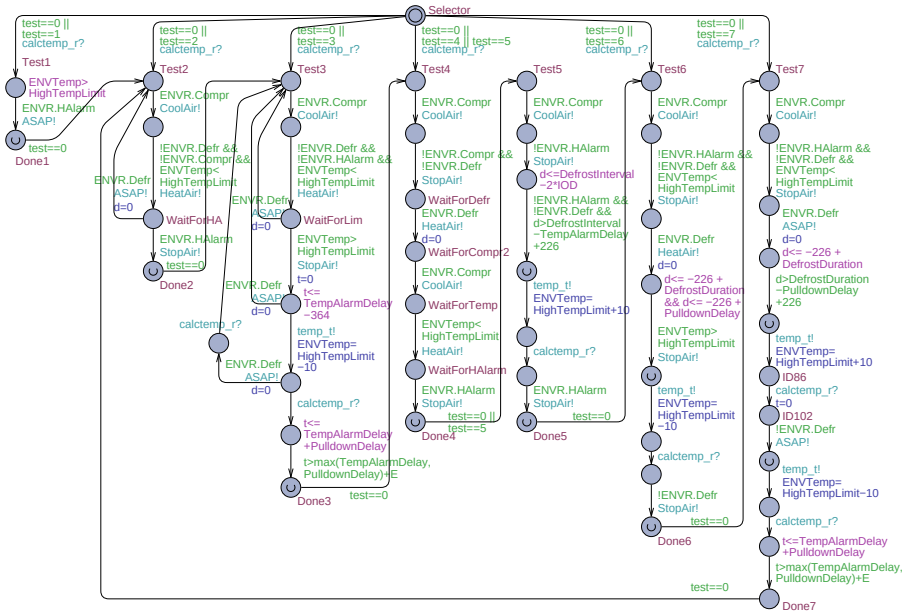
The new framework has been simplified by porting TRON to Windows, implementing TRON adapter which performs snapshoting while using native MODBUS drivers supplied by Danfoss.

Listing 6.2 demonstrates C++ loop from adapter code which takes controller relay snapshots and feeds input.

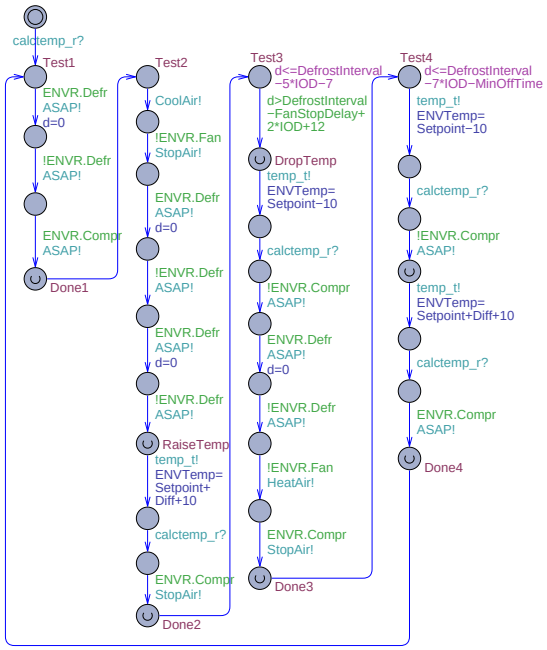
The test process is then launched using the following command line:

```
tron.exe -l 11500 -P 300,300 -F 500 -v 10 ekc2.xml -I Release/EKC.dll \
         -o tron.log -D driver.log -- IP:192.168.81.193 1
```

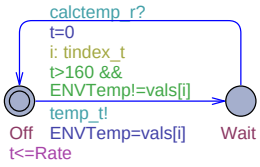
where the parameters do the following: set the communication latency to 11.5ms, the delay choice is limited by 30s, state estimate is precomputed with



(a) Guided tests for high temperature alarm: `t` and `d` are local clocks.



(b) TestFan: tests for fan, where `d` is a local clock.



(c) TempGenTest: generic temperature generator based on temperature bounds.

Figure 6.15: Two purpose-guided and one generic tests.

50s future horizon, model is loaded from `ekc2.xml` file, adapter loaded from `EKC.DLL` library, the TRON engine output recorded to `tron.log` file, the test

```
1  while (!stop) { //while testing continues
2    GetSnapshot(newSnap); //takes about 339723us, up to 1359631us
3    handleSnapshotDifference(lastSnap, newSnap); //report output if differ
4    tmp = lastSnap; lastSnap = newSnap; newSnap = tmp; //swap
5    a = waitForInput(330); //delay for 1/3 sec while checking input queue
6    while (a != NULL) { //if input action received
7      if (a->chanId==inps[EnvTemp].chanId) { //temp inject channel
8        // convert 100C integer to 1C floating point number:
9    double tempValue=((double)a->paramValues[0])/100;
10   if (tempValue<0) //may under-shoot above if negative temp:
11         INJECT(SENSOR2, tempValue-0.049); //send the temperature
12   else //may under-shoot below if positive temp:
13         INJECT(SENSOR2, tempValue+0.049); //send the temperature
14    lastInjected = tempValue; //expect this temp. displayed soon
15   delete a; a = NULL; //cleanup the data about input
16     }
17     a = tryGetInput(); //check input queue for more, just in case
18   }
19 }
```

Listing 6.2: Adapter C++ code sample.

events are logged into `driver.log` and at the end adapter parameters tell MODBUS drivers to connect to device number 1 at converter with given IP address.

6.7 Results

The final models are available on TRON web page:
http://www.cs.aau.dk/~marius/tron/Danfoss.

Figure 6.16 shows $10^5 s$ (27.7 hours) long trace from online test with FanTest environment (Figure 6.15b). The state set size varies between 1 and 156 states. Offline replay of 27 hour long trace with `fan` test takes about 2.5 seconds.

6.7.1 Undocumented Behavior

Before the right model is built, we have discovered undocumented behavior which manifested as test failures. In particular, manual does not mention interaction between `fan`, `compressor` and `defrost`. Figure 6.17 demonstrates three feature interaction which appeared as fault and the intended behavior is not clear. Around the 2:42:39 time `defrost` is turned OFF and followed by turning the `fan` off at 2:43:38 which is almost $1min$ = `DripOffTime` apart, thus the behavior here conforms to `d06` and `d09` requirements. At the time of 3:43:38 the `defrost` relay is stopped and situation seems identical to the last `defrost` cycle, however the `fan` is not stopped after $1min$ = `FanStopDelay` but instead at 3:46:22 which is 2min 44s after the defrost end, i.e. 0:02:44 too late.

The test was repeated several times and in each run there were exactly the same pattern: `fan` being turned OFF as `F01` says and later during the same run TRON complained that the `fan` is being late after `defrost` by 0:01:04, 0:02:00, 0:03:18, 0:03:47, 0:03:53 and even 0:03:59! A closer inspection revealed that

each such special `defrost` had always had a preceding `compressor` cycle:

1. In Figure 6.17 the `compressor` is turned OFF at 3:21:24, therefore according to `F02` the `fan` should have been turned OFF within $4min$ = `FanStopDelay`.
2. The `fan` apparently has been preempted by the `defrost` kicking-in at 3:23:39 and kept the `fan` being ON during the `defrost`.
3. The time difference between `compressor` turning OFF and `defrost` turning ON is 0:02:15 in Figure 6.17 and together with 0:02:44 of being late it makes the sum of almost 5min, which is consistent with a sum of `d06` and `F02` requirements (1min+4min=5min).

The hypothesis of 5min is tried on all other failing runs and the sums always added up to between 4:59 and 5:00, i.e. this provides evidence that `fan` stop timer is somehow suspended during the defrost cycle and the timeout used was a sum of the two requirements.

From modeling perspective, in order to reflect the `fan` stopping timer behavior one would need to stop the clock during the `defrost` cycle and resume clock with additional timeout. This directly asks for using stop watches, which was not available at the time of writing, but luckily the defrost cycle is governed by a constant (`d04` requirement, `DefrostDuration` = $20min$). Thus a simple `Defrost2` location with extended invariant is added to the `fan` process in case there is a defrost cycle preempting the `fan` going off.

6.7.2 Coverage

We have performed the coverage analysis post-mortem by replaying the recorded trace against the decorated model with a fan test. The covered edges are colored in the figures of timed automata in this chapter: the definitely covered edges are in blue, possibly covered are in magenta and not covered are dashed.

The dedicated test sequences such as TestFan result in fairly good coverage of that particular aspect, however they hardly allow discovering the hidden behaviour which is not in the model (the test sequence is biased). More randomised test environments such as TempGenTest are less biased towards the known model and thus exercise more obscure behavior and coverage is more complete, however diagnostics of failed traces is much more complicated than the dedicated ones.

6.8 Discussion

The case study resulted in a number of new features and fixes for both Uppaal and Tron: asynchronous I/O test adapter interface, latency option for better input scheduling, explicit model partitioning into environment and IUT requirements, improved failed test diagnostics, fast test driver trace replay possibility via TraceAdapter in virtual time, signal flow diagram generation from a given Uppaal model, edge and location coverage highlighting in Uppaal GUI, Tron port for Windows OS.

The modeling process showed that it is hardly possible to express requirements for embedded software in a systematic and consistent way when using

even well structured human readable text, tables and pictures. It is even harder to document the interaction of various features such as fan, compressor and defrost. The formal modelling of the system solves the specification problem and model-checking gives confidence that the model behaves as desired, however the modelling process is still cumbersome, iterative and lengthy if it is not done from the very beginning of product development.

The parallel composition of timed automata proved to be easy and efficient way to specify requirements grouped by features and test the system while monitoring all features at the same time in comparison to a test scripting where all combinations of correct and incorrect observations would have to be reasoned and enumerated separately.

The relativized part of conformance relation proved to be extremely useful in randomized testing in order to discover the intricate model details from the beginning. We showed how to specify explicit test scenarios which help ensure the syntactic model coverage of online tests and conclude that specific scenarios are useful to gain confidence in specific features, while randomised ones provide better model coverage overall.

As in a previous work [43] the adapter is also based on continuous register snapshoting and generating an output event when the value difference is detected. Here we showed how C-like code in UPPAAL language can be used to model the snapshots effectively. Besides providing a link to IUT the test adapter can also help solving the modeling problems where the modeling language lacks expressiveness: the adapter programming was used to compensate the PID-like temperature calculation instability problems. The adapter also exhibited significant signal delays which was explicitly and efficiently modeled and cannot be avoided if we want to examine how much control we can have over test inputs.

The study demonstrates how to obtain simple edge coverage of the model. The result shows that not all edges are covered and we provide the reasons why particular edges are not covered in TestFan:

1. `TempMonitor`: ShowTemp→Decide, Determine→Decide, Gradual→Decide, Update→Gradual ($CalcTL \leq IUTTemp \leq CalcTU$) which can be explained by the fact that the environment is designed to inject the new temperature only after the displayed temperature is stabilized (after >15s) and inject only the new temperature values.

