TRON2UPPAAL Backtracer Tool – From TRON Logs to UPPAAL Traces

Iqbal, Junaid; Truscan, Dragos; Vain, J; Porres Paltor, Ivan

Published: 01/01/2015

Document Version
Final published version

TUCS

Junaid Iqbal | Dragos Truscan | Jüri Vain | Ivan Porres

Tron2Uppaal Back Tracer Tool

– from TRON logs to UPPAAL traces

Turku Centre *for* Computer Science

TUCS Technical Report
No 1138, June, 2015

Tron2Uppaal Back Tracer Tool

– from TRON logs to UPPAAL traces

Junaid Iqbal
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
junaid.iqbal@abo.fi
Dragos Truscan
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
dragos.truscan@abo.fi
Jüri Vain
Tallinn University of Technology, Tallinn, Estonia
vain@ioc.ee
Ivan Porres
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3-5A, 20520 Turku, Finland
ivan.porres@abo.fi

TUCS Technical Report
No 1138, June, 2015

Abstract

UPPAAL TRON is a tool for on-line black-box conformance testing of real-time systems. Typically, the model-based specifications are edited, simulated, and verified in the UPPAAL model-checker and, later on, used in UPPAAL TRON for testing the Implementation Under Test (IUT). Whenever a non-conformance is detected between the specifications and IUT, a "failed" or "inconclusive" verdict is given. Analysing the test logs in order to identify the path taken in the specification and the source of the failure is tedious and time consuming, especially for long lasting test sessions. Therefore, in order to reduce the time and effort for debugging test sessions, we propose a tool-supported approach to transform TRON logs into a UPPAAL simulation trace. This allows us to take advantage of UPPAAL's capabilities for simulation and visualisation of the test traces. The approach is exemplified and evaluated on a traditional Smart Light controller example provided by UPPAAL TRON. The results show that the approach is scalable enough to be applied to more complex examples.

Keywords: UPPAAL, UPPAAL TRON, Difference Bounded Matrix, Active clock reduction, TRON Log transformation

1 Introduction

Model-based testing (MBT) [25] [26] is a testing approach which reduces the effort needed for testing. In MBT, test cases are generated as a whole or in part from a abstract model that describes the expected behaviour of the System Under Test (SUT). The two approaches for MBT test execution are on-line and off-line [26]. In the off-line test execution approach, the test suits are generated at once and executed later on via test automation frameworks. In contrast, on-line testing combines *on-the-fly* test generation and test execution. In the case of on-line test execution, the abstract test suite exists only as a concept but not as an explicit artifacts. In on-line testing, a single test primitive is generated from the model of IUT which is then mapped to an executable input on the IUT. The test inputs are designed and executed one at a time and then the produced output by the IUT as well as its time of occurrence are checked against the specification. A new test primitive is produced and so forth until it is decided to end the test or an error is detected [22].

The black-box conformance testing for real-time systems is an approach to check the external behaviour of a given implementation under test (IUT) against its formal specification. The IUT is black-box and thus we only rely on its observable input/output behaviour [19]. Automated tools for test generation have been developed for various languages and models, both un-timed [3, 15, 18] and timed [5–7, 17].

UPPAAL is an integrated tool environment for modelling, validation and verification of real-time systems [2]. The environment assumptions and system requirements are modelled as network of timed automata referred as UPPAAL Timed Automata (UPTA). UPPAAL TRON(Testing for Real-time system ONline) [24] in short TRON, an extension UPPAAL, which is used for on-line black-box conformance testing of real-time systems. The test generation in TRON is performed via symbolic execution [21]. At each step, the available symbolic states to be visited, are calculated and a decision is taken based on the received input or via random choice. A test session stops when the model goes to a final state, the test duration expires or an error is encountered. The TRON presents the progress of the test session in terms of symbolic states, available inputs and expected outputs, and valuations of clocks and integer variables in the models. Depending on the verbosity level of the logging, an observed test run is a timed trace consisting of a sequence of (input or output) actions and time delays [23]. In addition,

In many situations, especially when a failed or inconclusive test result is encountered, one would like to understand and visualise which traces have been taken on the model in a given test session, which symbolic states have been visited, and what were the values of the variables in each state. Such information is rather difficult to obtain from the test execution logs of TRON, especially for long lasting sessions.

Contribution: This work proposes a tool-supported approach to transform TRON testing logs into UPPAAL simulation traces, for *a posteriori* analysis. In order to implement the approach, we defined an algorithm for the reconstruction of the model's symbolic runs in order to expose the causes of test fails on the level of user comprehensible abstraction. The benefit of our approach is that it allows one to check the sequence of symbolic states taken in the model, including clock and variable valuations, as well as channel synchronizations. This allows one to take advantage of UPPAAL's capabilities for simulation and visualisation, and thus to improve the debugging process and reduce cognitive effort needed to identify the underline causes of inconclusiveness or failure.

