Databases and Information Systems VIII
H.-M. Haav et al. (Eds.)
© 2014 The authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License.
doi:10.3233/978-1-61499-458-9-289

Provably Correct Test Development for Timed Systems

Jüri VAIN[a,1], Aivo ANIER[a] and Evelin HALLING[a]
[a] *Department of Computer Science, Tallinn University of Technology, Estonia*

Abstract. Automated software testing is an increasing trend for improving the productivity and quality of software development processes. That, in turn, rises issues of trustability and conclusiveness of automatically generated tests and testing procedures. The main contribution of this paper is the methodology of proving the correctness of tests for remote testing of systems with time constraints. To demonstrate the feasibility of the approach we show how the abstract conformance tests are generated, verified and made practically executable on distributed model-based testing platform dTron.

Keywords. model-based testing, provably correct test generation, timed automata, verification by model checking

Introduction

The growing competition in software market forces manufacturers to release new products within shorter time frame and with lower cost. That imposes hard pressure to software quality. Extensive use of semi-automated testing approaches is an attempt to improve the quality of software and related development processes in industry. Although a wide spectrum of commercial and academic tools are available, the testing process still involves strong human factor and remains prone to human errors. Even fully automated approaches cannot guarantee trustable and conclusive testing unless the test automation is correct by construction or exhaustively covered with correctness checks. Test automation and test correctness are the main subjects of study in model based testing (MBT). Generally, MBT process comprises following steps: modelling the system under test, referred as Implementation-Under-Test (IUT), specifying the test purposes, generating the tests and executing them against IUT.

In this paper we study how the correctness of test derivation steps can be ensured and how to make the test results trustable throughout the testing process. In particular, we focus on model-based online testing of software systems with timing constraints capitalizing on the correctness of the test suite through test development and execution process. In case of conformance testing the IUT is considered as a black-box, i.e., only the inputs and outputs of the system are externally controllable and observable respectively. The aim of black-box conformance testing [1] is to check if the behaviour observable on sys-

[1]Corresponding Author: Jüri Vain; Department of Computer Science, Tallinn University of Technology, Akadeemia tee 15A, 12618 Tallinn, Estonia; E-mail: juri.vain@ttu.ee

tem interface conforms to a given requirements specification. During testing a tester executes selected test cases on an IUT and emits a test verdict (pass, fail, inconclusive). The verdict is computed according to the specification and a input-output conformance relation (IOCO) between IUT and the specification. The behaviour of a IOCO-correct implementation should respect after some observations following restrictions: (i) the outputs produced by IUT should be the same as allowed in the specification; (ii) if a quiescent state (a situation where the system can not evolve without an input from the environment [2]) is reached in IUT, this should also be the case in the specification; (iii) any time an input is possible in the specification, this should also be the case in the implementation.

The set of tests that forms a test suite is structured into test cases, each addressing some specific test purpose. In MBT, the test cases are generated from formal models that specify the expected behaviour of the IUT and from the coverage criteria that constrain the behaviour defined in IUT model with only those addressed by the test purpose. In our approach Uppaal Timed Automata (UPTA) [3] are used as a formalism for modelling IUT behaviour. This choice is motivated by the need to test the IUT with timing constraints so that the impact of propagation delays between the IUT and the tester can be taken into account when the test cases are generated and executed against remote real-time systems. Another important aspect that needs to be addressed in remote testing is functional non-determinism of the IUT behaviour with respect to test inputs. For non-deterministic systems only online testing (generating test stimuli on-the fly) is applicable in contrast to that of deterministic systems where test sequences can be generated offline. Second source of non-determinism in remote testing of real-time systems is communication latency between the tester and the IUT that may lead to interleaving of inputs and outputs. This affects the generation of inputs for the IUT and the observation of outputs that may trigger a wrong test verdict. This problem has been described in [4], where the Δ-testability criterion (Δ describes the communication latency) has been proposed. The Δ-testability criterion ensures that input/output interleaving never occurs.

1. Preliminaries

1.1. Uppaal Timed Automata

Uppaal Timed Automata (UPTA) [3] are widely used as one of the main modelling formalism for representing time constraints of software intensive systems. Before delving into test construction we shortly introduce the syntax and semantics of UPTA.

A timed automaton is given as a tuple $(L;E;V;Cl;Init;Inv;T_L)$. L is a finite set of locations, E is the set of edges defined by $E \in L \times G(Cl,V) \times Sync \times Act \times L$, where $G(Cl,V)$ is the set of transition enabling conditions - guards. *Sync* is a set of synchronization actions over channels. In the graphical notation, the locations are denoted by circles and transitions by arrows. An action *send* over a channel h is denoted by $h!$ and its co-action *receive* is denoted by $h?$. *Act* is a set of sequences of assignment actions with integer and boolean expressions as well as with clock resets. V denotes the set of integer and boolean variables. Cl denotes the set of real-valued clocks ($Cl \cap V = \emptyset$).

$Init \subseteq Act$ is a set of assignments that assigns the initial values to variables and clocks. $Inv : L \rightarrow I(Cl,V)$ is a function that assigns an invariant to each location, $I(Cl,V)$ is the set of invariants over clocks Cl and variables V. $T_L \rightarrow \{ordinary, urgent, committed\}$ is the function that assigns the type to each location of the automaton.

