Reconstructing Timed Symbolic Traces from *rtioco*-based Timed Test Sequences using Backward-Induction.*

Junaid Iqbal
Åbo Akademi University
Turku, Finland
junaid.iqbal@abo.fi

Dragos Truscan
Åbo Akademi University
Turku, Finland
dragos.truscan@abo.fi

Jüri Vain
Tallinn University of Technology
Tallinn, Estonia
juri.vain@ttu.ee

Ivan Porres
Åbo Akademi University
Turku, Finland
ivan.porres@abo.fi

ABSTRACT

As of today, model-based testing is considered as a leading-edge technology in the IT industry. In model-based testing, an implementation under test is tested for compliance with a model that describes the required behaviour of the implementation. Uppaal Tron is a popular tool for online model-based conformance testing of real-time systems; it uses the Uppaal verification engine to generate and convert on-the-fly timed symbolic traces into concrete test sequences. Among the advantages of online testing is the reduction of the symbolic state space needed for computing traces, better addressing non-determinism, as well as the possibility to execute longer-lasting test runs. However, analysing and debugging long test runs can be tedious and time-consuming especially when analysing root causes of failed tests. In game theory, backward-induction is a process to reason backwards in time, from the end of a problem or situation, in order to determine a sequence of optimal actions. In this paper, we propose an approach to reconstruct symbolic traces from test sequences generated by Uppaal Tron using backward-induction. The resulting symbolic traces can be imported in the Uppaal tool and visualised in the Uppaal simulator. The evaluation of the implementation of the approach shows that it has the potential to satisfy the needs of industrial level testing.

CCS CONCEPTS

• **Theory of computation → Timed and hybrid models**; **Formal languages and automata theory**; **Solution concepts in game theory**; • **Software and its engineering → Software testing and debugging**;

*This work has been *__partially__* supported by the ECSEL JU MegaM@Rt2 project under grant agreement No 737494.

ECBS '17, August 31-September 1, 2017, Larnaca, Cyprus

ACM ISBN 978-1-4503-4843-0/17/08…$15.00
https://doi.org/10.1145/3123779.3123813

KEYWORDS

timed automata, *rtioco*-based timed test sequences, timed symbolic traces, diagnostic traces, Uppaal model-checker, Uppaal Tron, game theory, backward-induction.

ACM Reference format:
Junaid Iqbal, Dragos Truscan, Jüri Vain, and Ivan Porres. 2017. Reconstructing Timed Symbolic Traces from *rtioco*-based Timed Test Sequences using Backward-Induction.. In *Proceedings of ECBS '17, Larnaca, Cyprus, August 31-September 1, 2017,* 10 pages.
https://doi.org/10.1145/3123779.3123813

1 INTRODUCTION

Model-based testing (MBT) [28] is increasing in popularity in both academia and industry due to its ability to automatically generate tests from abstract models depicting the expected behaviour of the system under test (SUT). Approaches for test modelling and test generation from different specification languages have been proposed in the past, both in the un-timed [3, 14, 19] and timed [8, 10, 12, 17] domain. In the latter category, testing using *timed automata* (TA) gained interest especially due to the maturity of model-checkers like Uppaal and adjacent work on test case generation from TA, e.g., using Uppaal Tron [16].

The Uppaal model-checking tool suite is an integrated environment for modelling, validation, and verification of real-time systems [2]. It uses a network of extended TA to specify the behaviour of the system and it can be used to verify whether given properties of the system, specified in temporal logic, are violated or not. In the former case, a counter example (aka *diagnostic trace)* is presented and can be visualized in the Uppaal simulator.

Hessel *et al.* proposed algorithms for online test generation from Uppaal timed automata (UTA) based on the *relativized timed input/output conformance relation* (*rtioco*) [16]. The algorithm relies on the capability of Uppaal to create traces via symbolic reachability analysis, which by interacting with the implementation under test (IUT) are transformed into *observed test runs* or *test sequences*.

When executing test sequences, the symbolic states to be visited are calculated and a decision on the next input is taken based on the received output of the IUT and choosing randomly one of the enabled model transitions. A test session stops when the model reaches a final state, the test duration expires, or when a violation of the conformance relation is encountered.

While online testing brings benefits in terms of reducing the state space stored in the memory, addressing non-determinism better, and executing long test runs, it suffers from difficulties in analysing and diagnosing the test result in case of long lasting test runs [16]. In many situations, especially when a failed or inconclusive test result is encountered, one would like to understand and visualise which traces have been explored in the model during a test session, which symbolic states have been visited, and which were the values of both clock and integer variables in each symbolic state.

In order to locate the fault at specification level, one potential solution is to include the functionally in Uppaal Tron to record and store the symbolic test traces corresponding to the generated test sequence. Since Uppaal Tron is not open-source, the updates can only be done by the original authors. However, the recording and storing of symbolic test traces during the test run might interfere with the real-time constraints of test generation due to i/o latency and, consequently, may lead to a incorrect test verdict. Furthermore, the new functionality will not be useful for previously created test sequences, generated with current and past versions of Uppaal Tron and stored for later inspection. These test sequences are not reproducible due to the random choices of inputs and time delays and because of different optimization techniques used for reducing the symbolic state space used by the test generation algorithm of Uppaal Tron [20]. Reconstructing a posteriori the symbolic trace by forward traversing of the timed test sequence might not be a feasible choice either since one has to generate all reachable symbolic states and to identify the choices which corresponded to the test sequence, which will require intense computation effort.

Therefore, in this work we propose a generic approach which uses *backward-induction* to reconstruct the symbolic trace corresponding to a conformance testing session of Uppaal Tron. While the timed test sequence will include events and delays observable on the test interface, the reconstructed symbolic trace will include both observable and non-observable events and delays, corresponding to model-level transitions.

Symbolic trace reconstruction allows us to take advantage of Uppaal's capabilities for simulation and visualisation in order to improve the debugging process and to reduce the cognitive effort needed to identify the underlying causes of inconclusiveness or failure. For clarity, we would like to point out that: a) we do not recreate the entire symbolic state space of the model, but only the symbolic trace (sequence of states and state transitions) leading to the verdict state and b) our approach can be applied to any sequence generated by Uppaal Tron regardless of its test verdict or number of test events.

This paper is divided into the following sections. Section 2 revisits its background information and concepts related to UTA and *rtioco*. We briefly introduce backward-induction in Section 2.4. Section 3 details our proposed approach, followed by a brief overview of tool support. We evaluate the scalability of approach with a smart lamp light controller and a temperature control system examples in Section 4. We give an overview of related literature in Section 5 and conclude in Section 6.

2 BACKGROUND

In this section, we briefly introduce the terminology and notations used throughout this paper.