2. `Compressor`: remarkably the basic functionality of switching On and Off is covered, and it is easy to see that all locations are traversed, but edges corresponding to more intricate cases are not covered, because the test was not designed to stress the compressor.

3. `LowTempAlarm`: Triggered→Off and SOff→Off are not covered because the temperature was dropping and rising slowly and both alarms were never On at the same time respectively. It is interesting to note that TRON could not distinguish which path (Alarm=0 or LAlarm=0 is executed first) is taken when the alarms is turned Off and thus marked both paths as possibly covered.

4. `HighTempAlarm` is not exercised almost at all because the temperature is set to high only in the initialisation of the fan test.

Although particular tests (models of environment) could be improved to yield better model coverage, the framework provides means of showing what has been tested and thus provides feedback on what could be improved.

The study does not find behaviour which significantly deviates from the last model, but it shows that TRON is able to detect intricate situations showing non-conformance to the intermediate models.

In the future it would be interesting to try different configuration settings, e.g. relation between `DripOffTime` in Fan, `MinOnTime` and `MinOffTime` in Compressor, and DefrostInterval may trigger more unknown interactions and additional functionality.

We conclude that TRON together with UPPAAL provides a powerful framework for specifying system level real-time models and testing industrial embedded systems against them using conformance relation.

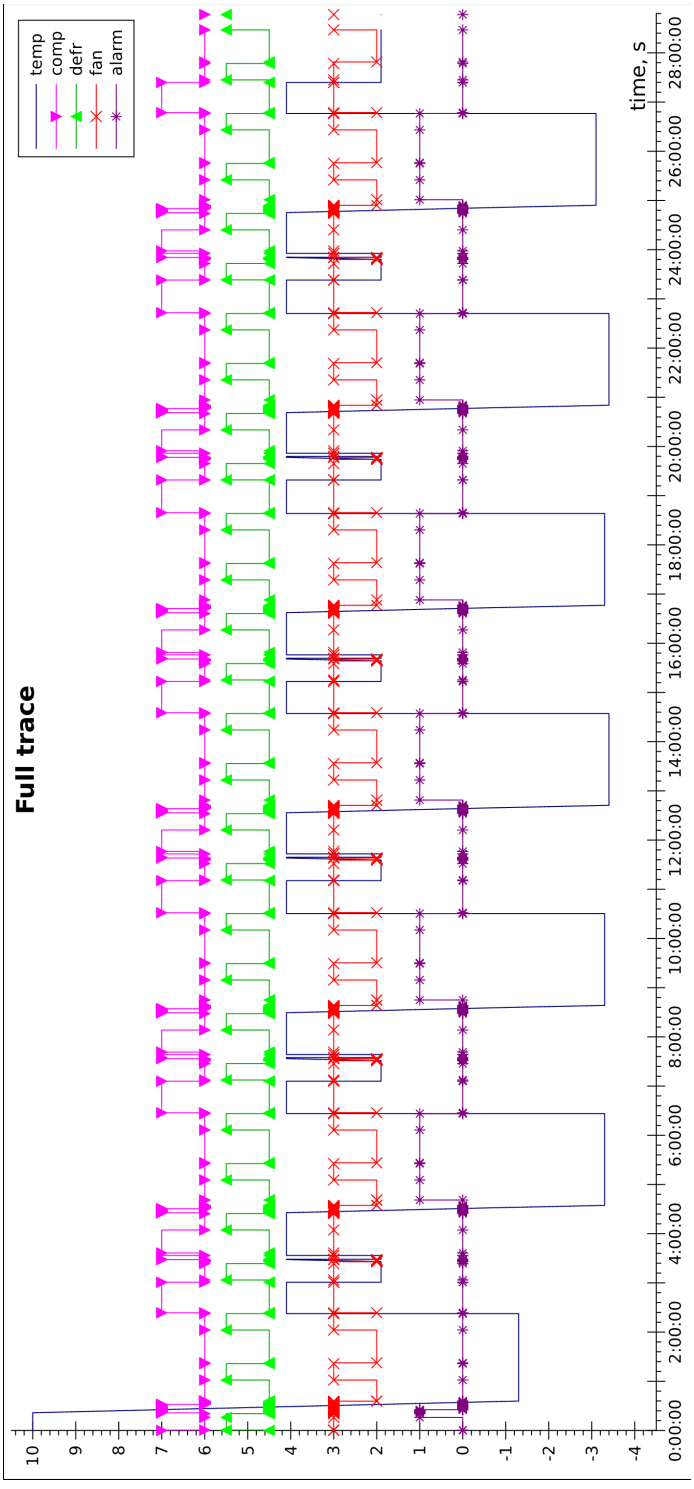


Figure 6.16: Visualisation of 27.7 hour test run with TestFan, stressing `fan` features: relay and temperature states are superimposed on the same graph. The x-axis shows the temperature values (blue curve without points), other signals up and down transitions correspond to relay switching ON and OFF.

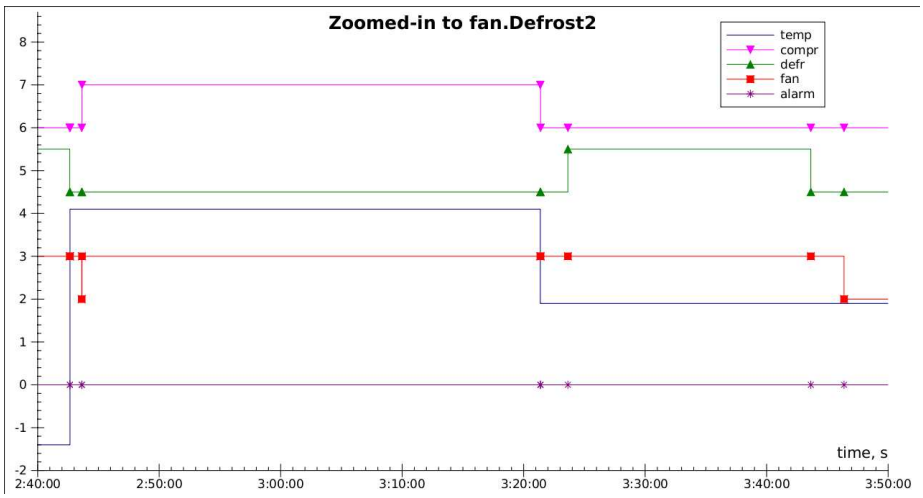


Figure 6.17: Undocumented `fan`, `compressor` and `defrost` interaction: relay changes superimposed with temperature curve, the signals go up and down denoting relay switches ON and OFF.

Chapter 7

Discussion

This chapter revisits the hypothesis and research questions outlined in the introduction, discusses the implications and possible directions for future work.

7.1 Theory

The thesis extends classical conformance testing framework of [60] for real-time systems by proposing timed input output conformance relation tioco . The conformance relation is developed further into relativized conformance relation rtioco_e which is a special case of tioco with environment. The thesis proposes an abstract (theoretical) algorithm which implements testing process to inspect the rtioco_e relation and proves that the algorithm is sound (IUT does not conform if test fails) and potentially complete (or exhaustive, i.e. is able to detect an existing fault) given enough time under digitizability assumptions. We conclude that the environment plays important role in real-time testing:

- The environment model makes testing assumptions explicit, i.e. the developer becomes aware of what kind of environment IUT is supposed to operate.
- The environment model provides additional structure on how the tests should be composed which is important for efficient test derivation online.

Thus we conclude that rtioco provides sufficient theory for real-time testing.

7.2 Implementation

The abstract testing algorithm operates on real-valued time and thus is not implementable by means of conventional hardware. To facilitate that, the thesis proposes a new symbolic online testing algorithm which operates on intervals as an over-approximation to capture the real-valued time stamps. The new algorithm retains most of the abstract algorithm structure, thus it can be used to determine the relativized timed conformance by using conventional means of computing. In addition, the thesis shows how the new algorithm is implemented in testing tool TRON by reusing UPPAAL components. The current state-of-the-art real-time model analysis is applied in online testing, thus we conclude that the most efficient analysis available is used to carry out the online test.

7.3 Adaptation

Timed automata formalism provides an abstract framework for reasoning about timed systems by assuming global time, instantaneous and atomic events, constituting Newtonian-like model of the Universe. The thesis argues that such formalism is still useful to reason about timed systems even with current understanding of nature, provided that we associate events with their physical time and space instants and reflect that fact in the specification model structure too. The thesis provides a methodology on how to develop the requirements and assumptions model together with the test adapter so that the principles of causality and measurement uncertainties are preserved by making the tester an independent observer referencing only its own physical clock. Thus all events are registered using the same clock and at the same location of a tester, and even then the precise instant of time is assumed to be unknown, except an explicit approximation of it.

The proposed interval time-stamping approach is very similar to digitization techniques [59], thus they can be used to prove the soundness of the technique for real valued time. Note that the duration of each interval corresponds to precision of a measurement, thus the approach constitutes an approximation in a sense that the fault may manifest but be undetected due to a limited precision.

We conclude that the proposed adaptation methodology makes the online real-time testing realistic for a large class of systems: larger than any other framework due to the fact that the tester and the IUTdo not share clocks and global time reference is absent.

7.4 Practice

The effectiveness of online test tool TRON has been measured empirically. The fault detection capability was examined by mutant study, in which we concluded that online test found almost all the seeded errors, except a few rare concurrency faults which probably did not have a chance to manifest in the first place. The source code coverage experiment confirmed that indeed almost all parts of the code have been exercised by the online test. The timely performance benchmarks concluded that the online test generation and monitoring impose insignificant overhead compared to scheduling of underlying operating system and thus online tests are applicable for many systems by deploying a regular computer. The technique also scales remarkably well with respect to a number of parallel components and in the future we expect even better performance if framework is distributed and multiple CPUs are deployed. Overall we conclude that online testing is an effective technique for finding real-time faults and has a wide range of applications.

The new testing tool TRON has been successfully applied in testing all essential real-time features of a single embedded device of an industrial refrigeration system. The case study demonstrates the methodology of using the UPPAAL tool suite:

1. Using UPPAAL to formalizing the requirements from a product manual.

2. Testing specific components one at a time by devising environment conditions stressing their functionality, while monitoring all other components

at the same time.

3. General online test of the whole system after gaining confidence in the complete requirement model.

Before the complete model is developed, the specific test cases are used as environment models. This allowed to track down specific conditions that lead to non-conformance and adjust the model accordingly. Thus we conclude that the novel treatment of environment model is useful in practice by providing modular structure for real-time requirements, optimize testing effort as well as focus testing on specific aspects. We also speculate that if the implementation is not robust enough (e.g. fails under universal environment), then explicit treatment of environment assumptions allow developer to formulate and discover the necessary conditions for correct behavior and such information can be used to create additional fixtures to ensure the discovered assumptions are fulfilled during deployment.