We can now define the semantics of UPTA in the way presented in [3]. A clock valuation is a function $val_cl : Cl \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. A variable valuation is a function $val_v : V \to \mathbb{Z} \cup BOOL$ from the set of variables to integers and booleans. Let $\mathbb{R}^{Cl}$ and $(\mathbb{Z} \cup BOOL)^V$ be the sets of all clock and variable valuations, respectively. The semantics of an UPTA is defined as a LTS $(\sum, init, \to)$, where $\sum \subseteq L \times \mathbb{R}^{Cl} \times (\mathbb{Z} \cup BOOL)^V$ is the set of states, the initial state $init = Init(cl, v)$ for all $cl \in Cl$ and for all $v \in V$, with $cl = 0$, and $\to \subseteq \sum \times \{\mathbb{R}_{\leq 0} \cup Act\} \times \sum$ is the transition relation such that:

$(l, val_{cl}, val_v) \to (l, val_{cl} + d, val_v)$ if $\forall d' : 0 \leq d' \leq d \Rightarrow val_{cl} + d \models Inv(l)$,
$(l, val_{cl}, val_v) \to (l', val'_{cl}, val'_v)$ if $\exists e = (l, act, G(cl, v), r, l') \in E$ i.e.
$val_{cl}, val_v \models G(cl, v), val'_{cl} = [re \to 0]val_{cl}$, and $val'_{cl}, val'_v \models Inv(l')$,

where for delay $d \in \mathbb{R}_{\geq 0}, val_{cl} + d$ maps each clock cl in Cl to the value $val_{cl} + d$, and $[re \to 0]val_{cl}$ denotes the clock valuation which maps (resets) each clock in re to 0 and agrees with val_{cl} over $Cl \backslash re$.

1.2. *Test Generation for On-line Testing*

Reactive on-line testing means that the tester program has to react to observed outputs of the IUT and to possible changes in the test goals on-the-fly. The rationale behind the reactive planning method proposed in [5] lies in combining computationally hard offline planning with time bounded online planning phases. Off-line phase is meant to shift the computationally hard planning as much as possible in the test preparation phase. Here the static analysis results of IUT model and the test goal are recorded in the format of compact planning rules that are easy to apply later in the on-line phase. The on-line planning rules synthesized must ensure close to optimal test runs and termination of the test case when a prescribed test purpose is satisfied.

The RPT synthesis algorithm introduced in [5] assumes that the IUT model is an output observable non-deterministic state machine ([6]). Test purpose (or goal) is a specific objective or a property of the IUT that the tester is set out to test. Test purpose is specified in terms of test coverage items. We focus on test purposes that can be defined as a set of boolean "*trap*" variables associated with the transitions of the IUT model ([7]). The goal of the tester is to drive the test so that all traps are visited at least once during the test run.

The tester synthesis method outputs tester model as UPTA where the rules for online planning are encoded in the transition guards called gain guards. The gain guard evaluates *true* or *false* at the time of execution of the tester determining if the transition can be taken from the current state or not. The value *true* means that taking the transition with the highest gain is the best possible choice to reach unvisited traps from current state. The decision rules for on-the-fly planning are derived by performing reachability analysis from the current state to all trap-equipped transitions by constructing the shortest path trees. Since at each test execution step only the guards associated with the outgoing transitions of the current state are evaluated, the number of guard conditions to be evaluated at one planning step is relatively small (equal to the location-local branching factor in the worst case). To implement such a gain guided model traversal, the gain guard is constructed using (model and goal specific) gain functions and the standard function max that return the maximum of those gain values that characterize alternative test paths.

Technically, the gain function of a transition returns a value that depends on the distance-weighted reachability of the unvisited traps from the given transition. The gain guard of the transition is *true* if and only if that transition is a prefix of the test sequence with highest gain among those that depart from the current state. If the gain functions of several enabled transitions evaluate to same maximum value the tester selects one of these transitions using either random selection or "least visited first" principle. Each transition in the model is considered to have a weight and the gain of test case is proportional to the length and the sum of weights of whole test sequence.

The RPT synthesis comprises three main steps (Figure 1):

1. extraction of the RPT control structure,

2. constructing gain guards,

3. reduction of gain guards according to the parameter *"planning horizon"* that defines the pruning depth of the planning tree.