2.1 Timed input-output transition systems

Definition 2.1. A *timed labelled transition system (TLTS)* [23] is a 4-tuple $\langle S, s_0, Act_{\tau,\epsilon}, \longrightarrow \rangle$, where S is a non-empty set of states; $s_0 \in S$ is the initial state; $Act_{\tau,\epsilon} \stackrel{def}{=} Act \cup \{\tau\} \cup \mathcal{D}$ are the actions, $Act_{\tau,\epsilon}$ including the *internal action* τ and *time-passage actions (delay)* ϵ ; where $\mathcal{D}$ is $\tau(d) | d \in \mathbb{R}^+$. A transition $\longrightarrow \subseteq (S \times Act_{\tau\epsilon} \times S)$ is a relation with the following consistency constraints:

- **Time determinism.** whenever $s \xrightarrow{\epsilon(d)} s'$ and $s \xrightarrow{\epsilon(d)} s''$ then $s' = s''$
- **Time additivity.** $\forall s, s'' \in S \wedge \forall d_1, d_2 \geq 0 : (\exists s' \in S : s \xrightarrow{\epsilon(d_1)} s' \xrightarrow{\epsilon(d_2)} s'')$ *iff* $s \xrightarrow{\epsilon(d_1+d_2)} s''$
- **Null delay.** $\forall s, s' \in S : s \xrightarrow{\epsilon(0)} s'$ *iff* $s = s'$

The labels in Act_ϵ ($Act_\epsilon \stackrel{def}{=} Act \cup \mathcal{D}$) represent the observable actions of a system, i.e. labelled actions and passage of time; the special label τ represents an unobservable internal action. A transition (s, μ, s') is denoted as $s \xrightarrow{\mu} s'$. A computation is a finite or infinite sequence of transitions:

$$s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_s} s_2 \xrightarrow{\mu_3} \cdots \xrightarrow{\mu_{n-1}} s_{n-1} \xrightarrow{\mu_n} s_n (\longrightarrow \cdots)$$

A timed trace captures the observable aspects of a computation; it is a sequence of observable actions. The set of all *finite* sequences of actions over Act including empty sequence ϵ is denoted by $Act_\epsilon^\star$.

We define *timed input-output transition system (TIOTS)* is a timed labelled transition system $\langle S, s_0, Act_{\tau\epsilon} \longrightarrow \rangle$ with Act partitioned into input actions, Act_I, and output actions, Act_U, such that $Act_I \cup Act_U = Act, Act_I \cap Act_U = \emptyset$.

Briones *et. al.* [9] suggest the convention that input actions are identified by names followed by symbol (?), and output actions by names followed by symbol (!). Thus, a timed trace σ is a sequence of *i/o* actions and delays, *e.g.* $\sigma = a? \cdot \epsilon(d1) \cdot \epsilon(d2) \cdot b!$. Obviously, the consecutive delays in a trace as in $\sigma = a? \cdot \epsilon(d1 + d2) \cdot b!$ can be aggregated and alternatively be written as sequences of actions with relative time stamps, viz. $\sigma = a?(0) \cdot b!(d1 + d2)$.

Definition 2.2. **(Formal definition of TIOTS)** For a set of actions Act, partitioned into two disjoint sets of input actions A_I and output actions A_U, by adding a unobservable action $\tau \notin Act$, we get an extended action set denoted $Act_\tau = Act \cup \{\tau\}$.

Thus, a TIOTS $\mathcal{S}$ is a tuple $(S, s_o, A_I, A_U, \rightarrow)$, where

- S is a set of states, and $s_0 \in S$;
- A_I and A_U are the set of input and output actions respectively;
- and $\rightarrow \subseteq S \times (Act_\tau \cup \mathbb{R}_{\geq 0}) \times S$ is a transition relation satisfying the usual constraints of time determinism (see definition 2.1) (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 $s \xrightarrow{d_1+d_2} s''$), and null-delay (for all states $s \xrightarrow{0} s$). $d, d_1, d_2 \in \mathbb{R}_{\geq 0}$, and $\mathbb{R}_{\geq 0}$ denotes the set of non-negative real numbers.

Several TIOTS can be composed in parallel to model the behaviour of SUT and its environment under a closed system which

interact via observable behaviour. We define TIOTS parallel composition as follows:

Definition 2.3. **(TIOTS composition)** Let $\mathcal{S} = (S, s_0, A_I, A_U, \rightarrow)$ and $\mathcal{E} = (E, e_o, A_U, A_I, \rightarrow)$ be TIOTSs. Here E is the set of environment states and e_0 being the initial state such that $e_0 \in E$. Both $\mathcal{E}$ and $\mathcal{S}$ share the identical set of input (output) actions. The parallel composition of $\mathcal{S}$ and $\mathcal{E}$ forms a closed system $\mathcal{S} \parallel \mathcal{E}$ whose observable behaviour is defined by the TIOTS $(S \times E, (s_0, e_0), A_I, A_U, \rightarrow)$, where $\rightarrow$ is defined as:

$$\frac{s \xrightarrow{a} s' \quad e \xrightarrow{a} e'}{(s,e) \xrightarrow{a} (s',e')} \qquad \frac{s \xrightarrow{\tau} s'}{(s,e) \xrightarrow{\tau} (s',e)} \qquad \frac{e \xrightarrow{\tau} e'}{(s,e) \xrightarrow{\tau} (s,e')} \qquad \frac{s \xrightarrow{d} s' \quad e \xrightarrow{d} e'}{(s,e) \xrightarrow{d} (s',e')}$$

and $a \in Act,\ d \in \mathbb{R}_{\geq 0}$ *and* $\tau \notin Act$.

2.2 UPPAAL timed automata

A timed automaton is essentially a finite automaton (that is a graph containing a finite set of nodes called locations and a finite set of labelled edges) extended with real-valued variables [17] [6]. Such an automaton may be considered as an abstract model of a timed system. The variables model the logical clocks in the system, that are initialized with zero when the system is started, and then incremented synchronously with the same rate. The behaviour of the automaton is restricted by using clock constraints *i.e.* guards on edges. A transition represented by an edge can be taken when the clock values satisfy the guard which labels the edge. The clocks may be reset to zero when a transition is taken.

Assume a finite set of real-valued variables C, ranged over by x, y, denotes the *clocks* and a finite set of alphabet A ranged over by a, b denotes *actions*. A **clock constraint** is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$ for $x, y \in C, \sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. The set of *clock constraints* $\mathcal{B}(C)$ is ranged over by g. A **zone** for a set of clocks C is a conjunction of atomic constraints consisting of a pair of clock variables where difference of clock variables is maintained using the tightest constraint between them [4].

Definition 2.4. A **timed automaton** (TA) T over action A is a tuple $\langle S, l_0, E, I\rangle$, where [23]:

- S is a non-empty finite set of locations;
- $l_0 \in S$ is the initial location;
- $E \subseteq S \times \mathcal{B}(C) \times A \times \mathbb{P}^C \times S$ is a set of edges where $\mathbb{P}^C$ represents the power set of C (clocks).
- $I : S \mapsto \mathcal{B}(C)$ is the location invariant mapping, that gives an invariant $g \in \mathcal{B}(C)$ for each location $l \in S$.

To model concurrent systems, TA can be extended with parallel composition. A **network of TA** $N_{TA} = (T_1 || \cdots || T_n)$ is a collection of concurrent TA T_i composed by parallel composition. The *state of the network* is modelled by a configuration $\langle \bar{\ell}, \bar{c}\rangle$. The first component is a location vector $\bar{\ell} = \langle \ell_1, \cdots, \ell_n\rangle$ where ℓ_i is the set of locations of automaton T_i. The second component $\bar{c} \in \mathbb{R}_+$ is the valuation of all clock variables. The initial state of the network is $\langle \bar{\ell}_0, \bar{0}\rangle$ where all automaton in N_{TA} are at initial locations and the valuation of clock variable is zero.