From software engineering perspective, TRON does not introduce any new extensions to UPPAAL language and many UPPAAL models may be used for the online testing purposes with small modifications to account for test adapter. We conclude that the methodology retains the idea of modeling abstract system level requirements and it is even possible to use partial system models (provided that features do not interact during test).

From software engineering perspective, the symbolic techniques implemented in UPPAAL and the pipeline architecture of operations are reusable for online testing purposes as well as model-checking tasks. Thus, the newly added features to UPPAAL (like stopwatches) gain support in UPPAAL TRON automatically.

Overall, we conclude that UPPAAL TRON, the result of this thesis, can be used to perform real-time tests online and determine the conformance relation with reasonable accuracy provided by the measurement instruments.

7.5 Future Work

This section suggests ways on how online testing framework can be utilized to provide more confidence in successful tests, improved test selection, generalized for hybrid systems and extended for distributed systems.

7.5.1 Coverage

So far TRON does not consider other confidence criteria apart from "tested long enough". However, it is possible to decorate UPPAAL models with coverage-tracking variable assignments as it is done in [31] ([29] generalizes the approach but uses special data structures to represent coverage in an efficient way). Given the measurement uncertainties and non-deterministic models it is not possible to determine definite coverage of a model. We envision that online testing will require a concept of a possible coverage in addition to definite coverage like it is documented in Section 6 and support for such notions could be implemented inside the tool.

The symbolic treatment of time opens possibilities for new kind of coverage: clock value coverage. The individual clock values from requirement constraints are not quite interesting by themselves. Moreover the model structure may be

(and most probably is) unrelated to the structure of a black-box IUT. However clocks may have more intrinsic interpretation and thus traversed values may be of interest. In particular, methods like [28] use real-valued clocks (stop-watches) to represent the state of a non-linear hybrid system, thus it is possible to estimate the state of a hybrid system by estimating timed automata state. Hence, particular clock valuations may characterize the structure of a hybrid state space, thus developer may be interested to know what states hybrid system may have visited during test execution. UPPAAL already contains the infrastructure for storing the clock valuations in various formats, thus TRON could take the advantage of such storage for recording coverage. The challenge is that the storage may demand a lot of memory for long test traces, thus the clocks would have to be selected carefully, storage organized separately from the explored state estimates and analysis performed offline or by a separate computation thread which would not disrupt the test execution.

7.5.2 Test Guiding

Current TRON implementation uses random choice to resolve test selection. The test selection could be improved by local constraint analysis like in [50], global static analysis of data flows in the model before the test begins or by information provided from recorded coverage.

7.5.3 Testing Hybrid Systems

TRON uses UPPAAL for model specification and analysis. It is easy to see that the online test approach can be generalized for hybrid systems by using a corresponding model-checker. We foresee a test framework setup shown in 7.1 where test generation and monitoring are split into two separate activities which synchronize via a hybrid adapter. In this case, the hybrid adapter would have to

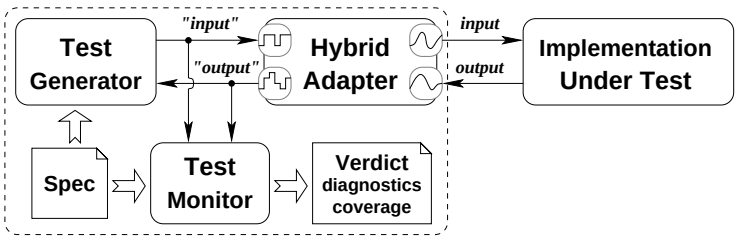


Figure 7.1: Framework for online testing of hybrid systems.

translate not just particular input/output actions and signal values, but also converting abstract actions into signal trajectories.

The Danfoss case study has stressed testing the real-time requirements but it has completely abstracted away the sensed temperature estimation aspect. Here we check this aspect by using PHAVER – model-checker for linear hybrid systems. The sensed temperature estimate is calculated by controller approximately each second by using equation $T_{n+1} = \frac{4 \cdot T_n + T_s}{5}$, where T_s is a temperature sensor reading and T_n is the n^{th} estimate of a temperature. The computation is not performed at strict time intervals and fixed-point arithmetics have

peculiar rounding effects, thus by having this information, we created a hybrid model with relaxed requirements which essentially say that the estimated temperature may fluctuate between narrow bounds. Figure 7.2 shows the hybrid automaton model of temperature estimation requirements. Similarly the model is complemented by the environment model shown in Figure 7.3 which describes how the room temperature may change.

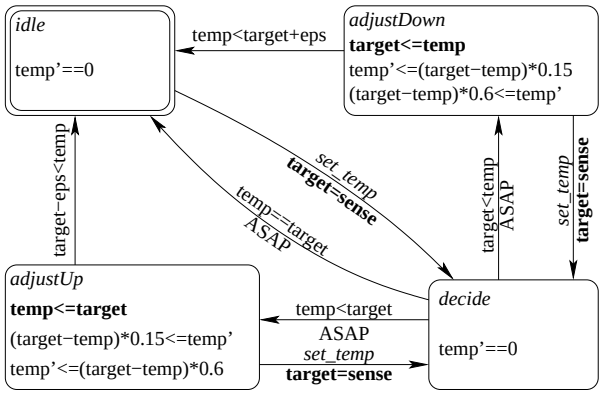


Figure 7.2: Model of a controller temperature sensing and calibration.

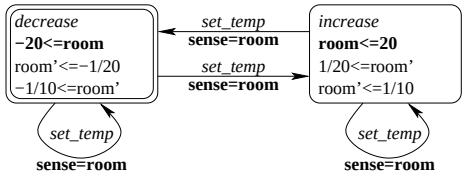


Figure 7.3: Model of a room temperature.

A small C++ program is used to generate a timed temperature input sequence of $0.9°C$ decrements followed by increments when temperature is below $-7°C$. The generated sensor values are fed into EKC, the displayed values are collected from EKC snapshots and fed into PHAVer tool. The resulting temperature estimate plot is shown in Figure 7.4. The zoomed-in part is commented as follows:

1. At time instance between 407s and 408s a new sensor temperature is set to $-7.7°C$.
2. The display temperature is estimated by a set of polygons up to 410s.
3. At time instance between 409s and 410s a new displayed temperature value of $-7.0°C$ is registered.
4. A new estimate for displayed temperature is calculated from 409s to 411.
5. At time instance between 410s and 411s a new displayed temperature value of $-7.2°C$ is registered.
6. A new estimate for displayed temperature is calculated from 410s to 413.

7. At time instance between 412s and 413s a new displayed temperature value of $-7.3°C$ is registered.

8. And so on, until the displayed temperature converges to $-7.7°C$ at instance between 417s and 418s.

9. At instance between 423s and 424s a new sensor temperature is set to $-6.8°C$ and the process is repeated until the displayed temperature converges at around 432s and 433s.

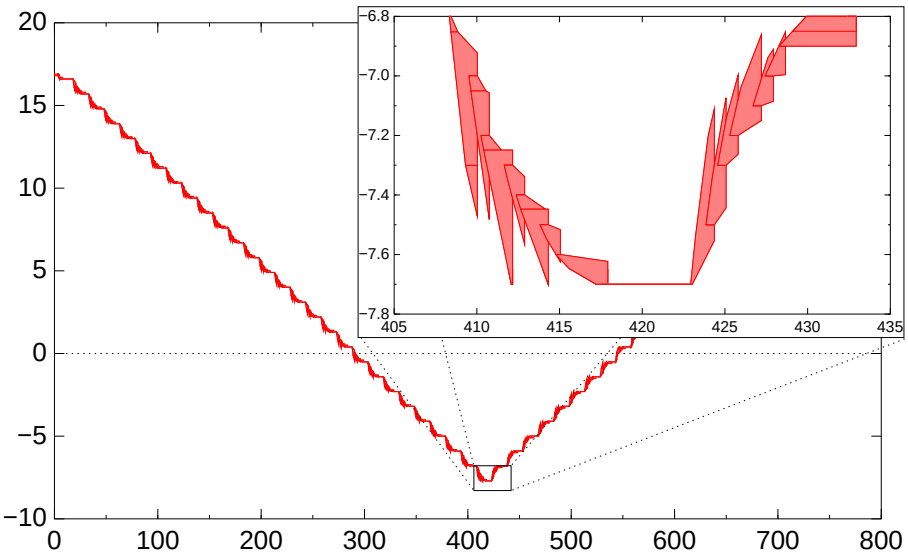


Figure 7.4: Symbolic state evolution in PHAVer from test trace monitoring: time in seconds on horizontal axis, temperature in $°C$ on vertical axis.

Here we have shown how to monitor the hybrid behavior aspects, however online test trajectory generation may demand much faster response and better sampling granularity than hybrid model-checker may provide, thus a more lightweight model simulator (like Matlab Simulink) may be used to emulate the environment model.

7.5.4 Testing Distributed Systems

This section shows how the framework could be extended to handle IUTwhich consists of a network of black-boxes.

An simple solution could be to create multiple TRON instances to monitor each black-box with corresponding requirement model and have dedicated TRON instances for input generation. In such setup, the effort of testing is distributed among many TRON instances and it could provide reasonable stress test, however the diagnostic is not so clear due to lack of orchestration and synchronization between TRON instances.

In a centralized approach with one big model of a distributed system running on one instance of TRON, would require the adapter framework to allow event

time-stamping from other sources than just the tester itself. In fact, such time-stamping has a potential to improve the measurement precision because the measurements could happen closer to the source of events. However the tester would have to consider every possible event interleaving because the event order can no longer be fixed (currently it is solved by serializing all events with tester's clock and considering the orders described by the adapter model).