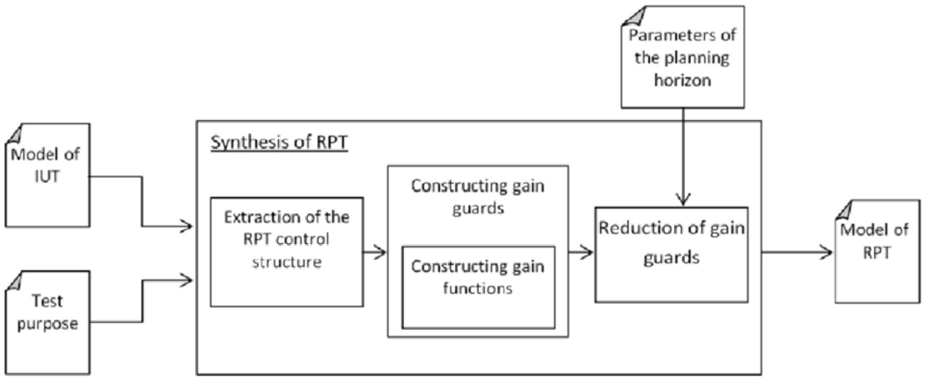
Parameters of the planning horizon
Synthesis of RPT
Model of IUT
Test purpose
Extraction of the RPT control structure
Constructing gain guards
Constructing gain functions
Reduction of gain guards
Model of RPT

Figure 1. RPT synthesis workflow

In the first step, the RPT synthesiser analyses the structure of the IUT model and generates the RPT control structure. In the second step, the synthesizer finds possibly successful IUT runs for reaching the test goal.

Last step of the synthesis reduces the gain functions pruning the planning tree up to some predefined depth that is given by parameter *"planning horizon"*. Since the RPT planning tree has the longest branch proportional to the length of Euler's contour in the IUT model control graph the gain function's recurrent structure may be very complex and for practical purposes needs to be bounded by some planning horizon. Traps being beyond the planning horizon still contribute in the gain function value but their distance is just ignored. Thus, for deep branches of planning tree the gain function returns an approximation of the gain value.

2. Correctness of IUT Models

2.1. Modelling Timing Aspects of IUT

For automated testing of input-output conformance of systems with time constraints we restrict ourselves with a subset of UPTA that simplifies IUT model construction. Namely, we use a subset where the data variables, their updates and transition guards on data variables are abstracted away. We use the clock variables only and the conditions expressed by clocks and synchronization labels. An elementary modelling pattern for representing IUT behaviour and timing constraints is Action pattern (or simply Action) depicted in Figure 2.

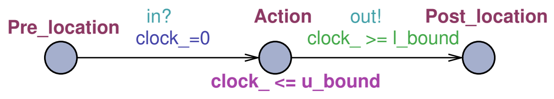

Figure 2. Elementary modelling fragment "Action"

An Action models a program fragment execution on a given level of abstraction as one atomic step. The Action is triggered by input event and it responds with output event within some bounded time interval (response time). The IUT input events (stimuli in testing context) are generated by Tester, and the output events (IUT responses) are to make the reactions of IUT observable to Tester. In UPTA, the interaction between IUT and Tester is modelled by synchronous channels that mediate input/output events. Receiving an input event from channel *in* is denoted by *in*? and sending an output event via channel *out* is denoted by *out*!.

The major timing constraint we represent in IUT model is *duration* of the Action. To make the specification of durations more realistic we represent it as a closed interval $[l\_bound, u\_bound]$, where *l_bound* denotes a lower bound and *u_bound* an upper bound of the interval. Duration interval $[l\_bound, u\_bound]$ can be expressed in UPTA as shown in Figure 2. Clock reset "$clock = 0$" on the edge "$Pre\_location \rightarrow Action$" makes the time constraint specification local to the Action and independent from current value at earlier execution steps. An invariant "$clock \leq u\_bound$" of location "$Action$" forces the Action to terminate latest at time instant *u_bound* after the clock reset and guard "$clock \geq l\_bound$" of the edge "$Action \rightarrow Post\_location$" defines the earliest time instant w.r.t. clock reset when the outgoing transition of Action can be executed.

From tester's point of view IUT has two types of locations: passive and active. In passive locations IUT is waiting for test stimuli and in active locations IUT chooses its next moves, i.e. presumably it can stay in that location as long as specified by location invariant. The location can be left when the guard of outgoing transition $Action \rightarrow Post\_location$ evaluates to $true$. In Figure 2, the locations $Pre\_location$ and $Post\_location$ are passive while $Action$ is an active location.

We compose IUT models from Action pattern using two types of composition rules: *sequential* and *alternative composition*.

Definition 1. Composition of Action patterns.

Let F_i and F_j be UPTA fragments composed of Action patterns (incl. elementary Action) with pre-locations l_i^{pre},l_j^{pre} and post-locations l_i^{post},l_j^{post} respectively, their composition is the union of elements of both fragments satisfying following conditions:

- sequential composition $F_i;F_j$ is UPTA fragment where $l_i^{post} = l_j^{pre}$;
- alternative composition $F_i + F_j$ is UPTA fragment where $l_i^{pre} = l_j^{pre}$ and $l_i^{post} = l_j^{post}$.

The test generation method we highlighted in Section 1.2 relies on the notion of well-formedness of the IUT model according to the following inductive definition.

Definition 2. Well-formedness (wf) of IUT models

- atomic Action pattern is well-formed;
- sequential composition of well-formed patterns is well-formed;
- alternative composition of well-formed patterns is well-formed if the output labels are distinguishable;

Proposition 1. Any UPTA model M with non-negative time constraints and synchronization labels that do not include state variables can be transformed to bi-similar to it well-formed representation $wf(M)$.