The **semantics of a timed automaton** T is defined by associating a timed labelled transition system S_T with T. A symbolic state s of a timed automaton is a pair $\langle \ell, c\rangle$, where $\ell \in L$ is a location and c is the valuation of all clocks in $\mathcal{B}(C)$. The valuation c must always satisfy the invariant constraints in the current location of the automaton $\ell : c \models I(\ell)$. There are three types of **transitions** in a TA network: A **transition** for TA network N_{TA} is defined by:

- **Action:** if $\ell_i \xrightarrow{g,a,r} \ell_i'$ is an action transition in the i-th automaton with a guard over clock constraint $g(\bar{c}), \bar{c}' \models I(\bar{\ell}')$ and $a \in A$ an action, and an action transition $\langle \bar{\ell}, \bar{c}\rangle \xrightarrow{a} \langle \bar{\ell}', \bar{c}'\rangle$.
- **Synchronization:** if $\ell_i \xrightarrow{g_1,a,r_1} \ell_i'$ and $\ell_j \xrightarrow{g_2,\bar{a},r_2} \ell_j'$ is synchronized transition in i-th and j-th $(i \neq j)$ automata with $\bar{c} \models (g_1 \wedge g_2)$ and $\bar{c}' \models I(\bar{\ell}')$ then $\langle \bar{\ell}, \bar{c}\rangle \xrightarrow{\tau} \langle \bar{\ell}', \bar{c}'\rangle$ is an internal action transition in N_{TA}, where $a, \tau \in A, \bar{\ell}' = \bar{\ell}[\ell_i'/\ell_i, \ell_j'/\ell_j]$ and $\bar{c}' = (r_1 \cup r_2)(\bar{c})$. The $\bar{\ell}[\ell_i'/\ell_i]$ represents that the i-th element of the $\bar{\ell}$ has been replaced by ℓ_i' [5].
- **Delay:** if $\delta \in \mathbb{R}_+$ is a delay with condition $\forall d < \delta : (\bar{c} + d) \models I(\bar{\ell})$, then $\langle \bar{\ell}, \bar{c}\rangle \xrightarrow{\delta} \langle \bar{\ell}, \bar{c} + \delta\rangle$ is a δ-*delay* transition in N_{TA}.

Then, Uppaal timed automata (UTA) [2] are an extension of TA with bounded integer variables and simple data types (aka, TA with data variables) as defined in [5]:

Definition 2.5. A **timed automaton with data variables** over actions A, clock variables C and data variables V is a tuple (L, ℓ_0, E) where

- L is a finite set of nodes (control-nodes),
- ℓ_0 is the initial node,
- $E \subseteq L \times \mathcal{G}(C, V) \times A \times \mathbb{P}^C \times L$ corresponds to the set of edges,
  - where $\mathcal{G}(C, V)$ is set of guards ranged over by g.
  - g is a constraint in the form: $c \sim n$ or $v \sim n$ for $c \in C$, $v \in V, \sim \in \{\leq, \geq, =\}$ and n being an natural number.
  - the guards $\mathcal{G}(C, V)$ can be divided into two parts: a conjunction of constraints over clocks variables in the form $c \sim n$ and conjunction of constraints over data variables in the form $v \sim n$.

2.3 Conformance testing with UTA

During a test session, Uppaal Tron uses the Uppaal verification engine to generate symbolic timed traces in the UTA model. For each symbolic state, the possible symbolic states to visit are calculated and the next state is chosen by randomly choosing one of the enabled transitions. A test session ends when the model reaches a final state, the test duration expires, or a violation of conformance between implementation and specification is encountered.

A symbolic timed trace TTr_S of an UTA model is a (possibly infinite) sequence of symbolic states, each state being defined as a tuple $(\bar{\ell}, D, \bar{v})$, where $\bar{\ell}$ is a location vector, D is the clock constraints (zone) [4] and $\bar{v}$ a vector of variable values [15]. The transition from one symbolic state to another can be either an action (a_i) or a delay (δ_i).

$$(\bar{\ell}_i, D_i, \bar{v}_i) \xrightarrow{a_i/\delta_i} (\bar{\ell}_j, D_j, \bar{v}_j) \tag{1}$$

In UTA, an action may be composed of an event e and a manipulation of the global data space V. As a consequence, when the system state changes, we can observe either an event e, a modified data space V, or both.

An example is shown in Figure 1. The symbolic state a has been visited after evaluating the guard (g), performing the reset operation (r), and observing an action (A). Similarly, states $b, c, \cdots, d, e$ are visited after evaluating their respective guards g, reset r and action event A. The state f represents an error state where Uppaal Tron assigned a verdict *failed* (f) to the test run.

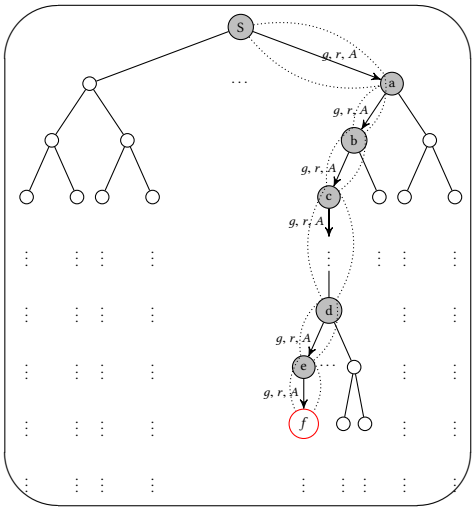


Figure 1: Symbolic state space and visited states

The decision on which state transitions are enabled in a given state is done based on the interaction between Uppaal Tron and the IUT [16] by evaluating received output, available inputs or delays.

In order to identify the observable behaviour between the tester and the IUT, Tron partitions the UTA model into two parallel partitions $\mathcal{S}$ and $\mathcal{E}$, which model respectively the IUT and its environment. The interaction between $\mathcal{S}$ and $\mathcal{E}$ is done via observable actions, further divided into input (A_I) and output (A_U) actions. The former are used as stimuli to the IUT during testing whereas latter are used for conformance. Additionally, $\mathcal{S}$ and $\mathcal{E}$ have internal actions confined to each partition, evolving the partition to next state where the next observable action can be taken.

During testing, the observable actions A_I and A_U are triggered based on a testing event e, following an observable delay $\Delta \in \mathbb{R}_{\geq 0}$ which abstracts the internal events (described later in this section). A vector of externally visible variables ($\bar{v}$) which contains the value of data variable at the time of event is also observable. The events and variables are partitioned into three disjoint sets of input events/variables Ev_{in}/V_{in}, output events/variables Ev_{out}/V_{out}, and internal events/variables Ev_{int}/V_{int} [15].

Thus, after partitioning the model into environment and SUT partitions, a symbolic trace can be rewritten as a *timed input output trace*. The latter is a (possibly infinite) sequence of observations starting from a given state, where each observation is a tuple $(e, D, \bar{v})$ consisting of an event $e \in Ev_{(in/out)}$, a clock zone D in which event occurs, and a vector $\bar{v} \in V_{(in/out)}$ containing the values of data variables that are externally visible as inputs/outputs at the time of event e.

$$ttr_{i/o} = (e_0, D_0, \bar{v}_0), (e_1, D_1, \bar{v}_1), \ldots (e_i, D_i, \bar{v}_i), \ldots \quad (2)$$

Uppaal Tron is only using the externally visible (observable) events to interact with the IUT, while abstracting the internal actions (τ) and the internal delays (d) to observable delays (Δ).

Thus, the result of a test session will be a finite sequence of events T_{seq} of the form:

$$\begin{aligned} T_{seq} =&(e_0, (\tau_0 + d_0), \bar{v}_0), (e_1, (\tau_1 + d_1), \bar{v}_1), \cdots, \\ &(e_n, (\tau_n + d_n), \bar{v}_n), (e_{n+1}, (\tau_{n+1} + d_{n+1}), \bar{v}_{n+1}) \end{aligned} \quad (3)$$

which can be written in terms of observable delays and actions as:

$$\begin{aligned} T_{seq} =&(e_0, \Delta_0, \bar{v}_0), (e_1, \Delta_1, \bar{v}_1), \cdots, \\ &(e_n, \Delta_n, \bar{v}_n), (e_{n+1}, \Delta_{n+1}, \bar{v}_{n+1}), \cdots \end{aligned} \quad (4)$$

This allows one to check the timed conformance of the IUT against the specification via the *rtioco* relation, by allowing the IUT to refine the timing behavior of the specification [16].