We foresee that state estimate would have to be performed incrementally by keeping track of which events are already recorded and leave possibility to compute alternative interleaving if another event is recorded with a similar time-stamp. The state estimation would then result in maintenance of state-set trees like shown in Figure 7.5. In order to preserve the memory the state-sets can be

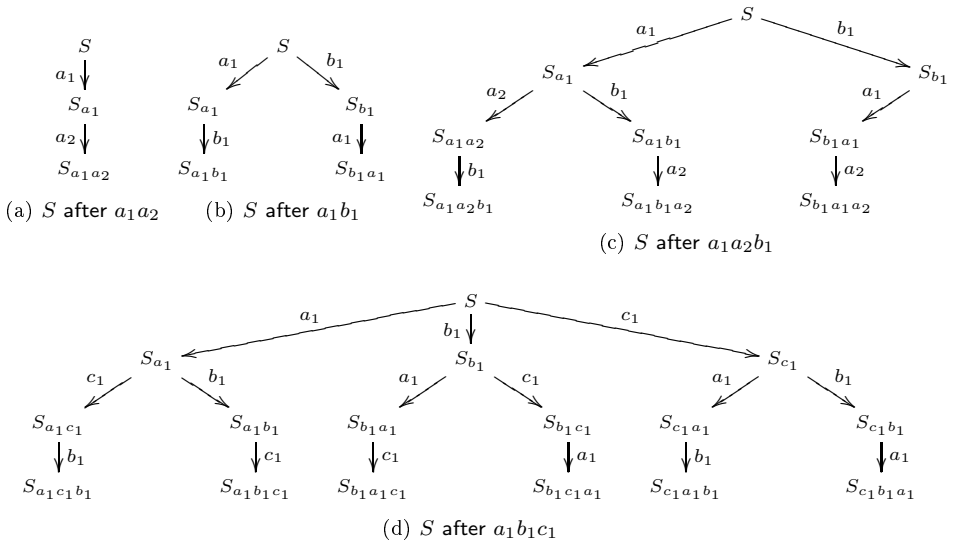
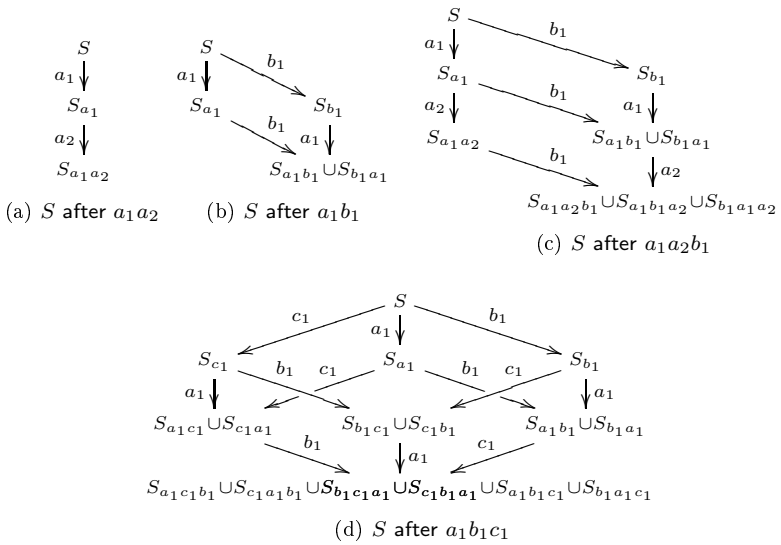


Figure 7.5: Examples of state-sets trees for events a_i, b_i and c_1 which are serialized in parallel channels a, b and c respectively.

merged incrementally. The merging may potentially lead into exponential reduction of symbolic states if the events happen to be independent (the resulting end states are equivalent and/or clock valuation zones can be merged into one zone).



(a) S after a_1a_2 (b) S after a_1b_1 (c) S after $a_1a_2b_1$

(d) S after $a_1b_1c_1$

Figure 7.6: Collapsed state-sets trees for events a_i, b_i and c_1 which are serialized in parallel channels a, b and c respectively.

Bibliography

[1] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *ICALP '90: Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pages 322–335, London, UK, 1990. Springer-Verlag.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *Formal Methods for the Design of Real-Time Systems*, pages 1–24. 2004.

[4] G. Behrmann, K. G. Larsen, and R. Pelánek. To store or not to store. In *Proc. Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 433–445. Springer, 2003.

[5] Gerd Behrmann. *Data Structures and Algorithms for the Analysis of Real Time Systems*. PhD thesis, Aalborg University, November 2003.

[6] Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer Aided Verification*, pages 341–353, 1999.

[7] A.F.E. Belinfante. Timed testing with TorX: The oosterschelde storm surge barrier. In M. Gijsen, editor, *Handout 8e Nederlandse Testdag*, Rotterdam, 2002. CMG.

[8] Johan Bengtsson. *Clocks, DBMs and States in Timed Systems*. PhD thesis, Uppsala University, 2001.

[9] Johan Bengtsson and Wang Yi. On clock difference constraints and termination in reachability analysis of timed automata. In J. S. Dong and J. Woodcock, editors, *Proc. of ICFEM'03*, number 2885 in Lecture Notes in Computer Science. Springer–Verlag, 2003.

[10] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.

[11] Henrik Bohnenkamp and Alex Belinfante. Timed testing with torx. In *FM 2005: Formal Methods*, Lecture Notes in Computer Science 3582, pages 173–188, 2005.

[12] Laura Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. PhD thesis, University of Twente, Enschede, The Netherlands, September 2007.

[13] Laura Brandán Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In *Formal Approaches to Software Testing*, pages 64–78, Linz, Austria, September 2004. Springer Berlin / Heidelberg.

[14] Laura Brandán Briones and Ed Brinksma. Testing multi input-output real-time systems. In *ICFEM 2005 Seventh International Conference on Formal Engineering Methods.*, page to appear, Manchester, UK, Nov 2005. Springer-Verlag GmbH.

[15] Laura Brandán Briones and Mathias Röhl. 8 test derivation from timed automata. In *Model-Based Testing of Reactive Systems*, pages 201–231. 2005.

[16] Rachel Cardell-Oliver. Conformance tests for Real-Time systems with timed automata specifications. *Formal Aspects of Computing*, 12(5):350–371, December 2000.

[17] J. M. Chambers. *Statistical Models in S*. Wadsworth & Brooks/Cole, Pacific Grove, California, 1992. Chapter 4: Linear Models.

[18] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: a symbolic test generation tool. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–173. 2002.

[19] Alexandre David. Uppaal dbm library. http://www.cs.auc.dk/∼adavid/UDBM/, December 2006.

[20] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. Cooperative testing of timed systems. *Electronic Notes in Theoretical Computer Science*, 220(1):79–92, December 2008.

[21] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. A game-theoretic approach to real-time system testing. In *Proceedings of the conference on Design, automation and test in Europe*, pages 486–491, Munich, Germany, 2008. ACM.

[22] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. Timed testing under partial observability. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 61–70. IEEE Computer Society, 2009.

[23] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989.

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

[25] 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.

[26] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager. Adding symmetry reduction to UPPAAL. In K. G. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, number 2791 in LNCS, pages 46–59. Springer–Verlag, 2004.

[27] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for Real-Time systems. *Information and Computation*, 111(2):193–244, June 1994.

[28] Thomas A. Henzinger and Pei-Hsin Ho. Algorithmic analysis of nonlinear hybrid systems. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 225–238. Springer-Verlag, 1995.

[29] Anders Hessel. *Model-Based Test Case Generation for Real-Time Systems*. PhD thesis, Department of Information Technology, Uppsala University, 2007.

[30] Anders Hessel, Kim Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, pages 77–117. 2008.

[31] Anders Hessel, Kim Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-Optimal test cases for Real-Time systems. In *Formal Modeling and Analysis of Timed Systems*, pages 234–245. 2004.

[32] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48(5):1038–1068, 2001.

[33] 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.

[34] Reactive Systems Inc. Reactis tester (product information). http://www.reactive-system.com/, December 2009.

[35] J. Ouaknine and J. Worrell. Revisiting digitization, robustness, and decidability for timed automata. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003) Ottawa, Canada*, pages 198–207. IEEE Computer Society, june 2003.

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

[37] 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.

[38] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *SPIN*, pages 109–126, 2004.

[39] Moez Krichen and Stavros Tripakis. An expressive and implementable formal framework for testing Real-Time systems. In *Testing of Communicating Systems*, pages 209–225. 2005.

[40] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, June 2009.

[41] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[42] Kim G. Larsen, Marius Mikučionis, and Brian 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.

[43] Kim G. Larsen, Marius Mikučionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306, New York, NY, USA, 2005. ACM.

[44] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[45] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In *Fundamentals of Computation Theory*, pages 62–88, 1995.

[46] Marius Mikucionis, Kim Guldstrand Larsen, and Brian Nielsen. T-uppaal: Online model-based testing of real-time systems. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 396–397, Linz, Austria, 2004. IEEE Computer Society.

[47] Marius Mikučionis, Brian Nielsen, and Kim G. Larsen. Real-time system testing on-the-fly. In Kaisa Sere and Marina Waldén, editors, *the 15th Nordic Workshop on Programming Theory*, number 34 in B, pages 36–38, Turku, Finland, October 29–31 2003. Åbo Akademi, Department of Computer Science, Finland. Abstracts.

[48] Marius Mikučionis and Eglė Sasnauskaitė. On-the-fly testing using UPPAAL. Master's thesis, Department of Computer Science, Aalborg University, http://www.cs.aau.dk/∼marius/master.pdf, June 2003.

[49] Ivan Moore. Jester - a junit test tester. In Giancarlo Succi Kent Beck, Michele Marchesi, editor, *2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, Villasimius, Sardinia, Italy, May 2001. http://jester.sourceforge.net.

[50] Brian Nielsen. *Specification and Test of Real-Time Systems*. PhD thesis, Department of Computer Science, Aalborg University, 2000.

[51] Brian Nielsen and Arne Skou. Automated test generation from timed automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 343–357. 2001.

[52] 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/.

[53] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005. ISBN 3-900051-07-0.

[54] Tomas Gerhard Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Standford University, 1993.

[55] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In *Integrated Formal Methods*, pages 338–357. 2000.

[56] U. Sammapun, Insup Lee, and O. Sokolsky. Rt-mac: runtime monitoring and checking of quantitative and probabilistic properties. In *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*, pages 147–153, Aug. 2005.

[57] Steve Sims and Daniel C. DuVarney. Experience report: the reactis validation tool. *SIGPLAN Not.*, 42(9):137–140, 2007.

[58] Jan Springintveld, Frits Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.

[59] T.A. Henzinger and Z. Manna and A. Pnueli. What good are digital clocks? In Werner Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria*, volume 623 of *LNCS*, pages 545–558. Springer, july 1992.

[60] Jan Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[61] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99 Concurrency Theory*, page 779. 1999.

[62] Jan Tretmans and Axel Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[63] Stavros Tripakis. Fault diagnosis for timed automata. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 205–224, London, UK, 2002. Springer-Verlag.

[64] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7th Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.

Appendix A

UPPAAL TRON Manual

A.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 A.1.3, proceed with formal and practical framework setup in sections A.1.4, A.1.5, A.2 and use sections A.3, A.4, A.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.

A.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.
- The 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 good location, edge and variable value coverage.
- 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.

A.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 smart-lamp example.
4. Graphviz [25] utilities for model signal-flow diagrams layouts in pictures.
5. R language and environment for statistical computing and graphics for displaying scheduling latency experiment results.
6. GhostViewer `gv` for displaying PostScript pictures generated from scheduling latency experiment.
7. GNU Compiler Collection (GCC) and `make` for dynamic library (DLL) adapters on Linux (button example).

8. Microsoft Visual Studio 2005 for dynamic library (DLL) adapters on Windows (MSVC button example).

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`

A.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.

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] 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.

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.

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.