Note without the detailed proof that for those locations and edges of UPTA that do not match with the Definition 2, the well-formedness needs adding auxiliary pre-, and post-locations and ε-transition, that do not violate the i/o behaviour of original model. For representing internal actions that are not triggered by external events (their incoming edge is ε-labelled) we restrict the class of pre-locations with type "committed". In fact, the subclass of models transformable to well-formed is broader than given by Definition 2, including also UPTA that have data variable updates, but in general wf-form does not extend to models that include guards on data variables.

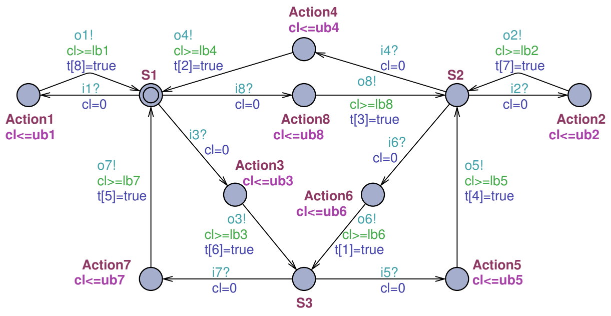

Figure 3. Simple example of well-formed IUT model

In the rest of paper, we assume for test generation that M^{IUT} is well-formed and denote this fact by $wf(M^{IUT})$. An example of such an IUT model we use throughout the paper is depicted in Figure 3.

2.2. Correctness of IUT Models

The test generation method introduced in [5] and developed further for EFSM models in [8] assumes that the IUT model is connected, input enabled, output observable and strongly responsive. In the following we demonstrate how the validity of these properties usually formulated for IOTS (Input-Output Transition System) models can be ensured for well-formed UPTA models (see Definition 2).

2.2.1. Connected Control Structure and Output Observability

We say that UPTA model is connected in the sense that there is an executable path from any location to any other location. Since the IUT model represents an open system that is interacting with its environment we need for verification by model checking a non-restrictive environment model. According to [9] such an environment model has the role of canonical tester. Canonical tester provides test stimuli and receives test responses in any possible order the IUT model can interact with its environment. A canonical tester can be easily generated for well-formed models according to the pattern depicted in Figure 4b (this is canonical tester for the model shown in Figure 3).

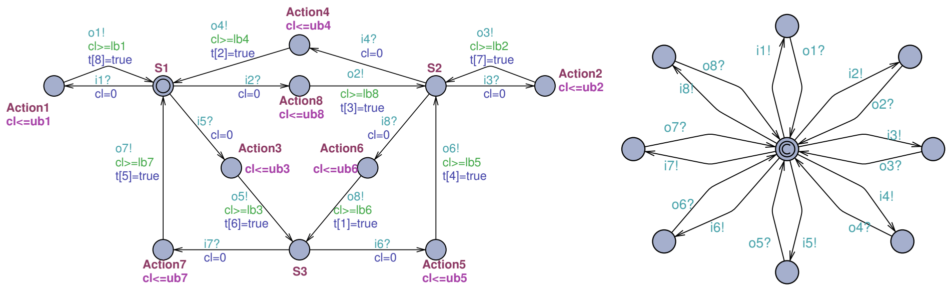

Figure 4. Synchronous parallel composition of a) IUT and b) canonical tester models

The canonical tester implements the "random walk" test strategy that is useful in endurance testing but it is very inefficient when functionally or structurally constrained test cases need to be generated for large systems.

Having synchronous parallel composition of IUT and the canonical tester (shown in Figure 4) the connectedness of IUT can be model checked with query (1) that expresses the absence of deadlocks in interactions between IUT and canonical tester.

$$A[]\, not\ deadlock \tag{1}$$

The output observability condition means that all state transitions of the IUT model are observable and identifiable by the outputs generated by these transitions. Observability is ensured by the definition of well-formedness of the IUT model where each input event and Action location is followed by the edge that generates a locally (w.r.t. source location) unique output event.

2.2.2. Input Enabledness

Input enabledness assumption means that blocking due to irrelevant test input during test execution is avoided. Naive way of implementing this assumption in IUT models presumes introducing self-looping transitions with input labels that are not represented on other transitions that share the same source state. That makes IUT modelling tedious and leads to the exponential increase of the M^{IUT} size. Alternatively, when relying on the notion of observational equivalence one can approximate the input enabledness in UPTA by exploiting the semantics of synchronizing channels and encoding input symbols as boolean variables $I_1 ... I_n \in \Sigma$. Then the pre-location of the Action pattern (see Figure 2) needs to be modified by applying the Transformation 1.

2.2.3. Transformation 1

- assume there are k outgoing edges from pre-location l_i^{pre} of $Action_i$, each of these transitions is labeled with some input $I_1 ... I_k \in \Sigma^i(Action_i) \subseteq \Sigma$;
- we add a self-looping edge $l_i^{pre} \rightarrow l_i^{pre}$that models acceptance of all inputs in Σ except those in Σ^i. Because of that we specify the guard of edge $l_i^{pre} \rightarrow l_i^{pre}$ as boolean expression: $not(I_1 \vee ... \vee I_k)$.