Definition 2.6. **Relativized timed input/output conformance (*rtioco*):** An implementation I conforms to its specification S under the environmental constraints $\mathcal{E}$ if for all timed input traces $\sigma \in TTr_i(\mathcal{E})$ the set of timed output traces of I is a refinement of the set of timed output traces of S for the same input trace.

$$\begin{aligned} I \ rtioco \ S \quad \text{iff} \quad &\forall \sigma \in TTr_i(\mathcal{E}) : \\ &TTr_o((I, \mathcal{E}), \sigma) \sqsubseteq TTr_o((S, \mathcal{E}), \sigma) \end{aligned} \quad (5)$$

A conceptual mapping between a symbolic timed trace and test sequence is shown in Figure 2.

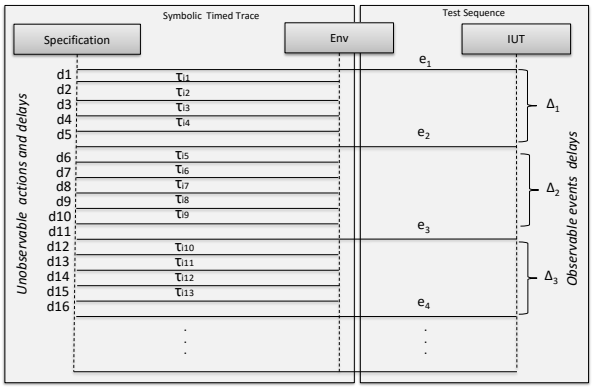


Figure 2: Conceptual mapping between symbolic and observable test sequence

The resulting test sequence is provided by Uppaal Tron as a sequence of test events. For each test event, symbolic state in which the event occurred is specified in terms of clock constraints, variable valuation, list of next available states, and list of input/output actions as shown in Figure 3.

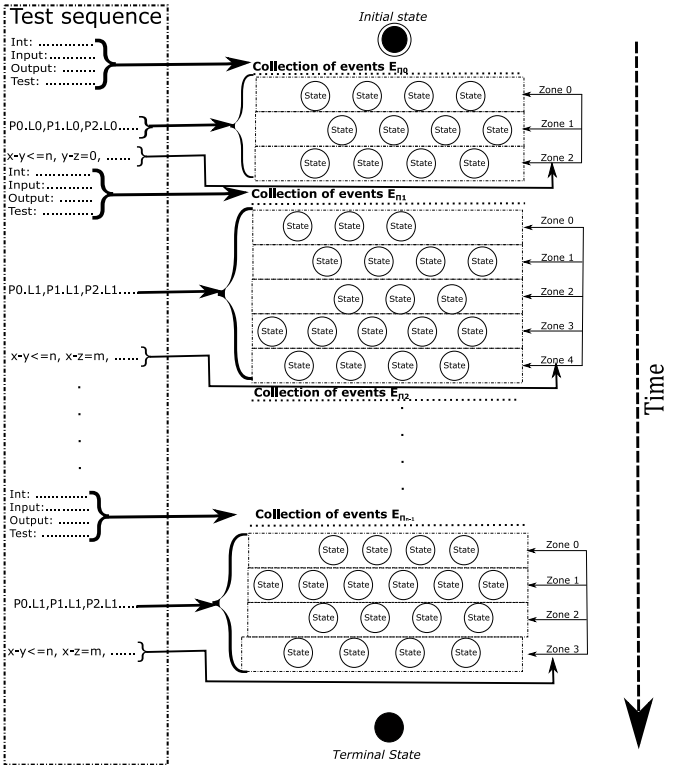


Figure 3: Structure of the test sequence

A new test event occurs at a specified time instant, the clock constraints are updated, a transition to a new symbolic state occurs and the list of the next available states is updated. However, no information is provided regarding which transition is taken to reach next state and under which conditions. Thus, in order to be able to reconstruct a symbolic trace that can be simulated in the Uppaal simulator one needs to infer all the missing information from the last state of the trace to the initial one.

2.4 Backward-induction

The process of inferring a sequence of optimal actions by deducting backwards from the end of a scenario to the initial conditions is known in game theory as *backward-induction*. The process starts in the last step of a two-player game and, by anticipating what the last player did, it determines what moves are likely to lead to it. Backward-induction was first mentioned by game theory inventors John von Neumann and Oskar Morgenstern in 1944 [29].

The scope of game theory is out of the context of this paper. Therefore, we restrict our discussion concerning building a game tree and rationale about the choice function in a stochastic behaviour concerning reconstruction of a timed symbolic trace from a timed test sequence. One may refer to [29] for details.

Walter *et. al.* [7] proposed an algorithm showing that every choice rule is backward-induction rationalizeable. Jiangtao Li *et. al.* [21] extended this work by including random choices. The algorithm constructs a *tree* based on the available finite universal solution set X of *alternatives*, and denotes by $\mathcal{P}(X)$ as the *collection* of all non-empty subsets of X. A *choice function* $f : \mathcal{P}(X) \longrightarrow X$ is a map such that $f(\mathcal{A}) \in \mathcal{A}$ for all $\mathcal{A} \in \mathcal{P}(X)$, selects an alternative from $\mathcal{A}$. A *random choice rule* is a map $\rho : X \times \mathcal{P}(X) \longrightarrow [0, 1]$ such that for all $\mathcal{A} \in \mathcal{P}(X)$. The random choice rule defined as:

$$\rho(x, \mathcal{A}) = \begin{cases} 1 & \text{if } x \in \mathcal{A} \\ 0 & \text{if } x \notin \mathcal{A} \end{cases} \tag{6}$$

where $\mathcal{A}$ is a set of alternatives and x is a random choice. The interpretation of the mapping rule denotes the probability that alternative x is chosen when the possible alternatives are in $\mathcal{A}$. Following definition are taken from [21] and [7].

A preference is the ordering of alternatives based on their relative utility, a process which results in an optimal choice.

Definition 2.7. **Preference ordering**. A *preference ordering* is a reflexive, complete, transitive and antisymmetric binary relation. We denote by $\mathcal{R}_{\mathcal{A}}$ the set of all preference orderings on $\mathcal{A} \in \mathcal{P}(X)$.

By using preference ordering, the order of alternatives is decided. To decide the relation between alternatives we define a *precedence relation* amongst alternatives.

Definition 2.8. **Precedence relation.** Let $\rhd$ be a transitive and asymmetric binary relation on a non-empty and finite set N. We say that $n \in N$ is a *direct predecessor* of $n' \in N$ if $n \rhd n'$, and there is no $n'' \in N$ such that $n \rhd n'' \rhd n'$. Similarly, we say that $n \in N$ is a *direct successor* of $n' \in N$ if $n' \rhd n$ and there is no $n'' \in N$ such that $n' \rhd n'' \rhd n$. The set of direct predecessors of $n \in N$ is

denoted by $P(n)$. The set of direct successors of $n \in N$ is denoted by $S(n)$.