Smart-lamp Demo

The goal of this example is to demonstrate how TRON can automatically test the temporal constraints of a simple yet realistic system. The idea is based on concepts of commodity "smart" lamp that changes the light level upon human touch. The interaction protocol is that the level should go up or down while a wire is grasped and stop at the current light level when the wire is released. The lamp also reacts on fast grasp-and-release "touch" gesture which turns the lamp off or turns back on to the light level it was on before. Smartlamp is a Java application that mimics such behavior. The example files are located in `java` directory of TRON distribution.

Figure A.1 shows the smartlamp test setup. The LightController is the main executable class. Internally the application consists of three parts: graphical user interface (GUI), LightController and for TRON adapter. The GUI shows the 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 .  java/LightController -M 0
```

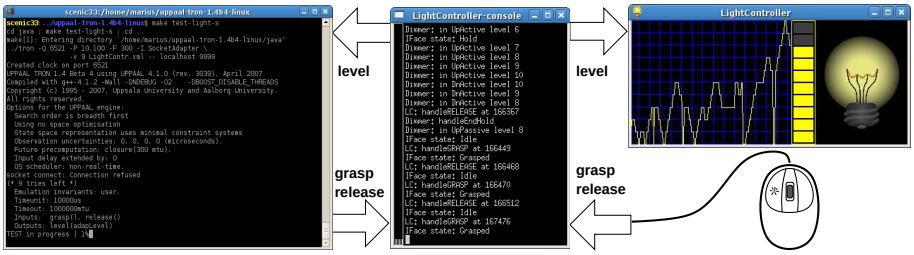


Figure A.1: Smartlamp setup: LightController (in the middle) connected to TRON (on the left), level view window and a mouse (on the right).

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

1. Start smart-lamp at one command prompt:
   ```
   java -cp .  java/LightController -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 .  java/LightController -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 A.6 if TRON reports test failures on mutant M0 in real time.

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

Smart-lamp Mutant Exercise

The purpose of this exercise is to demonstrate TRON's capability of catching faulty implementations called mutants. 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 .  java/LightController -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 LightController 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. You typically need to edit the `java/LightController.java` and `java/Dimmer.java` files. Remember to recompile the LightController once edited:

```
javac -cp .  java/*.java
```

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
```

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 .  java/OnOffLightController -C localhost 8989 -M 0
../tron -Q 8989 -P 10,200 -F 300 -I SocketAdapter -v 10 onOffLight.xml - localhost
9999
```

A.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 [60, 61]. 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 A.1 augments the formal definition of rtioco [42] 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. A.1 Relativized timed input/output conformance relation *for input enabled timed input/output labeled transition systems* $p, s \in \mathcal{S}$ *and* $e \in \mathcal{E}$*:*

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

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,*

$\mathsf{TTr}(e)$ *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,*

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

$\mathsf{Out}(\text{states})$ *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 $\delta \in \mathsf{Out}\big(\langle e, p\rangle \text{ after } \sigma\big)$ and $o \in \mathsf{Out}\big(\langle e, p\rangle \text{ after } \sigma\delta\big)$. The tester should compute the largest delay d such that $d \in \mathsf{Out}\big(\langle e, s\rangle \text{ after } \sigma\big)$ 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 $o \in \mathsf{Out}\big(\langle e, p\rangle \text{ after } \sigma d'\big)$ 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 $o \in \mathsf{Out}\big(\langle e, p\rangle \text{ after } \sigma\delta\big)$ 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 $o \in \mathsf{Out}\big(\langle e, p\rangle \text{ after } \sigma d'\big)$ 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 rtioco_{e_U} s coincides with timed trace inclusion and is equivalent to p tioco s.

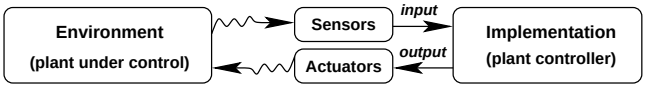
   (b) Silent environment e_S which does not allow any inputs but merely consumes outputs and lets the time pass: $\mathsf{TTr}\big(e\big) = (A_{out} \cup \mathbb{R}_{\geq 0})^*$. This is the same as $A_{inp} = \emptyset$ where tester is allowed only to observe the behavior of implementation. Such activity 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 [42] 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.

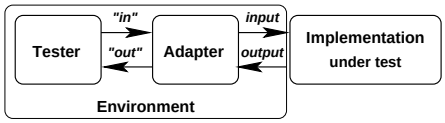
A.1.5 Online Test Setup

We consider closed systems, where implementation together with its environment can be isolated from the rest of the world. Figure A.2a shows typical

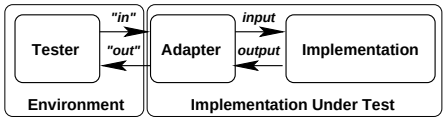
system setup during the system deployment: environment is a plant that needs to be steered and controlled, and implementation under test is a software/hardware controller taking inputs from the sensors embedded in the environment and producing output to actuators influencing the environment. Notice that 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.

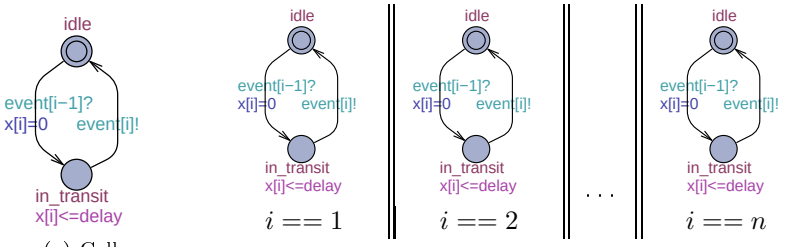
Figure A.2: Implementation during deployment and testing.

In Figure A.2b we replace the environment, sensors and actuators with a tester and a test adapter in order to test such controller. In a 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. Hence we arrive to TRON test setup shown as tester's perspective in Figure A.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 A.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 the 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 A.3a and n-size buffer is a parallel composition of n cells composed in a sequence as in Figure A.3b. Based



(a) Cell.

(b) n-size FIFO buffer.

Figure A.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 [36, 37] 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 a model time interval, whose bounds are the closest integers to the 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.

A.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.

We will use the fridge system from Figure A.4 as a running example to demonstrate how typical system model is composed for testing using TRON. The fridge

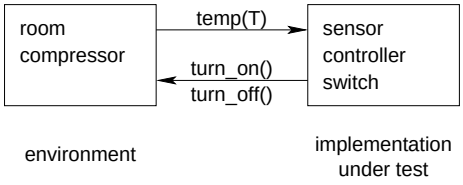


Figure A.4: Fridge model setup.

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 A.5b. The sensor process identifies whether the sensed temperature is *High*, *Med* or *Low*, see the timed automaton in Figure A.5c. The controller process is controlling whether the compressor should be turned *On* or *Off* via shortcutting a switch, see Figure A.5d. The switch process is relaying the signal to compressor by *turn_on* and *turn_off* like automaton in Figure A.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 A.5f and Figure A.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.

A.2.1 Properties of the Model

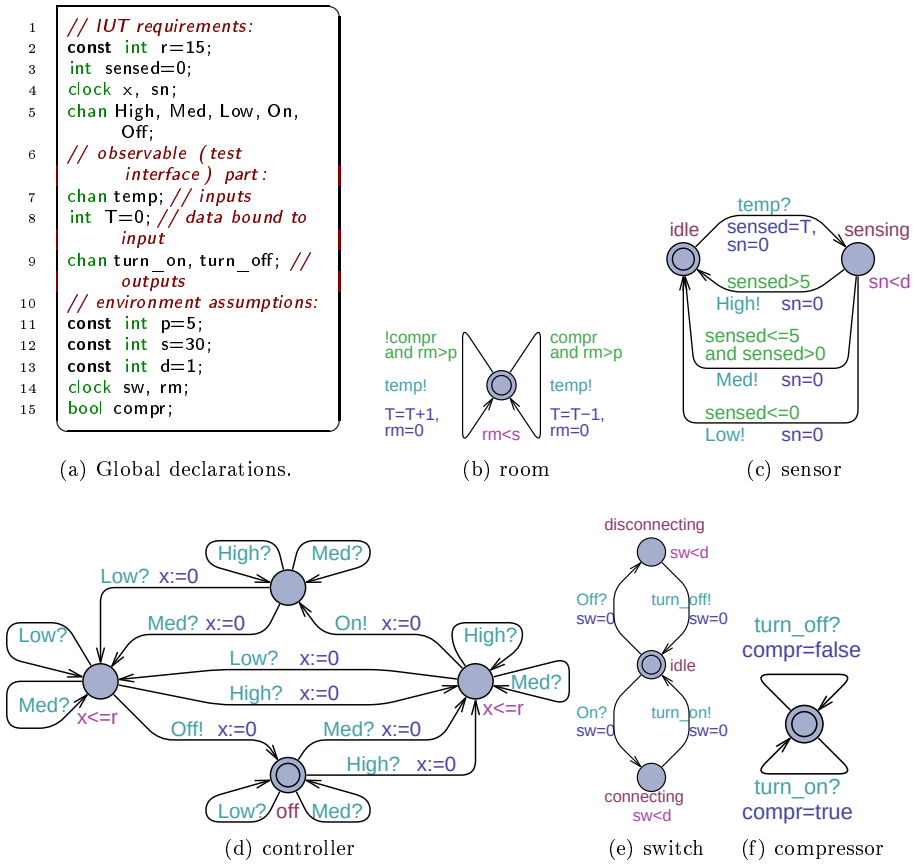
TRON allows non-determinism in the model. For some models the resulting state space can even be beyond the verification. For example, the requirements for the controller in Figure A.5d are non-deterministic in two ways:

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 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

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



(a) Global declarations. (b) room (c) sensor

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

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

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.

A.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* [25] 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 A.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] [24]. The diagram shows how processes are communicating where arrows indicate the direction of synchronization and data flow direction. 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.

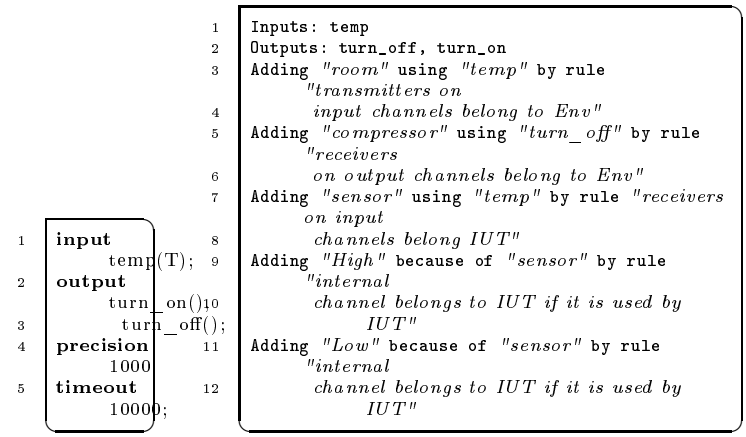
red items could not be partitioned consistently or have some suspicious properties (like variable is updated but is never read).