Provided the outgoing branching factor $\mathscr{B}_i^{out}$ of l_i^{pre} is, as a rule, substantially smaller than $|\Sigma|$ we can save $|\Sigma| - \mathscr{B}_i^{out} - 1$ edges at each pre-location of Action patterns. Note that by wf-construction rules the number of pre-locations never exceeds the number of actions in the model. That is due to alternative composition that merges pre-locations of the composition. A fragment of alternative composition accepting inputs in Σ^i with described additional edge for accepting symbols in $\Sigma \setminus \Sigma^i(Action_i)$ is depicted in Figure 5 (time constraints are ignored here, I_1 and I_2 in the figure denote predicates $Input == i_1$ and $Input == i_2$ respectively).

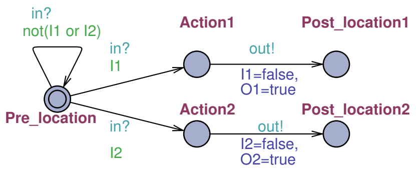

Figure 5. Input enabled fragment

2.2.4. *Strong Responsiveness*

Strong responsiveness (SR) means that there is no reachable livelock (a loop that includes only ε-transitions) in the IUT model M^{IUT}. In other words, M^{IUT} should always enter the quiescent state after finite number of steps. Since transforming M^{IUT} to $wf(M^{IUT})$ does not eliminate ε-transitions there is no guarantee that $wf(M^{IUT})$ is strongly responsive by default. To verify the SR propety of M^{IUT} we apply Algorithm 1.

2.2.5. *Algorithm 1*

1. According to the Action pattern of Figure 5 the information of i/o events is specified using synchronization channel *in* and a boolean variable that represents receiving an input symbol I_i. Since Uppaal model checker is state based we need recording the occurrence of input events in states. Therefore, the boolean variable representing an input event is kept true in the destination location of the edge that is labelled with given event and reset $false$ immediately after leaving this location. For same reason the ε-transitions are labelled with update $EPS = true$ and following output edge with update $EPS = false$.
2. Next, we reduce the model by removing all the edges and locations that are not involved in the traces of model checking query: $l_0 \models E\Box EPS$, where l_0 denotes initial location of M^{IUT}. The query checks if any ε-transition is reachable from l_0 (necessary condition for violating SR-property).
3. Further, we remove all non ε-transitions and locations that remain isolated thereafter.
4. Remove recursively all locations that do not have incoming edges (their outgoing edges will be deleted with them).
5. After reaching the fixed point of recursion of step 4 we check if the remaining part of model is empty. If yes then we can conclude that M^{IUT} is strongly responsive, otherwise it is not.

It is easy to show that all steps except step 2 are of linear complexity in the size of the M^{IUT}.

3. Correctness of RPT Tests

3.1. Functional Correctness of Generated Tests

The tester program generated based on IUT model can be characterized using some test coverage criteria it is designed for. As shown in Section 1.2, the RPT generating algorithm is aimed at structural coverage of IUT model elements and can be expressed by means of boolean "trap" variables. To recall, the traps are assignment expressions of boolean trap variables and the valuation of traps indicates the status of the test run. For instance, one can observe if the edges labeled with them are already covered or not in the course of test run. Thus, the relevant correctness criterion for the tester generated is its ability to cover traps.

Definition 3. Coverage correctness of the test.

We say that the RPT tester is coverage correct if the test run covers all the transitions that are labelled with traps in IUT model.

Definition 4. Optimality of the test.

We say that the test is length (time) optimal if there is no shorter (accordingly faster) test runs among all those being coverage correct.

We can show that the RPT method generates tests that are coverage correct (and in general, close to optimal) by construction, if the planning horizon of gain function is greater or equal to the depth of reduced reachability tree of M^{IUT}. Though, the practical limit of planning depth is set by Uppaal tool where the largest integer value of type '$long$' is 2^{31}. That allows distinctive encoding of gain function co-domain for test paths up to depth 31. It means that if the IUT is fully connected and deterministic RPT provides a test path that covers all traps length-optimally. In non-deterministic case it provides the best strategy against any legal strategy the IUT chooses (legal in this context means that any behaviour of IUT either conforms to its specification or is detectably violating it).

While the reachability tree exceeds given by the horizon depth limit the gain function becomes stochastic (insensible to reachability tree structure deeper than the horizon). It is distinctive on the number of deeper traps only, but it is not distinctive on their co-reachability. Even though, the planning method with cross horizon depth has shown to be statistically efficient by providing close to optimal test paths in large examples there is threat of choosing infeasible paths if the model is not well-formed and/or not connected.

Instead of going into details of the proof (by structural induction) of RPT tester generation correctness and optimality we provide ad-hoc verification procedure in terms of model checking queries and model construction constraints.

Direct way of verifying the coverage correctness of the tester is to run a model checking procedure with query:

$$A \diamond \forall (i : int[1,n]) \, t[i], \tag{2}$$

where $t[i]$ denotes i-th element of the array of traps. The model the query is running on is synchronous parallel composition of IUT and Tester automata. For instance, the RPT automation for IUT modelled in Figure 3 is depicted in Figure 6.