By considering the preference ordering and precedence relation, we construct a *game tree* define as:

Definition 2.9. **Tree.** A tree Γ is given by a quadruple $(r, \mathbb{D}, \mathbb{T}, \rhd)$, where the variables are defined as follows: (i) the notation r is the *root*; (ii) the variable $\mathbb{D}$ is a finite set of decision nodes such that $r \in \mathbb{D}$; (iii) the variable $\mathbb{T}$ is a non-empty and finite set of terminal nodes such that $\mathbb{D} \cap \mathbb{T} = \emptyset$; (iv) the notation $\rhd$ is a transitive and asymmetric precedence relation on the set of all nodes $\mathbb{N} = \mathbb{D} \cap \mathbb{T}$ such that: (a) $P(r) = \emptyset$; and $|S(r)| \geq 1$; (b) for all $n \in \mathbb{D} \setminus r$, $|P(n)| = 1$, and $|S(n)| \geq 1$; (c) for all $n \in \mathbb{T}$, $|P(n)| = 1$, and $S(n) = \emptyset$;.

The nodes in a game tree are connected via edges forming a *path*, defined as follows:

Definition 2.10. **Path.** A *path* in Γ from a decision node $n \in \mathbb{D}$ to a terminal node $n' \in \mathbb{T}$ (of length $K \in \mathbb{N}$) is an ordered $(K+1)$ tuple $(n_0, n_1, ..., n_K) \in N^{|K+1|}$ such that $n_0 = n, \{n_{k-1}\} = P(n_k)$ for all $k \in 1, 2, ..., K$, and $n_K = n'$.

Definition 2.11. A **game** is a triple $G = (\Gamma, \int, \pi)$. where :

- $\Gamma = (r, \mathbb{D}, \mathbb{T}, \rhd)$ is a tree;
- $\int : \mathbb{T} \longrightarrow X$ is an outcome function that maps each terminal node $n \in \mathbb{T}$ to an alternative $\int(n) \in X$,
- π is a probability measure over the space of preference assignment maps, where each preference assignment map $R : \mathbb{D} \longrightarrow \mathcal{R}_X$ specifies for each decision node $n \in \mathbb{D}$ a preference ordering $R(n) \in \mathcal{R}_X$. We denote by $\mathfrak{R}_{\mathcal{D},X}$ the space of all such preference assignment maps, and denote by $\triangle\ \mathfrak{R}_{\mathbb{D},X}$ the set of all probability measures over $\mathfrak{R}_{\mathbb{D},X}$. Formally, $\pi \in \triangle\ \mathfrak{R}_{\mathbb{D},X}$.

To play a legitimate game, the restrictions are defined as follows:

Definition 2.12. **Restriction of a game:** For a game $G = (\Gamma, \int, \pi)$, we define *restriction* of game G on $\mathcal{A} \in \mathcal{P}(X)$ as $G|\mathcal{A} = G_{\mathcal{A}} = (\Gamma_{\mathcal{A}}, \int_{\mathcal{A}}, \pi_{\mathcal{A}})$, where

- $r_{\mathcal{A}} = r$;
- $\mathbb{D}_{\mathcal{A}} = \{n \in \mathbb{D} :$ there exists $n' \in \int^{-1}(\mathcal{A})$ and a path in Γ from n to $n'\}$;
- $\Gamma_{\mathcal{A}} = \int^{-1}(\mathcal{A})$;
- $\rhd_{\mathcal{A}}$ is the restriction of $\rhd$ to $N_{\mathcal{A}} = \mathbb{D}_{\mathcal{A}} \cap \mathbb{T}_{\mathcal{A}}$;
- $\int_{\mathcal{A}}$ is the restriction of $\int$ to $\mathbb{T}_{\mathcal{A}}$;
- $\pi_{\mathcal{A}} \in \triangle\ \mathfrak{R}_{\mathbb{D}_{\mathcal{A}},\mathcal{A}}$ is the induced probability measure from $\pi \in \triangle\ \mathfrak{R}_{\mathbb{D},X}$. For any $R_{\mathcal{A}} \in \mathfrak{R}_{\mathbb{D}_{\mathcal{A}},\mathcal{A}}$, $\pi_{\mathcal{A}}(\{R_{\mathcal{A}}\}) = \pi(\{R \in \mathfrak{R}_{\mathbb{D},X} : R_{\mathcal{A}}$ is the restriction of R to $\mathbb{D}_{\mathcal{A}}$ and $\mathcal{A}\})$.

  For any $R \in \mathfrak{R}_{\mathbb{D},X}$, we denote by $R_{\mathcal{A}}$ the restriction of R to $\mathbb{D}_{\mathcal{A}}$ and $\mathcal{A}$.

Definition 2.13. **Backward-induction rationalizable.** A choice function f is *backward-induction rationalizable* if there is a game $G = (\Gamma, \int, R)$ such that: $e(\Gamma_A, \int_{\mathcal{A}}, R_{\mathcal{A}}) = f(\mathcal{A})$ for any $\mathcal{A} \in \mathcal{P}(X)$. We say that G is a backward-induction rationalization of f or that G *backward-induction rationalizes* f. A random choice rule ρ is *backward-induction rationalizable* if there is a game $G = (\Gamma, \int, \pi)$ such that $\pi(\{R \in \mathfrak{R}_{D,X} : e(\Gamma_{\mathcal{A}}, \int_{\mathcal{A}}, R_{\mathcal{A}}) = x\}) = \rho(x, \mathcal{A})$ for any $x \in \mathcal{A} \in \mathcal{P}(X)$. We say that G is a *backwards-induction rationalization* of ρ or that G *backward-induction rationalizes* ρ.

A game tree $\Gamma = (r, \mathbb{D}, \mathbb{T}, \rhd)$ consists of an initial node r , a set of decision nodes $\mathbb{D}$; a set of terminal nodes $\mathbb{T}$; a set of restrictions $\rhd$. An outcome function $\int$ such that $\int(n) = \int^{\mathcal{A},x}(n)$ gives a choice x from set of alternatives $\mathcal{A}$ of n (see [7][21] for detailed proofs).

3 RECONSTRUCTING SYMBOLIC TIMED TRACES FROM TEST SEQUENCES

As stated in the introduction, online testing brings benefits with respect to reducing the size of the state space, addressing non-determinism better, and executing long test runs. However, it suffers from difficulties in analysing and diagnosing the test result in the case of long-lasting test runs.

During the online testing session, the internal state of the IUT is not visible. Notably, Uppaal Tron presents an obscure execution test sequence which consists of a list of reachable states along with events. The ambiguity might lead a human tester to spend increased cognitive effort to diagnose faults and errors in the IUT and the specification, respectively.

Therefore, we propose an approach to reconstruct a symbolic timed trace from a test sequence by applying backward-induction. Backward-induction fits well to our needs since it is able to deal with random behaviour and to reconstruct the initial state of a problem starting from its outcome. In the following, we describe our approach to reconstruct a symbolic trace using backward-induction. In short, the approach takes as input a test sequence and returns a symbolic trace which can be imported in Uppaal. There are three main steps: (1) build an initial game tree; (2) define an outcome function and (3) identify the test path in the form of a symbolic trace.

3.1 Building the initial game tree (Γ) from *rtioco*-based test sequence

From the test sequence provided by Uppaal Tron after a test session, we build the *initial game tree* (Γ) as a set of nodes, belonging to different collections, each collection having multiple layers. The game tree is constructed as a one-to-one mapping to the structure of the Uppaal Tron test sequence shown in Figure 3) as follows. A *collection* represents an observable delay in the test sequence. Internally, each observable delay consists of a set of internal transitions between symbolic states corresponding to different clock zones. Each such clock zone is represented as a *level* in the collection. The levels are in chronological order. The *nodes* included in a given level depict the set of next available symbolic states that were available before executing an internal transition and moving to a new clock zone. These nodes are used as alternatives from which a single solution is selected to be included in the test path.

3.2 Outcome function for Γ

In order to identify a path in the game tree Γ, we need to identify the predecessor node $Node_{l_i}$ of a $Node_{l_j}$ situated on level i and, respectively j, where $i < j$.