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 A.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



(a) `fridge.trn`

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

(b) Diagnostics sample.

```
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"
```

Figure A.6: The files in automatic model partitioning

error streams whose content is similar to Figure A.6b. The first two lines of Figure A.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 A.7 shows the sample image of the partitioning. The image might have different layout each time it is generated as *dot* gets different initial random seed.

A.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 A.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.

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

[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).

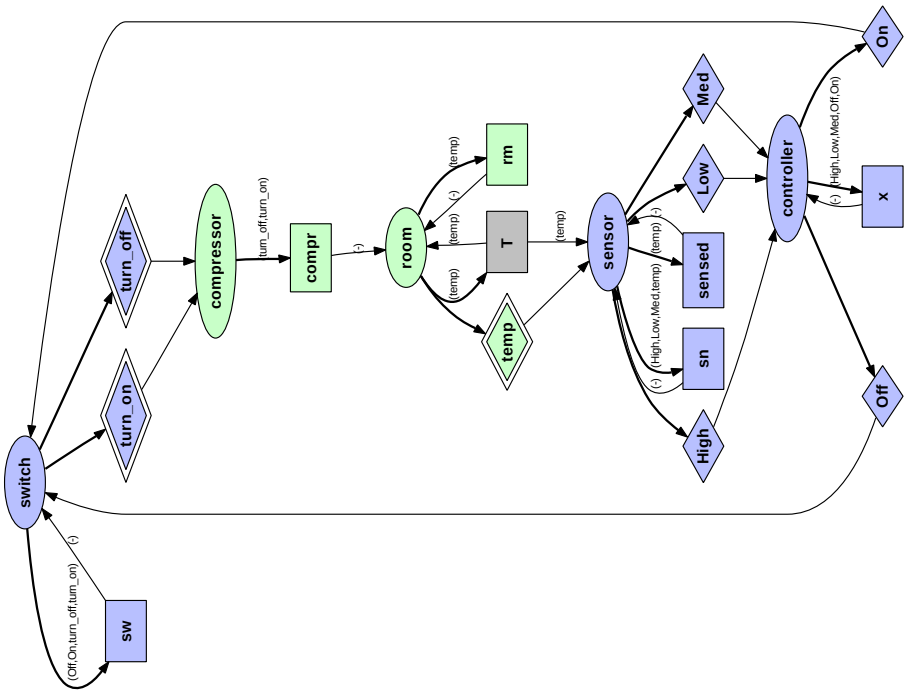


Figure A.7: Decorated signal flow diagram (`fridge.eps`) of the system model.

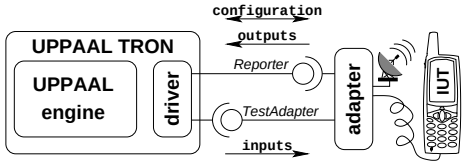


Figure A.8: Adapter API and physical interface.

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 A.1 summarizes advantages and disadvantages of adapter APIs. Textual API (Section A.3.4) 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 A.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 A.3.2) 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.

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 A.3.3).

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 A.1: Brief comparison of supported adapter APIs.

A.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 A.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 A.11) and command line arguments. It should create a `TestAdapter` interface to the adapter (see Figure A.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

```
1 extern "C" {
2     TestAdapter* adapter_new(Reporter* r, int argc, const char* args[] );
3     void adapter_delete(TestAdapter* adapter);
4 }
```

Figure A.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 A.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

```
1  struct TestAdapter {
2      void (*start)(TestAdapter* adapter);
3      void (*perform)(TestAdapter* adapter, int32_t chan, uint16_t n,
4                                            const int32_t data□ );
5      Reporter* const rep;
6      TestAdapter(Reporter* r): rep(r) { start = 0; perform = 0; }
7  };
```

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

`TestAdapter` interface is to inherit it (or extend in Java terms), provide `start` and `perform` (non-member) function implementations (which probably access `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 A.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.

Figure A.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 actions are executed during the sample test run:

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

```
1  struct Reporter {
2      void (*report_now)(Reporter*, int32_t chan, uint16_t n, const int32_t
           data[] );
3      int32_t (*getInputEncoding)(Reporter*, const char* inputChanName);
4      int32_t (*getOutputEncoding)(Reporter*, const char* outputChanName);
5      int32_t (*addVarToInput)(Reporter*, int32_t chan, const char* variable );
6      int32_t (*addVarToOutput)(Reporter*, int32_t chan, const char*
           variable);
7      int32_t (*setTimeUnit)(Reporter*, const int64_t& microsecs_per_unit);
8      int32_t (*setTimeout)(Reporter*, int32_t timeout_in_units);
9      const char* (*getErrorMessage)(Reporter*, int32_t error_code);
10 };
```

Figure A.11: Reporter: C-interface to UPPAAL TRON driver (`tron/adapter.h`).

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.

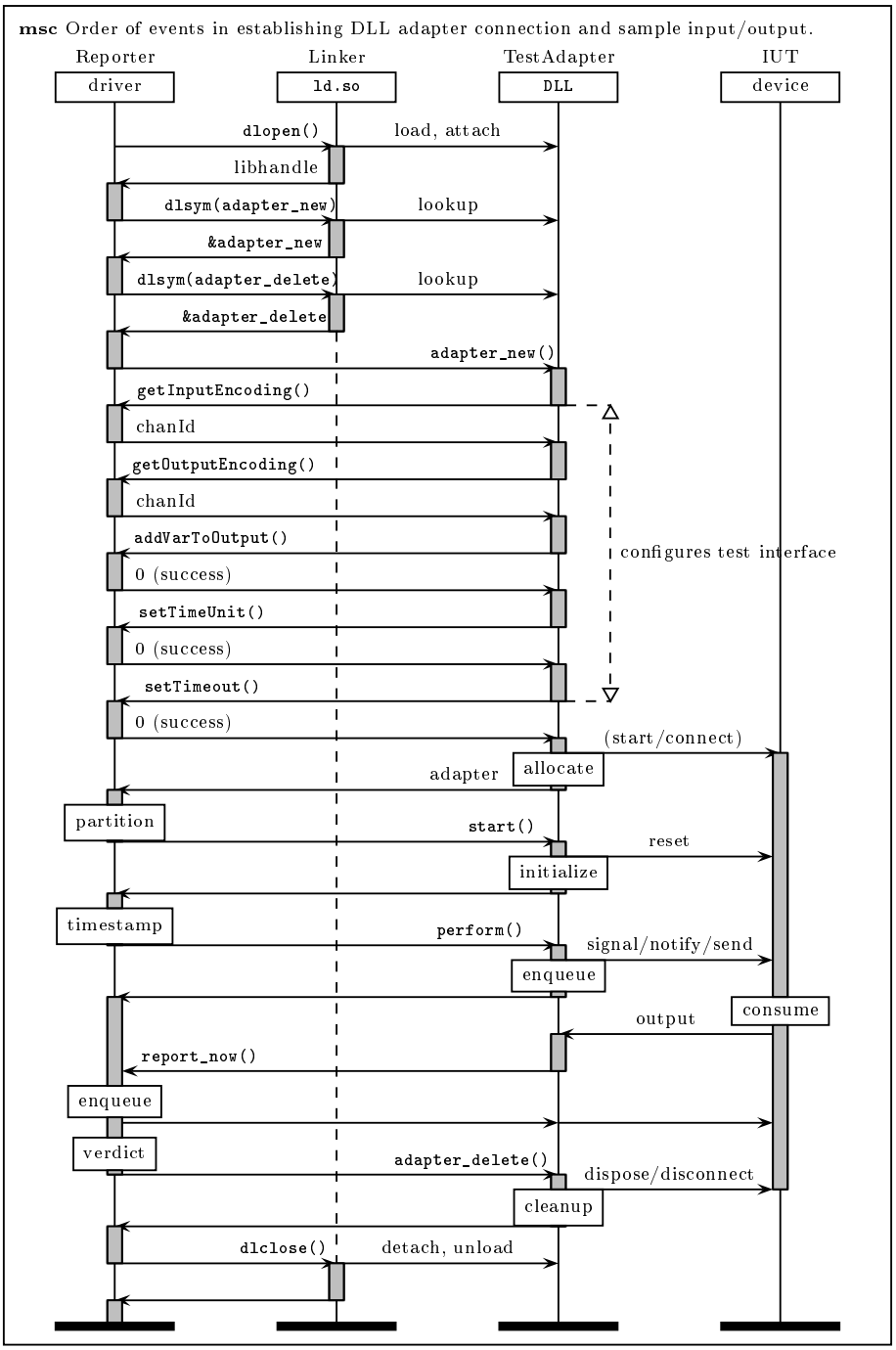


Figure A.12: Sample event sequence in dynamic library adapter during testing.

A.3.2 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 A.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.

A.3.3 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 A.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 A.13 shows the `Reporter` interface for Java programs. The base class `VirtualThread` denotes that it is also suitable for virtual time framework (see Section A.3.5 for details). In order to establish a connection to TRON, one

```
1  public class Reporter extends VirtualThread {
2      public Reporter(Adapter adapter, int port);
3      public Reporter(Adapter adapter, String host, int port);
4      public int addInput(String channel)  throws TronException, IOException;
5      public int addOutput(String channel) throws TronException, IOException;
6      public void addVarToInput(int channel, String variable)
7                                  throws TronException, IOException;
8      public void addVarToOutput(int channel, String variable)
9                                  throws TronException, IOException;
10     public void setTimeUnit(long microsecs)
11                                 throws TronException, IOException;
12     public void setTimeout(int timeout_in_units)
13                                 throws TronException, IOException;
14     public String getErrorMessage(int error_code);
15     public void report(int chanId);
16     public void report(int chanId, int[] params);
17     public boolean isConnected();
18     public void disconnect();
19     public void shutdown();
20     public void run();
21 }
```

Figure A.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 A.3.1. The methods may throw `IOException` upon usual socket connection problems or `TronException` (see Figure A.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.

```
1 public interface Adapter {
2     public void configure(Reporter reporter) throws TronException,
          IOException;
3     public void perform(int chanId, int[] params);
4 }
```

Figure A.14: Adapter: Java interface to adapter.

```
1 public class TronException extends IOException {
2     public TronException(String message) { super(message); }
3 }
```

Figure A.15: TronException thrown upon testing interface configuration error.