3.2. Invariance of Tests with Respect to Changing Time Constraints of IUT

In section 2.2 the coverage correctness of RPT tests was discussed without explicit reference to M^{IUT} time constraints. The length-optimality of test sequences can be proven

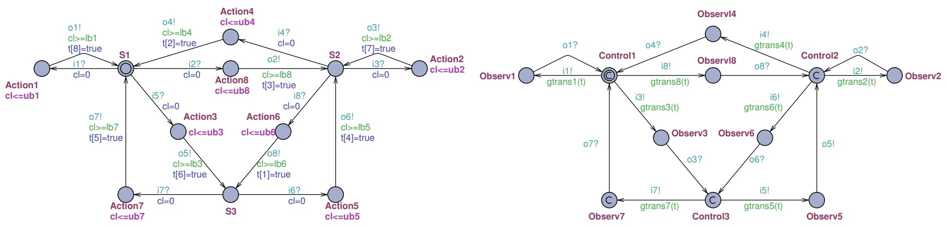

Figure 6. Synchronous parallel composition of IUT and RPT models

in Uppaal when for each $Action_i$ both the duration lower and upper bounds lb_i and ub_i equal to one, i.e., $lb_i = ub_i = 1$ for all $i \in 1,...,|Action|$. Then the length of the test sequence and its duration in time are numerically equal. For instance, having some integer valued (time horizon) parameter TH as an upper bound to the test sequence length the following model checking query proves the coverage of n traps with a test sequence of length at most TH stimuli and responses:

$$A \diamond \forall (i : int[1,n])\, t[i] \wedge TimePass \leq TH \tag{3}$$

where $TimePass$ is Uppaal clock that represents global time of the model.

Generalizing this result for IUT models with arbitrary time constraints we assume that all edges of M^{IUT} are attributed with time constraints as described in Section 2.1. Since not all the transitions of model M^{IUT} need to be labelled with traps (and thus covered by test) we apply compaction procedure to M^{IUT} to abstract away from the excess of information and derive precise estimates of test duration lower and upper bounds. With compaction we aggregate consecutive trapless transitions with one trap-labelled transition the trapless ones are neighbours to. Now, the aggregate Action becomes like atomic Action of Figure 2 that copies the trap of the only trap labelled transition included in the aggregate. The first transition of the aggregate contributes its input event and the last transition its output event. The other I/O events of the aggregate will be hidden because all internal transitions and locations are substituted with one aggregate location we call composite Action. Further, we compute the lower and upper bounds for the composite action. The lower bound is the sum of lower bounds of the shortest path in the aggregate and the upper bound is the sum of upper bounds of the longest path of the aggregate plus the longest upper bound (the later is needed to compute the test termination condition). After compaction of deterministic and timed IUT model it can be proved that the duration TH of a coverage correct tests have length that satisfies following condition:

$$\sum_i lb_i \leq TH \leq \sum_i ub_i + \max_i(ub_i), \tag{4}$$

where index i ranges from 1 to n (n - number of traps in M^{IUT}).

In case of non-deterministic IUT models, for showing length- and time-optimality of generated tests the bounded fairness of M^{IUT} needs to be assumed. We say that a model M is $k-fair$ iff the difference in the number of executions of alternative transitions of non-deterministic choices never exceeds the bound k. This assumption excludes unbounded "starvation" and "conspiracy" behaviour in non-deterministic models. During the test run our test execution environment dTron [10] is monitoring k-fairness and reporting error message "violation of IUT k-fairness assumption" when this constraint is broken. Due to k-fairness monitoring by dTron the safe estimate of the test length upper bound in case of non-deterministic models can be found for the worst case by multiplying the deterministic upper bound by factor k. The lower bound still remains $\sum_i lb_i$.

Proposition 2. Assuming a trap labelled UPTA model M^{IUT} is well-formed in the sense of Definition 2 and compactified, the RPT that is generated based on M^{IUT} remains invariant with respect to variations of the time constraints specified in M^{IUT}.

The practical implication of Proposition 2 is that a RPT once generated for a timed trap labeled UPTA model M^{IUT}, one can use it for any syntactically and semantically feasible modification of M^{IUT} where only timing parameters and initial values of traps have been changed. Invariance does not extend to structural changes of M^{IUT}.

Due to the limited space we sketch the proof in two steps by showing that (i) the control decisions of M^{RPT} do not depend on the timing of M^{IUT} and (ii) the M^{RPT} behaviour does not influence the timing on controllable transitions of M^{IUT}.

(i) The behaviour of M^{RPT} depends on the gain guards of its controllable edges and responses (output events) of M^{IUT}, not on the time instances when these responses are generated. Same applies to the gain guards. They are boolean functions defined on the structure of M^{IUT} and the valuation vector of traps. Thus the timing constraints specified in M^{IUT} do not influence the behaviour of M^{RPT}.