For this we define an outcome function that calculates the backward *reachability* of node $Node_{l_i}$ from $Node_{l_j}$ as follows.

$$\int (Node_{l_i}, Node_{l_i}, E_{\prod}) =$$

$$(Node_{l_i} \xrightarrow{\delta} Node_{l_j}) \models (g \wedge z_{l_j})_r \wedge I(Node_{l_j}) \wedge \delta \in E_{\prod} \quad (7)$$

where g is a guard of the transition between the symbolic states corresponding to the to nodes, z_{l_j} is the clock zone corresponding to level l_j of the tree, I is the invariant of the symbolic state corresponding to $Node_{l_j}$, and δ is an observable delay or an unobservable action associated with test event $E_{\prod}$.

Since the two nodes have been explored during the test generation process, we know that $Node_{l_j}$ is reachable from $Node_{l_i}$.

3.3 Identification of a test path

Using the outcome function, we can identify the path in the tree Γ corresponding to the symbolic trace of the test session, as shown in Algorithm 1. The starting node of the algorithm is the one corresponding to the terminal state which is selected as a goal state. Then we search (lines 4 – 10) for a predecessor amongst the alternatives on the level above it. Once a predecessor node is found, the goal and the predecessor node along with the connected action are pushed to a stack *XTR*. The goal state is updated with the discovered predecessor node and the search process continues until algorithm reaches the initial node.

Figure 4 shows a generic example of a reconstructed symbolic trace using our approach. The resulting trace can then be loaded in the Uppaal simulator and used to visualize the trace in the model.

Algorithm 1 Trace Reconstruction algorithm to transform a game tree Γ to a test path

```
1:  function TRANSFORM(Γ)
2:      XTR := ∅        → Initilaze a stack XTR
3:      Goal :=Terminal Node        → Initilaze the goal state
4:      for (k := n − 1; k > 0; k − −) do        → n collection in a game tree
5:          E_k := E_∏k        → Collection events
6:          for (j := m; j ≥ 0; j − −) do        → m level of alternatives
7:              for (i := 0; i ≤ p; i + +) do
8:                  if ∃ e = (∫(Node_i^m, Goal, E_k)) then
9:                      PUSH(Node_i^m, e, Goal) → XTR;
10:                     Goal := Node_i^m;        → update goal with new node
11:                     break;
12:                 end if
13:             end for
14:         end for
15:     end for
16:     return XTR
17: end function
```

3.4 Tool support

During testing, the IUT is attached to Uppaal Tron via a test adapter [20] and the user supplies Uppaal Tron with a UTA model. The test primitives are generated directly from the model, executed through the adapter, and the system responses are checked. The state transitions evolve the testing session into a next state along with new clock and variable valuations.

The proposed TR algorithm was implemented in Java as a prototype tool called "Tron2Uppaal Back-Tracker"[1]. The tool uses *Uppaal Timed Automata Parser Library* [13] to parse the UTA model.

[1]available at http://users.abo.fi/jiqbal/back-tracker/

As shown in Figure 5, our tool takes as an input a Tron test run log file, a UTA model, and an intermediate format file. The latter contains an internal representation of the model using the numeric indices used by UPPAAL to improve the performance and to reduce the memory utilization during model-checking. These indices are used to map the elements of XTR stack to the elements of the UTA model. The tool produces a Uppaal simulation trace which can be imported into Uppaal. More technical details about the implementation of the tool can be found in the technical report [18].

4 EXAMPLES AND EVALUATION

The applicability of our approach is exemplified using the *smart lamp light controller* example available with the Uppaal Tron distribution [22] and a real-time *temperature control system* presented in [27].

4.1 Smart lamp light controller (SLLC)

The UTA model of SLLC mainly consists of seven component automata namely *interface, dimmer, switcher graspAdapter, release adapter, levelAdapter* and *user* shown in Figures 6—10c. The user automaton models the environment. The other automata model the expected behaviour of the SLLC, where the level of light is defined by the time interval between consecutive user *grasp* and *release* events. Tron is behaving like a smart lamp user by issuing grasp and release events and at the same time observing if the level of light is correct according to specification of light controller behaviour written in Uppaal timed automata modelling language [22]. Moreover, the *grasp-*, *release-* and *level-* adapter processes shown in Figure 10a, 10b and 10c, model the latency of issuing the input-stimuli.

4.2 Temperature control system (TCS)

The TCS is a piece of software responsible for controlling a host device temperature by sensing a number of sensors (between 20 to 40) and controlling the speed of several fans situated at different physical locations. Due to space limitation we defer details to [27]. The UTA model for TCS consists of one *temperature process* for each sensor (Fig.12), a *fan process* for each fan (Fig.11) and the *environment* process (Fig.13).

4.3 Evaluation of tool support

To evaluate our tool with SLLC and TCS, we used an Intel based 64-bit quad-core machine running at 3.20 GHz with 8 GB of RAM. The maximum size of allocated memory for the tool was 1024 MB by using virtual machine option Xmx.

For each of the two examples, we executed five test sessions of different lengths as shown in Table 1, which resulted in 10 test sequences with different number of events. We used the VisualVM profiler [24] to benchmark the time and memory used by Tron2Uppaal-Back-tracker tool. The results are shown in Figure 14. Figures 14 (a) and (b) show that the transformation time and memory used varies proportionally with the size of the test sequence.

As one may notice, the time used to reconstruct the symbolic trace is directly proportional to the size of the test sequence due to file read operations. However, the memory utilization levels off due to the fact that the tool maintains information about the