Related Work. To our best of knowledge there is no other work on converting TRON test logs into UPPAAL simulation traces. The one of closest work to ours is used by David in the UPPAAL Automata Parser library [9] which is used to transform the UPPAAL simulator trace into human-readable format. The second closest work is used by de Kock [13] named UPPAAL2OCTOPUS developed by Embedded System Innovation, which converts UPPAAL simulation trace to Octopus-format latter used by a tool called ResVis. In contrast, our approach converts the TRON testing log into a UPPAAL simulation trace by reconstructing the symbolic state space.

This paper is divided into the following sections. Section 2 revisits background information and concepts related to timed automata, UPPAAL and TRON. Section 3 details our transformation approach, followed by a brief overview of tool support in Section 4. We exemplify and evaluate the scalability our approach with a Smart Lamp controller example in Section 5. The conclusions will be outlined in Section 6.

2 Background

In the following subsection, we briefly discuss the timed automata formalism, followed by a short introduction of the UPPAAL and TRON tools.

2.1 Timed Automata

A timed automaton $\mathcal{A}$ is a tuple $(\mathcal{S}, \mathcal{X}, \mathcal{L}, \varepsilon, I)$ [10] where $\mathcal{S}$ is a finite set of *locations*, the $s_0 \in \mathcal{S}$ denotes the *initial location* and $\mathcal{X}$ is a finite set of *clocks*. The finite set of *labels* is denoted by $\mathcal{L}$ and ε is a finite set of *edges*. Each edge e is a tuple (s, L, ψ, ρ, s') where $s \in \mathcal{S}$ is the source location and

$s' \in \mathcal{S}$ is the target location, $L \subseteq \mathcal{L}$ is a set of *action labels* (input and output actions), $\psi \in \Psi_{\mathcal{X}}$ it the enabling condition, and $\rho : \mathcal{X} \to \mathcal{X}^*$ is the assignment function. I is a function that associates a conditions $I(s) \in \Psi_{\mathcal{X}}$ to every control location $s \in \mathcal{S}$ called *invariant* of s. A *state* of $\mathcal{A}$ is a pair $(s, v) \in \mathcal{S} \times \mathcal{V}_x$ such that v satisfy the condition $I(s)$. At any state, $\mathcal{A}$ can evolve either by moving through an edge that changes the location and the value of some clocks *(discrete transition)*, or by letting time pass with out changing the location *(time transition)* [11].

Discrete transitions. Let $e = (s, L, \psi, \rho, s') \in \epsilon$ be an edge. The automaton has discrete transition from state (s, v) to state (s', v'), denoted $(s, v) \longrightarrow_0^L (s', v')$, if v satisfies ψ and $v' = v[\rho]$ [11].

Time transitions. Let $t \in \mathbb{R}^+$. The state (s, v) has a time transition to $(s, v + t)$. denoted $(s, v) \longrightarrow_t^{\emptyset} (s, v + t)$, if for all $t' \leq t, v + t'$ satisfies the invariant $I(s)$ [11].

Clocks, Bounds and Zones. Let $\chi = \{x_1, \ldots, x_n\}$ be a set of variables called clocks, ranging over the non-negative reals $\mathbb{R}_{\geq 0}$. A *clock valuation* is a function $v : \chi \mapsto \mathbb{R}_{\geq 0}$, assigning to each clock x a non-negative real value $v'(x)$. For $X \subseteq \chi, v[X := 0]$ is the valuation v', such that $\forall x \in X.\ v'(x) = 0$ and $\forall x \notin X. v'(x) = v(x)$. For every $t \in \mathbb{R}_{\geq 0}$, $v + t$ is the valuation v' such that $\forall x \in \chi.\ v(x) = v'(x) + t$.