(ii) In the synchronous parallel composition $M^{IUT}||_{sync}\ M^{RPT}$ the actions of M^{IUT} and M^{RPT} take the effect over progress of time alternatively. Though the communication of input and output events is synchronous, it is due to the semantics of UPTA, that execution of transitions is instantaneous, and does not pose any constraint on the delay between earlier or later event. Since the planning time of M^{RPT} is assumed to be negligible comparing to the response time of M^{IUT} we model the control locations in M^{RPT} always as committed locations (denoted by "c" in Figure 6) and there is no additional waiting in obsevation locations of M^{RPT} either. Thus, M^{RPT} does not set any restriction to the time invariants $inv(Action_i)$and transition guards $grd(Action_i \rightarrow PostLocation_i)$ of M^{IUT} actions.

4. Test Execution Environment dTron

Uppaal TRON is a testing tool, based on Uppaal [3] engine, suited for black-box conformance testing of timed systems [11]. dTron [12] extends this enabling distributed execution. It incorporates Network Time Protocol (NTP) based real-time clock corrections to give a global timestamp (t_1) to events at IUT adapter(s). These events are then globally serialized and published for other subscribers with a Spread toolkit [13]. Subscribers can be other SUT adapters, as well as dTron instances. NTP based global time aware subscribers also timestamp the event received message (t_2) to compute and possibly compensate for the overhead time it takes for messaging overhead $\Delta = t_2 - t_1$.

Δ is essential in real-timed executions to compensate for messaging delays that may lead to false-negative non-conformance results for the test-runs. Messaging overhead caused by elongated event timings may also result in messages published in on order, but revived by subscribers in another. Δ can then also be used to re-order the messages in a given buffered time-window t_Δ. Due to the online monitoring capability dTron supports the functionality of evaluating upper and lower bounds of message propagation delays by allowing the inspection of message timings. While having such a realistic network latency monitoring capability in dTron our test correctness verification workflow takes into account theses delays. For verfication of the deployed test configuration we make corresponding time parameter adjustments in the IUT model. By Proposition 2 the RPT tester generated is invariant to time parameter variations. Thus final verification against the query 3 is proving that the test is feasible as well in the presence of realistic configuration constraints of the testing framework dTron.

5. Web Testing Case-study

We describe street light control system (SLCS) to show the applicability of the proposed testing workflow. The SLCS has a central server and multiple controllers each controlling one or more streetlight. The controllers have programmable high-power relays (contactors) to manipulate the actual lights, but also have various sensor and communication extensions to provide supplementary capabilities like dimming and following more complex lighting programs.

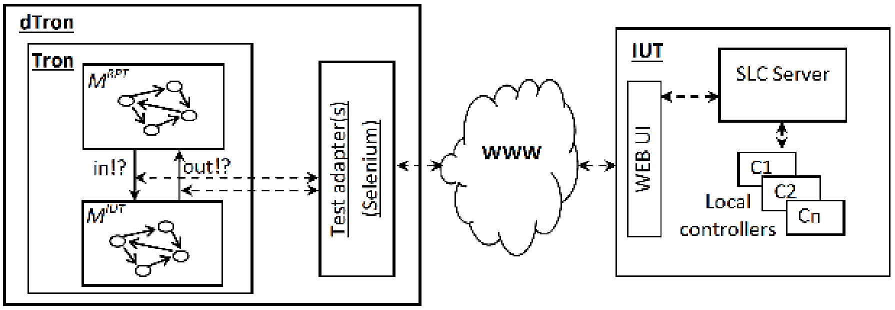

Figure 7. Street light control system test architecture

Light-controllers have minimal memory and do not persistently store their state in the memory. They poll the central server to retrieve their designated state information. This state information is stored in the array of bits, each bit denoting a specific parameter value for the controller. Controller polls the server and the server responds whether it has new state info for the controller. If this is the case, the information is provided with the response. The server holds the state information for each controller. This information can be manipulated by users via an Internet web user interface (UI). Figure 7 shows an abstract view of test architecture. The test purpose is to test if when a user has logged in and tries to turn on a light using the UI, the light will eventually get lit and that is reported back with message lights on.

Figure 8 shows an extract of IUT model M^{IUT} and generated tester M^{RPT}. The test adapters shown in Figure 7 interface symbolic interactions specified by channels in the model with real interface of IUT. These channels are distinguished by name convention. We use names *in* and *out* in the model and they are intercepted by dTron and executed by *adapters*. Adapters translate synchronizations in the model in to actions against the actual system and feed information back to the model.

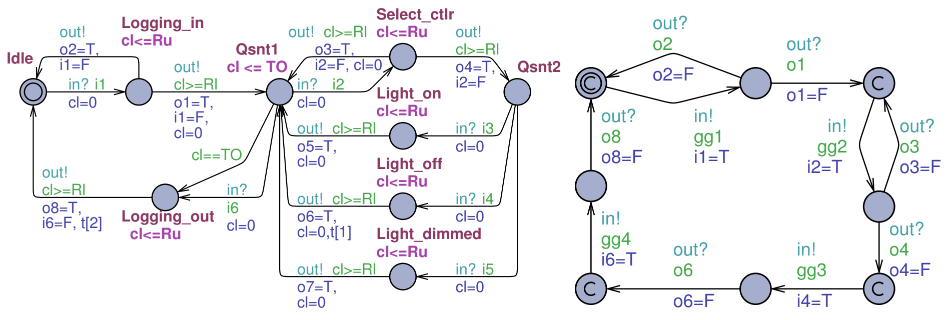

Figure 8. IUT and RPT models

Table 1. Tester input and output variables.