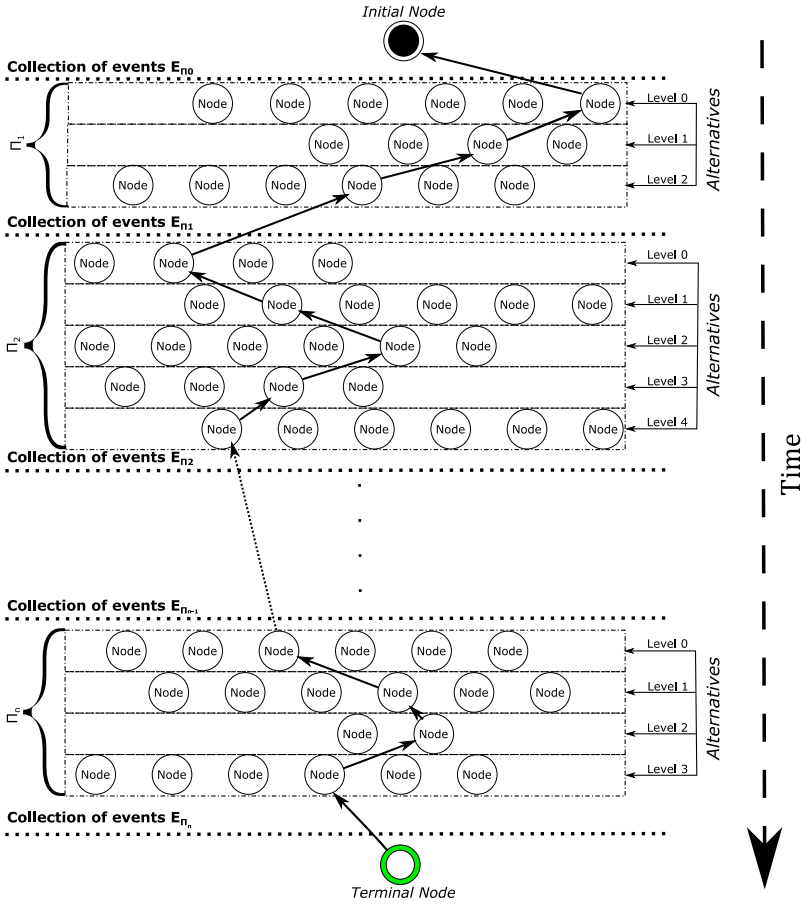


Figure 4: Test path identified by outcome function

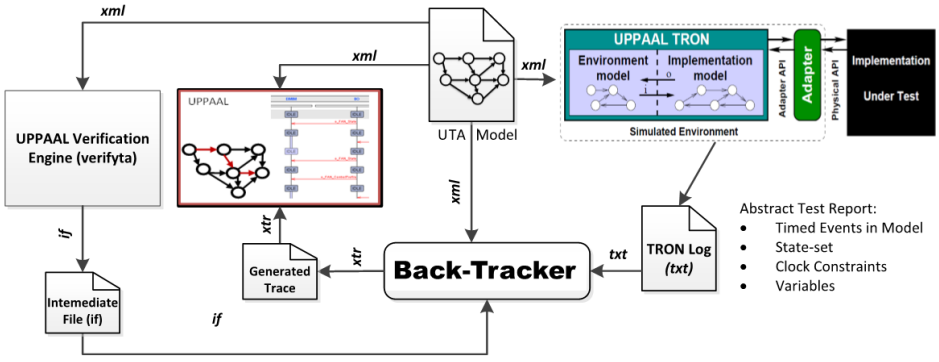


Figure 5: Back-Tracker tool setup

states and transitions. When most of the information is already present, the tool reuses it for other events which occur later. Further optimizations can be applied to the tool in order to reduce the transformation time and memory usage during the transformation process which is beyond the scope of this paper. Nevertheless, the results show that the approach has the potential to scale for large test logs.

5 RELATED WORK

The closest work we found to our approach was proposed by [25] which describes several approaches to generating concrete delays

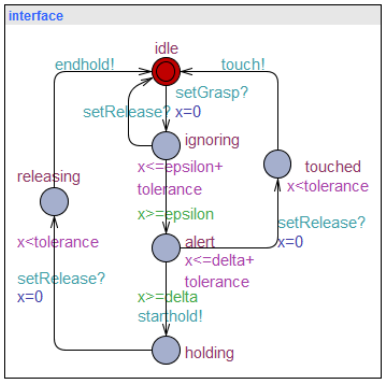


Figure 6: Interface

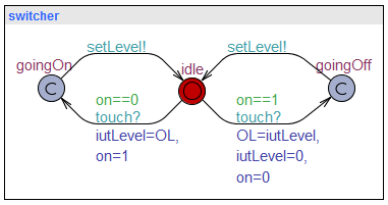


Figure 7: Switcher

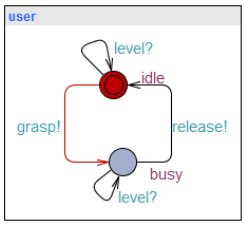


Figure 8: user

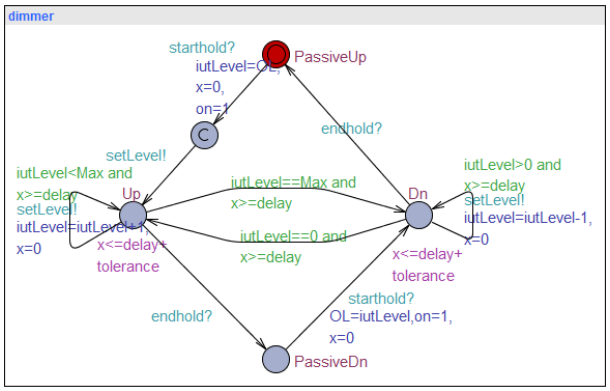
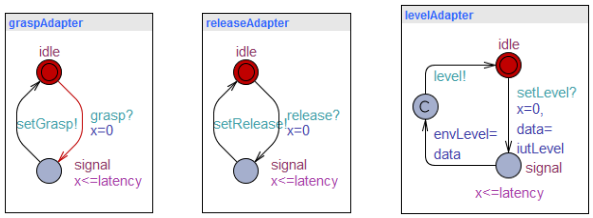


Figure 9: Dimmer



(a) grasp- **(b) release-** **(c) level-**

Figure 10: Adapters in SLLC

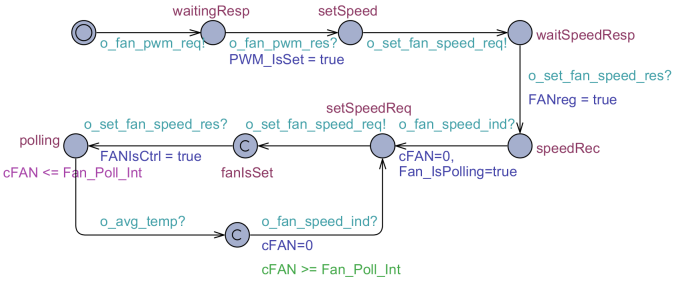


Figure 11: Model for fan speed controller

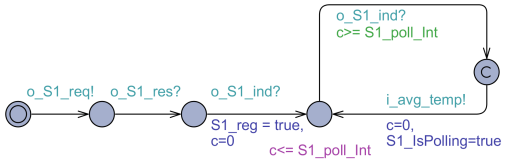


Figure 12: Model for sensor

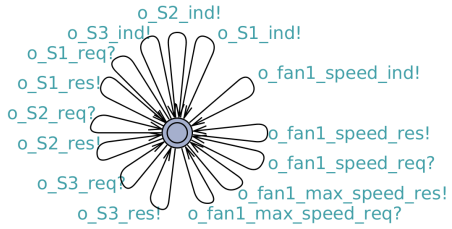


Figure 13: Environment model

for diagnostic traces. Two of these approaches are based on the diagnostic traces that violate safety properties and one approach is based on the diagnostic traces that violate liveness properties. Our approach differs in two ways. Firstly, we reconstruct the symbolic trace from the timed test sequences and visualise the symbolic trace by loading the reconstructed symbolic trace into Uppaal simulator. Secondly, we use backward-induction to identify the predecessor symbolic state, while they are using backward re-compositionality.

Franck *et. al.* [11] proposed an algorithm based on on-the-fly strategy for timed games and Behrmann *et. al.* [1] developed an extension of Uppaal called Uppaal-Tiga based on game theory. The author proposed timed-games on-the-fly strategies for a control program. Another work [26] describes the reconstruction of the symbolic state space by using the *use-definition chains* to reduce symbolic state space and proposed reductions in transformations required to reconstruct a state space concerning clock operations. Contrarily to [1] and [26], our approach rebuilds the part of the state space involved in the test run.

6 CONCLUSIONS

We suggest a tool-supported approach to reconstruct timed symbolic traces from *rtioco*-based timed test sequences with the purpose

Table 1: Computational results

Case study	Test events	Transformation time (mili-sec)	Memory utilization (bytes)
TCS	469	741	9 786 848
	725	908	21 816 888
	27 003	10 424	213 396 016
	209 893	90 032	327 112 424
	4 949 651	2 074 653	378 172 784
SLLC	711	1 124	394 44 952
	1 145	1 720	51 103 440
	39 554	34 851	279 789 688
	156 480	137 918	319 562 376
	1 787 238	1 123 903	347 847 200

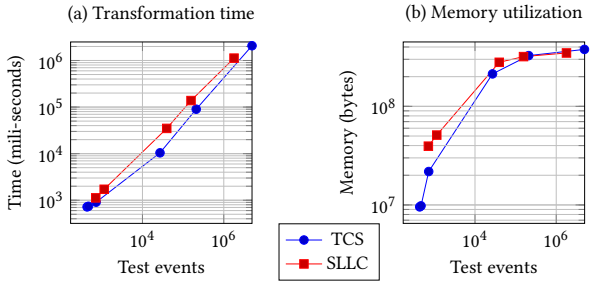


Figure 14: Graphs showing (a) Transformation time, (b) Memory utilization

of analysing and visualizing a given test session in the Uppaal simulator. The proof of concept and applicability of our reconstruction approach have been demonstrated on two examples; a smart lamp light controller and a temperature control system. The performance of the algorithm used is not influenced by the verdict of the test session, but only by the number of events in the test sequence. The results show that the approach is feasible and our first prototype has demonstrated satisfactory scalability.