A.3.4 Interactive Text Interface

TRON has a build-in adapter called `TraceAdapter` for interacting via standard input and output streams. The adapter uses ANTLR [52] 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 A.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 A.6a shows an example of trace preamble. The 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 A.4.1 and Section A.4.2 to obtain such traces). The emulation mode declares all channels as inputs (even the

```
1 preamble: inputs outputs precision timeout ;
2 inputs    : "input" ( siglist )? ";" ;
3 outputs   : "output" ( siglist )? ";" ;
4 precision : "precision" INT ";" ;
5 timeout   : "timeout" INT ";" ;
6
7  siglist   : signature ("," signature)* ;
8 signature : ChanID "(" ( idlist )? ")" ;
9 idlist    : VarID ("," VarID)* ;
```

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

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 A.17a shows the grammar of language the `TraceAdapter` is expecting from standard input. The *trace* consists of a sequence of *command*s. Current

```
1  trace : (command)* ;
2  command : "input" expect ("," expect)* ";"
3            | "output" action ("," action)* ";"
4            | "delay" timestamp ("," expect)* ";"
5            ;
6
7  action    : ChanID "(" (valuelist )? ")" ;
8  valuelist : INT ("," INT)* ;
9
10 expect    : action (timestamp)? ;
11 timestamp: "@"? "[" time "," time "]" ;
12 time      : FLOAT | INT ;
```

(a) EBNF grammar of trace.

```
1  delay [2.0,3.0];
2  output trigger();
3  delay 11.0, reply()[0.0,10.0];
4  delay [0.0,1.0];
5  output send(4);
6  input receive(16);
7  output one2many();
8  delay [11.0, 15.0];
9  output many();
10 input reply()[0.0,0.0];
11 input reply()[0.0,0.0];
12 delay 10.0;
```

(b) Trace from `tracer` example.

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

`TraceAdapter` implementation supports three types of *command*s:

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[5].

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

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).

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 A.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.

A.3.5 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 [33] thread functions (Portable Operating System Interface 1003.1b-1993 realtime extension).

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

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.

```
#include <pthread.h>
#include <deque>
class MyMonitor {
  pthread_mutex_t lock;
  pthread_cond_t cond;
  std::deque<int> buffer;
  MyMonitor():
    lock(MUTEX_INITIALIZER),
    cond(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 A.18: Sample monitor implementations for producer-consumer problem.

- 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.

Dynamic Library IUT

TRON binary itself exports a set of functions necessary to implement POSIX-like monitor. Figure A.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 A.20 shows the list of symbols TRON is exporting. The symbols refer to corresponding POSIX function implementations and more. Almost all 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