Input		**Output**	
Variable	**Meaning**	**Variable**	**Meaning**
i1	login	o1	login sucessful
i2	select controller (for setting)	o2	login failed
i3	set light on	o3	empty selection of controllers
i4	set light off	o4	mode setting menu for chosen controllers
i5	dimming the light	o5	status report “light on”
i6	logout	o6	status report “light of”
		o7	status report “light dimmed”
		o8	log out completed

Table 2. Pre-execution correctness checks of tests.

Correctness condition	**Verification method**
Output observability of M^{IUT}	Static analysis of test stimulus - response pairs
Connected control structure of M^{IUT}	Generating canonical tester and running query 1
Input enabledness of M^{IUT}	Transformation 1 (see Section 2.2)
Strong responsivness of M^{IUT}	Algorithm 1 (see Section 2.2)
Coverage correctness of M^{RPT}	Model checking query 2
Time-bound checks of tests	Compaction procedure (Section 3.2), calculate 4

The tester is controlling that the test run will cover traps $t[1]$ and $t[2]$. The inputs and outputs of M^{IUT} are explained in the table 1.

The timing constraints of IUT are specified in M^{IUT} as follows:

- TO denotes the time-out to log off after being logged in if there is no activity over UI during TO time units
- All actions controllable and observable over UI have pre-specified duration interval $[Rl, Ru]$. If the responses to IUT inputs do not conform with given interval the timing conformance test fail is reported. Implicitly $[Rl, Ru]$ includes also parameter Δ. The estimate $\widehat{\Delta}$ of Δ is generated by dTron as the result of monitoring the traffic logs at the planned test interface

Before running the executable test dTron performs a sequence of test model verifications. Table 2 illustrates the verification tasks available with current version of dTron.

6. Conclusion

We have proposed a MBT testing workflow that incorporates steps of IUT modelling, test specification, generation, and execution that are alternating with their correctness verification steps. The online testing approach of timed systems proposed relies on Reactive Planning Tester (RPT) synthesis algorithm and distributed test execution environment dTron. As shown in the paper the behaviour of generated RPT tester model does not set extra timing constraints to controllable input/output of IUT and the on-line decisions of the tester do not depend on the timing of IUT. dTron provides support to estimate time delays in real test configuration and allows to take them into account while verifying the test correctness properties with real environment delay constraints. This is a first practical step towards provably correct automated test generation for Δ-testing outlined as a new MBT challenge in [4].

Acknowledgements

This research is partially supported by ELIKO and the European Union through the European Regional Development Fund and by the Tiger University Program of the Information Technology Foundation for Education.

References

[1] Tretmans, Jan. Test Generation with Inputs, Outputs and Repetitive Quiescence In: Software - Concepts and Tools, 1996, 17 (3), 103 -120.

[2] Roberto Segala. Quiescence, Fairness, Testing, and the Notion of Implementation. In: Inf. Comput., 1997, 138 (2), 194-210.

[3] Behrmann, G., David, A., Larsen, K. A tutorial on uppaal. In: Bernardo, M., Corradini, F. (ed.) *Formal Methods for the Design of Real-Time Systems*. Springer, Berlin Heidelberg, 2004. 200 – 236.

[4] Alexandre David, Kim G. Larsen, Marius Mikucionis, Omer L. Nguena Timo, Antoine Rollet. Remote Testing of Timed Specifications. Springer, 2013, 65-81. (Lecture Note in Computer Science, 8254).

[5] Vain, J., Raiend, K., Kull, A., and Ernits, J. Synthesis of test purpose directed reactive planning tester for nondeterministic systems. In: 22nd IEEE/ACM Int. Conf. on Automated Software Engineering. ACM Press, 2007, 363 – 372.

[6] Luo, G., von Bochmann, G., & Petrenko, A. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. IEEE Transactions in Software Engineering, 1994, 20 (2), 149 – 162.

[7] Hamon, G., de Moura, L., & Rushby, J. Generating efficient test sets with a model checker. In: SEFM 2004: Proceedings of the Software Engineering and Formal Methods, Second International Conference. IEEE Computer Society, 2004, 261 – 270.

[8] Kääramees, M. A Symbolic Approach to Model-based Online Testing [dissertation]. Tallinn: TUT Press, 2012.

[9] Brinksma, Ed., Alderen, R., Lngerak, R., Lagemaat, J.d.v., Tretmans, J., A Formal approach to conformance testing. 2nd Workshop on Protocol Test Systems. Berlin, October 1989.

[10] A.Anier, J.Vain. Model based continual planning and control for assistive robots. HealthInf 2012. Vilamoura, Portugal. 1-4 Feb, 2012.

[11] UPPAAL TRON. [WWW] http://people.cs.aau.dk/˜marius/tron/ (accessed 20.04.2014)

[12] DTRON home page. [WWW] http://dijkstra.cs.ttu.ee/˜aivo/dtron/ (accessed 20.04.2014)

[13] The spread toolkit. [WWW] http://spread.org/ (accessed 20.04.2014)