Acknowledgement: This work has been *__partially__* supported by the ECSEL JU MegaM@Rt2 project under grant agreement No 737494.

REFERENCES

[1] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. 2007. *UPPAAL-Tiga: Time for Playing Games!* Springer Berlin Heidelberg, Berlin, Heidelberg, 121–125. https://doi.org/10.1007/978-3-540-73368-3_14

[2] Gerd Behrmann, Alexandre David, and Kim G. Larsen. 2004. *A Tutorial on Uppaal.* Springer Berlin Heidelberg, Berlin, Heidelberg, 200–236. https://doi.org/10.1007/978-3-540-30080-9_7

[3] Axel Belinfante, Jan Feenstra, René G. de Vries, Jan Tretmans, Nicolae Goga, Loe Feijs, Sjouke Mauw, and Lex Heerink. 1999. *Formal Test Automation: A Simple Experiment.* Springer US, Boston, MA, 179–196. https://doi.org/10.1007/978-0-387-35567-2_12

[4] Johan Bengtsson. 2002. *Clocks, DBMs and States in Timed Systems.* Ph.D. Dissertation. Dept. of Information Technology, Uppsala University.

[5] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1996. *Hybrid Systems III: Verification and Control.* Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter UPPAAL — a tool suite for automatic verification of real-time systems, 232–243. https://doi.org/10.1007/BFb0020949

[6] Johan Bengtsson and Wang Yi. 2004. *Timed Automata: Semantics, Algorithms and Tools.* Springer Berlin Heidelberg, Berlin, Heidelberg, 87–124. https://doi.org/10.1007/978-3-540-27755-2_3

[7] Walter Bossert and Yves Sprumont. 2013. Every choice function is backwards-induction rationalizable. *Econometrica* 81, 6 (2013), 2521–2534. http://www.jstor.org/stable/23524325

[8] Victor Braberman, Miguel Felder, and Martina Marré. 1997. Testing Timing Behavior of Real-Time Software. In *In International Software Quality Week.* Proceedings of the 10th International Software Quality Week '97, San Francisco, CA.

[9] Laura Brandán Briones and Ed Brinksma. 2005. *A Test Generation Framework for quiescent Real-Time Systems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 64–78. https://doi.org/10.1007/978-3-540-31848-4_5

[10] Rachel Cardell-oliver and Tim Glover. 1998. A Practical and Complete Algorithm for Testing Real-Time Systems. In *In Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 1486.* Springer-Verlag, Berlin, Heidelberg, 251–261.

[11] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. 2005. *Efficient On-the-Fly Algorithms for the Analysis of Timed Games.* Springer Berlin Heidelberg, Berlin, Heidelberg, 66–80. https://doi.org/10.1007/11539452_9

[12] D. Clarke and Insup Lee. 1997. Automatic test generation for the analysis of a real-time system: Case study. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE.* IEEE, Montreal, Quebec, Canada, 112–124. https://doi.org/10.1109/RTTAS.1997.601349

[13] Alexandre David. 2007. Uppaal Timed Automata Parser Library. (2007). Retrieved March 11,2015 from http://people.cs.aau.dk/~adavid/utap/

[14] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. 1996. Using on-the-fly verification techniques for the generation of test suites. In *Computer Aided Verification*, Rajeev Alur and ThomasA. Henzinger (Eds.). LNCS, Vol. 1102. Springer Berlin Heidelberg, 348–359. https://doi.org/10.1007/3-540-61474-5_82

[15] Paula Herber. 2010. *A Framework for Automated HW/SW Co-Verification of SystemC Designs using Timed Automata.* Logos Verlag Berlin GmbH.

[16] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. 2008. Testing Real-Time Systems Using UPPAAL. In *Formal Methods and Testing*, RobertM. Hierons, JonathanP. Bowen, and Mark Harman (Eds.). LNCS, Vol. 4949. Springer Berlin Heidelberg, 77–117. https://doi.org/10.1007/978-3-540-78917-8_3

[17] Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. 2004. *Time-Optimal Real-Time Test Case Generation Using Uppaal.* Springer Berlin Heidelberg, Berlin, Heidelberg, 114–130. https://doi.org/10.1007/978-3-540-24617-6_9

[18] Junaid Iqbal, Dragos Truscan, Jüri Vain, and Ivan Porres. 2015. *TRON2UPPAAL Backtracer Tool – From TRON Logs to UPPAAL Traces.* Technical Report 1138. Turku Centre for Computer Science. Online at http://tucs.fi/publications/view/?pub_id=tIqTrVaPo15a.

[19] Thierry Jéron. 2009. Symbolic Model-based Test Selection. http://www.sciencedirect.com/science/article/pii/S157106610900173X. *Electronic Notes in Theoretical Computer Science* 240 (2009), 167 – 184. https://doi.org/10.1016/j.entcs.2009.05.051 Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008).

[20] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. 2005. Online Testing of Real-time Systems Using Uppaal. In *Formal Approaches to Software Testing*, Jens Grabowski and Brian Nielsen (Eds.). LNCS, Vol. 3395. Springer Berlin Heidelberg, 79–94. https://doi.org/10.1007/978-3-540-31848-4_6

[21] Jiangtao Li and Rui Tang. 2016. Every Random Choice Rule is Backwards-Induction Rationalizable. (2016).

[22] Marius Mikučionis. 2012. Uppaal TRON. http://people.cs.aau.dk/~marius/tron/. (2012). [Online; accessed 20-April-2015].

[23] Brian Nielsen. 2000. *Specification and Test of Real-Time Systems.* Ph.D. Dissertation. Aalborg Universitetsforlag, Denmark.

[24] Oracle. 2015. VisualVM. http://visualvm.java.net/. (2015). [Online; accessed 13-May-2015].

[25] Danny Bøgsted Polsen and Jones van Vliet. 2010. *Concrete Delays for Symbolic Traces.* Master Thesis. Department of Computer Science, Aalborg University.

[26] Jonas Rinast, Sibylle Schupp, and Dieter Gollmann. 2013. State Space Reconstruction for On-Line Model Checking with UPPAAL. In *VALID 2013, The Fifth Internation Conference on Advances in System Testing and Validation Lifecycle.* 21–26.

[27] Dragos Truscan, Tanwir Ahmad, Faezeh Siavashi, and Pekka Tuuttila. 2015. A Practical Application of UPPAAL and DTRON for Runtime Verification. In *Proceedings of the Second International Workshop on Software Engineering Research and Industrial Practice (SER&IP '15).* IEEE Press, Piscataway, NJ, USA, 39–45. http://dl.acm.org/citation.cfm?id=2821387.2821397

[28] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. http://dx.doi.org/10.1002/stvr.456. *Software Testing, Verification and Reliability* 22, 5 (2012), 297–312. https://doi.org/10.1002/stvr.456 online; last accessed : 26.05.2015.

[29] John Von Neumann and Oskar Morgenstern. 2007. *Theory of games and economic behavior.* Princeton university press.