```
1  int pthread_create(pthread_t*, pthread_attr_t*, void* (*start)(void*),
       void*);
2  int pthread_join(pthread_t, void**);
3  int pthread_mutex_init(pthread_mutex_t*, const
       pthread_mutexattr_t*);
4  int pthread_mutex_destroy(pthread_mutex_t*);
5  int pthread_mutex_lock(pthread_mutex_t*);
6  int pthread_mutex_unlock(pthread_mutex_t*);
7  int pthread_cond_init(pthread_cond_t*, const pthread_condattr_t*);
8  int pthread_cond_destroy(pthread_cond_t*);
9  int pthread_cond_wait(pthread_cond_t*, pthread_mutex_t*);
10 int pthread_cond_timedwait(pthread_cond_t*, pthread_mutex_t*,
       const struct timespec*);
11 int pthread_cond_signal(pthread_cond_t*);
12 int pthread_cond_broadcast(pthread_cond_t*);
13 int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Figure A.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, int port=0x1979);
19 int setSocketClock(const char* host, int port=0x1979, bool reg=true);
```

Figure A.20: TRON functions to replace a subset of POSIX.

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 A.3.5), 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 A.3.5). 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 A.18a to use virtual time framework.

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 A.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 A.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 A.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 A.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 A.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 A.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 A.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 A.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.

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 A.18b to use virtual time framework.

A.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.

A.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 A.4.2.

`-D path` specify a file path to store driver log (default /dev/null), see Section A.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. It is an optimization feature and the value can safely be very large (like testing timeout value) if there are few internal transitions in IUT model, however it should be limited to smaller delays if there are internal transition loops or similar many-transition structures. 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 almost no 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 A.3.4;

`SocketAdapter` remote TCP/IP socket adapter, see Section A.3.2.

`-P delay` specifies the delay choice strategy (see also Section A.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 `-P` is orthogonal to `-F` 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` turns on the logical (virtual) time framework. In this framework TRON also creates a virtual clock service on TCP/IP socket for remote processes. Parameter `log` specifies the default 6521 port number, the parameter can be replaced by a customized port number or even a `hostname:port` to

connect to remote virtual clock service (in case several TRON instances are used), where `hostname` is the name of the remote host and `port` is the remote port number. See Section A.3.5 for details about TRON's virtual clock services.

`-S filename` Append the verdict, I/O and duration to file (default /dev/null), see Section A.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 [25] graph, expects formated standard input (see Section A.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 A.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` Experimental option for automatic adapter abstraction (see Section A.4.3). Option specify 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 A.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.

A.4.2 Logging

There are four ways to log test runs:

Engine log contains information about operations performed in the 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 A.16 and A.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 A.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.

A.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 in order to determine correctness using I/O conformance relation. TRON offers over-approximating method to match real time values into model time that is sound, i.e. it records all I/O instances with available precision and allows potentially false test passes (limited by timing measurement precision of each individual I/O) but does not introduce false failure announcements (non-conformance verdicts). 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.

Virtual Time

Virtual time framework provides ideal "lab" conditions for testing experiments by removing the computation time, scheduling and communication latency disturbances in I/O timing. It allows to focus solely on the actual I/O timing and is therefore simplest to introduce first.

Consider the following input offering scenario shown in message sequence chart (MSC) in Figure A.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. The driver passes the input i to the adapter without delay 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 $[L_e, U_e]$, which is potentially much narrower interval than $[L_i, U_i]$). The actual mapping is explained in Section A.4.3.
12. TRON asks UPPAAL to update (affectively filter and constrain) the state set to describe system states within model time interval $[L_e, U_e]$.

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.

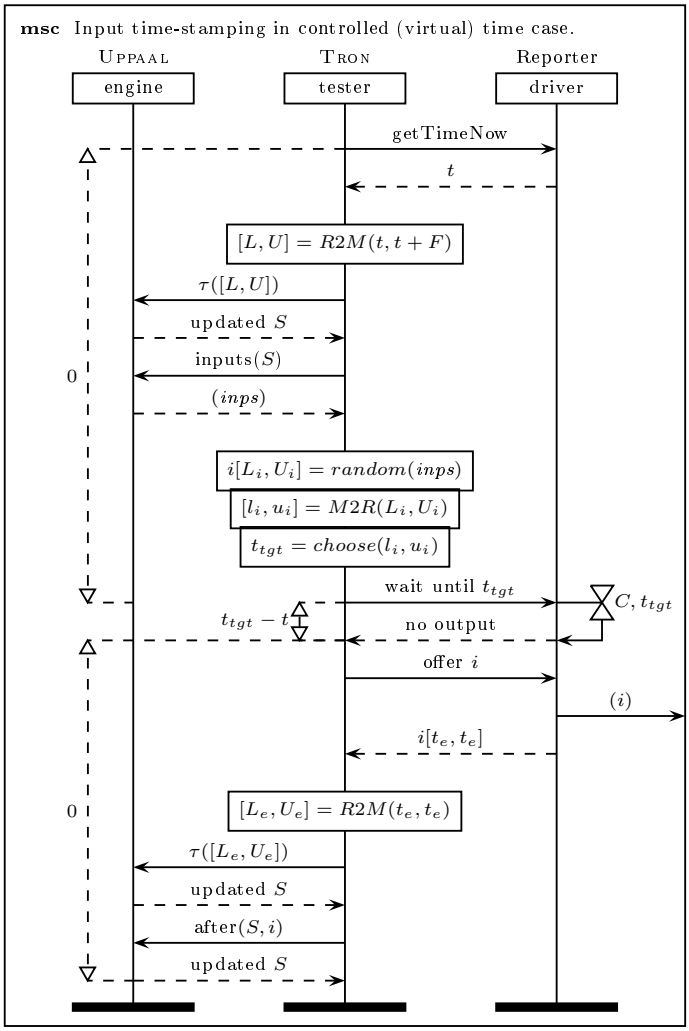


Figure A.21: Scenario for offering an input to IUT and relevant timestamps in virtual time case.

Naïve Real Time

From Figure A.21 it is evident that in virtual time framework the time spent for computing, choosing and scheduling 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 A.22 shows

the input offering scenario adjusted for real time, which differs from virtual time in the following ways:

1. The calculation time for calculations is hardly predictable 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 measured by TRON. 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 hope 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.

Internal Latency

So far, we still rely on the fact that TRON is woken up at precisely t_{tgt} moment and further input delivery 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 user supplied OS dependent estimate `-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.

External Latency

Often the test adapter introduces significant delays (communication latency) and I/O buffering. Since TRON has almost no control of adapter part, a fair way to reflect such delays is to model test adapter as part of IUT. A straightforward adapter modeling is to provide an explicit model in the system specification (e.g. add timed automata processes for adapter). Typical adapter receives a signal, puts it into buffer, delays the signal (signal is "on the wire") and forwards the signal to destination process. In this section we show how to acquire I/O timing characteristics of such adapter.

Figure A.23 shows how the IUT and tester use digital clocks to timestamp I/O events. For simplicity we assume a perfect digital clock, which updates the time value with a period of it's resolution, and time is synchronized globally, i.e. the values on different time-lines but on the same vertical line have the same absolute time value. The IUT sends output at t_1 while its clock with resolution

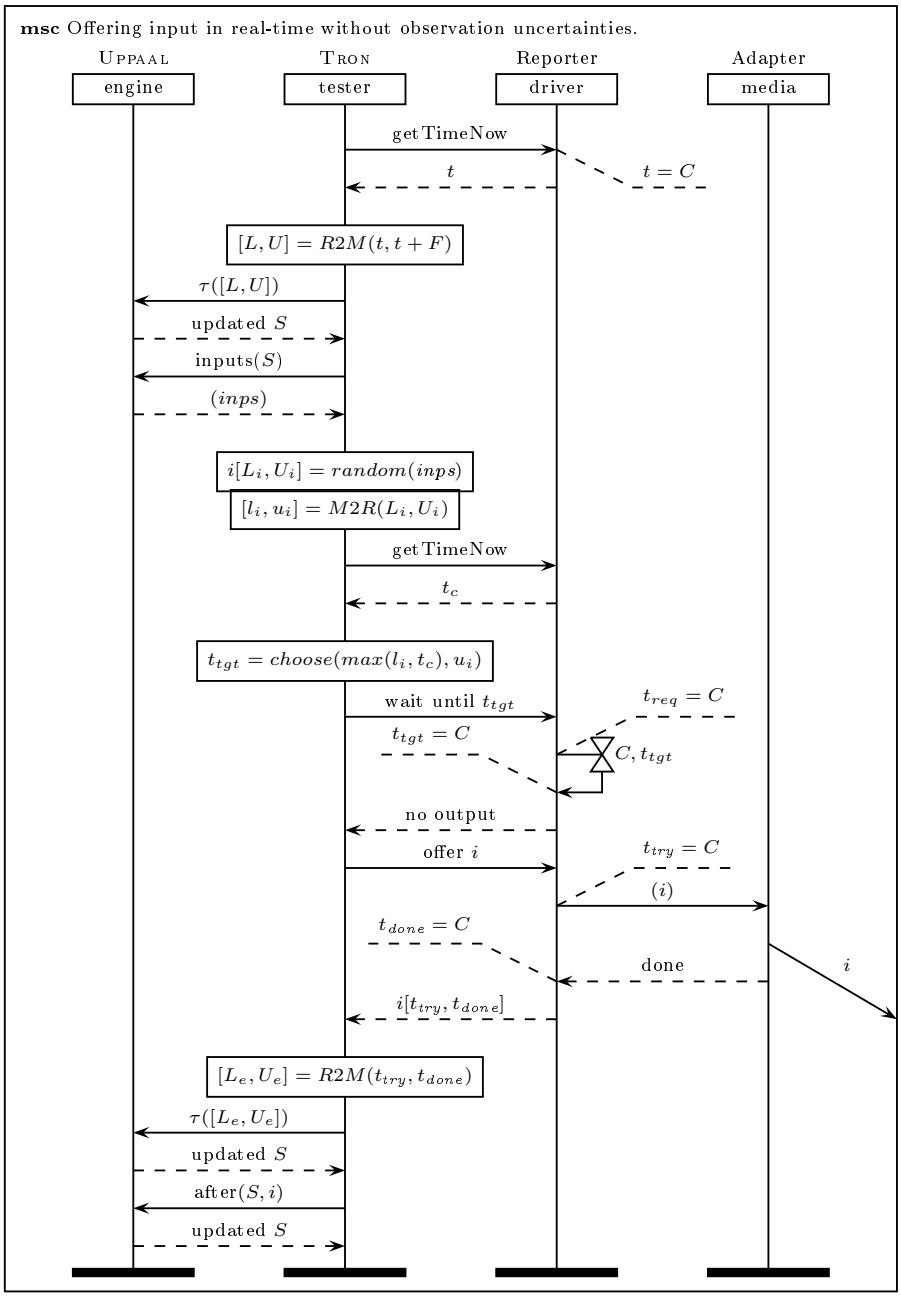


Figure A.22: Scenario for offering an input to IUT and relevant timestamps in real time case without observation uncertainties.

R_1 is showing t_2, the output is delayed by the adapter by duration D_1 and sensed by the tester at t_3 while tester's clock with resolution R_2 is showing t_4. Before sending input the 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

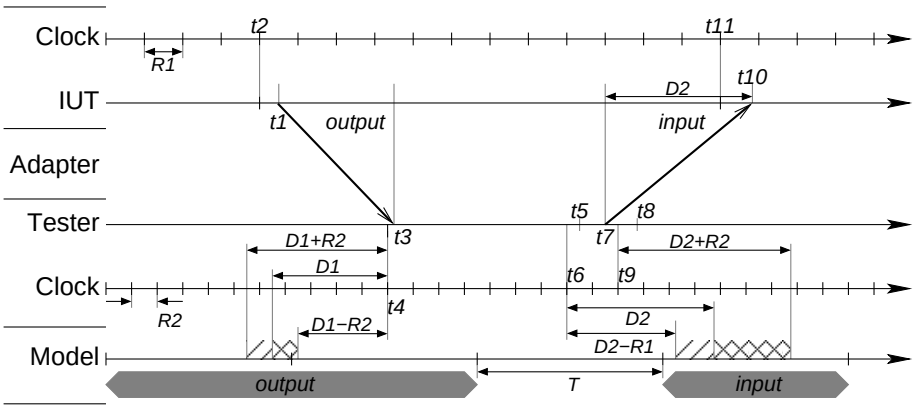


Figure A.23: I/O delays and time-stamps in the adapter.

then mapped onto model time scale with resolution of T (real time value of one model time unit).

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 any delay, otherwise this deterministic delay can be added to adapter delay. At the IUT side I/O happens at t_1 and t_{10} instances, however due to its digital clock time sampling the IUT may think it happens at t_2 and t_{11}. Similarly at tester side I/O happens at t_3 (output) and t_7 (input), while tester timestamps these events at t_4 (output) and $[t_6, t_9]$ (input). Then observe the following inequalities over timestamps:

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

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

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 A.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 the 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:

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

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.

Automatic Adapter Abstraction

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 A.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

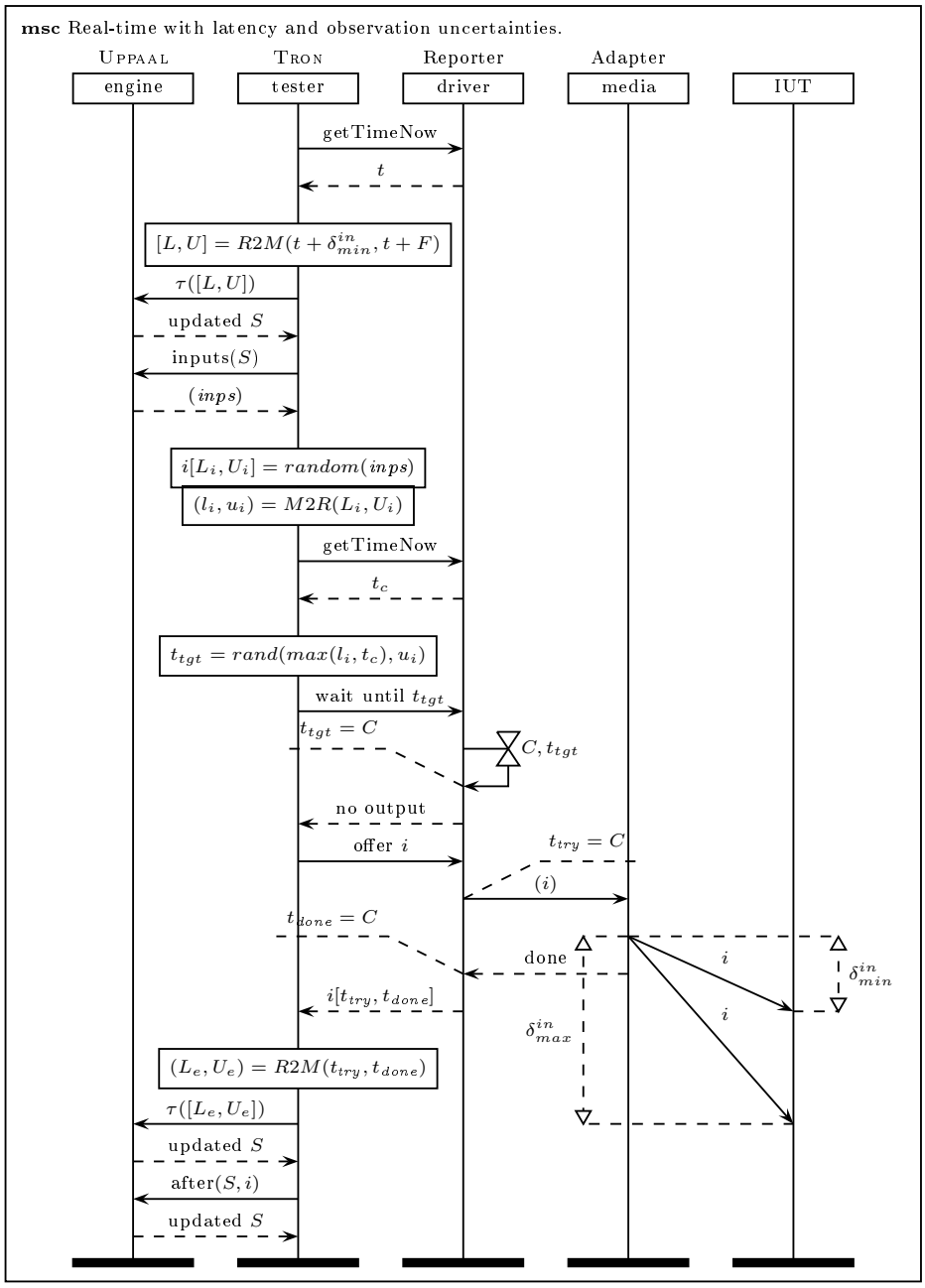


Figure A.24: Scenario for offering an input to IUT and relevant timestamps in real time case with observation uncertainties, assuming $F \geq \delta^{inp}_{max}$.

$u_i - \delta^{inp}_{max} - \mathcal{L} < l_i$ 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 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.

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:

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

From Figure A.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*:

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

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

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

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

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

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

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

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

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).

A.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.

A.5 Diagnostics

Currently TRON provides a verdict and simple conclusion based on last good state set. Algorithm 5 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.

Where t_I, t_O, t_T and t_S are:

t_S – the largest permissible delay for IUT without observable I/O.

t_O – the largest permissible delay for IUT output.

t_T – the largest permissible delay for the environment without inputs, i.e. this is how much tester can delay at most without issuing any input. Such delay is determined by `ChoiceFilter` which computes the system's behavior without IUT invariants.

t_I – the largest permissible delay for the input by the environment, computed by `ChoiceFilter`. If the set of input choices is empty, then t_0 is taken instead.

A.6 Limitations and Workarounds

A.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 A.2.2).

A.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

Algorithm 5: Verdict based on a last good state set.

Input: StateSet backup, Event e, Choice c
Output: verdict: Passed, Failed or Inconclusive

A_{inp} =EnvOutput(backup); A_{out} =ImpOutput(backup);
if e **then** `// state set empty upon observable I/O`
  if *e.isInput* **then** `// if e is input, then there was a choice`
    "Decided to input c, but executed as e.channel@[e.lowerBound,e.upperBound)";
    "The target state was: c.targetState";
    if $c.maxBound < e.lowerBound$ **then**
      return *Inconc(Input executed too late)*;
    else if $e.upperBound < c.minBound$ **then**
      return *Inconc(Input executed too early)*;
  else `// e is an output`
    "Got unacceptable output e.channel@[e.lowerBound,e.upperBound)";
    "Expected outputs: A_{out}";
    boolean tooLate=false, tooEarly=false;
    forall $c_o \in A_{out}$ *s.t.* $e.channel == c_o.channel$ **do** `// see outputs`
      if $e.upperBound < c_o.minBound$ **then** tooEarly=true;
      if $e.lowerBound > c_o.maxBound$ **then** tooLate=true;
    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, only time delay`
  "Last time-window is beyond maximum allowed delay";
  if $t_S < t_O$ **then**
    return *Inconc(Bug: output deadline behind allowed delay)*;
  else if $t_O < t_S$ **then**
    return *Inconc(Model contains time lock)*
  else if $t_S < t_T$ **then**
    return *Failed(IUT failed to send output in time)*
  else if $t_I < t_O$ **then**
    return *Failed(IUT failed to send output in time)*
  else return *Inconc(Model contains deadlock)*
return *Inconc(Empty stateset. Bug, please report it.);*

being actively developed (see e.g. Ingo Molnar's work on O(1) and CFS schedulers). 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 A.3.5 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.