A *bound* over X is a constraint of the form $x_i \# c$ or $x_i - x_j \# c$, where $1 \leq i \neq j \leq n, \# \in \{<, \leq, \geq, >\}$ *and* $c \in \mathbb{N} \cup \{\infty\}$. In order to have unified form for clock constraints (so called *clock difference*), [4] introduces a *reference clock* **0** with constant value of 0. All the other clocks are assumed to progress with same speed. If we introduce a reference clock variable x_0 to represent **0**, then bounds can be uniformly written as:

$$x_i - x_j \prec d \ , \ where\ 0 \leq i \neq j \leq n, \prec \in \{<, \leq\}\ and\ d \in \mathbb{Z} \cup \{\infty\} \text{ [12].}$$

The set of clocks including the reference clock became $\mathcal{C}_0 = \mathcal{C} \cup \{\mathbf{0}\}$, then any *zone* $D \in \mathcal{B}(\mathcal{C})$ can be rewritten as a conjunction of constraints of the form:

$$x - y \preceq n\ for\ x, y \in C_0, \preceq \in \{<, \leq\} \text{ and } n \in \mathbb{Z} \text{ [4].}$$

If the rewritten zone has two constraints on the same pair of variables, only the tightest of them is significant. Thus, a zone can be represented using at most $|\mathcal{C}_0|^2$ atomic constraints of the form of $x - y \leq n$ such that each pair of clocks $x - y$ is mentioned only once. These zones are then stored in $|\mathcal{C}_0| \times |\mathcal{C}_0|$ matrices, where each element in such a matrix represents the bound on the difference between two clocks [4]. The matrix is called *Difference Bound Matrix* (DBM) [14].

Since zones are frequently used objects in symbolic state-space exploration, their effective representation is a major issue when building a verification tool [4]. UPPAAL verification engine computes reachability relation on zones [16]. The zones represent the reachable set of states and clock valuations and represented symbolically as DBM. The DBM are the key objects for symbolic state-space exploration for timed system.

To compute the DBM representation for a zone D, all clocks in $\mathcal{C}_0$ are numbered and assigned one row and one column in the matrix. The row is used to store lower bounds on the difference between the clocks and all other clocks whereas the columns are used for upper bounds [4]. The elements in the matrix are then computed, according to [4] in three steps.

- For each bound in $x_i - x_j \leq n$, in D, Let $D_{ij} = \{n, \leq\}$.
- For each clock difference $x_i - x_j$, that is unbounded in D, let $D_{ij} = \infty$, Where ∞ is a special value denoting that no bound is present.
- Finally add the implicit constraints that all clocks are positive *i.e.* $\mathbf{0} - x_i \leq 0$ and that the difference between a clock and itself is always 0, *i.e.* $x_i - x_i \leq 0$.

As an example, consider the zone $D = x - \mathbf{0} < 20 \wedge y - \mathbf{0} \leq 20 \wedge y - x \leq 10 \wedge x - y \leq -10$. To construct the matrix representation of D, the clocks are numbered as $\mathbf{0}, x, y$. The resulting matrix representation is shown as:

$$M(D) = \begin{pmatrix} (0, \leq) & (0, \leq) & (0, \leq) \\ (20, \leq) & (0, \leq) & (-10, \leq) \\ (20, \leq) & (10, \leq) & (0, \leq) \end{pmatrix}$$

2.2 UPPAAL

UPPAAL is a tool for modelling, simulation and verification of real-time systems and a suitable choice for modelling of set of non-deterministic process having finite control structure and real-valued clocks. The model processes communicate via channels or shared variables. Typical area of application of the tool includes real-time controller and communication protocols [1].

The main components of UPPAAL are: a model editor, a simulator, and a model checker. The editor allows modelling of UPPAAL timed automata (UPTA) which is an extension of TA with bounded integer variables and simple data types. It serves as a modelling or design language to describe system behaviour as networks of automata extended with clock and data variables. The simulator is a validation tool which enables the examination of possible dynamic executions of a system during early design (or modelling) stages. It provides an inexpensive means of fault detection prior to verification by the model checker. The latter provides exhaustive coverage of dynamic behaviour of the system. The simulator also allows visualization of error (diagnostic) traces found as result of verification efforts. The model checker is used to check invariant and bounded-liveness properties by exploring the symbolic state-space of a system, *i.e.,* reachability analysis in terms of symbolic states represented by constraints [1].

In addition to features listed above, UPPAAL modelling language supports process templates and (bounded) data structures as data variables, constants, arrays, etc. A process template is a timed automaton extended with a list of formal parameters and a set of locally declared clocks, variables, and constants. Typically, a system description consists of a set of instances of timed automata declared as a parametrized instance of the process templates, and of some global data, such as global clocks, variables, synchronization channels, etc. In addition, the instances may be defined from templates re-used from existing system descriptions. Thus, the adopted notion of process templates (particularly when used in combination with the possibility to declare local process data) allows for convenient re-use of existing models [1].

2.2.1 UPPAAL Simulation and Trace format:

In UPPAAL, a system is modelled as a network of several timed automata in parallel. The (symbolic) state of the system is defined by the locations of all automata, the constraints currently satisfied, and the values of the discrete variables [2]. One symbolic state is displayed at a time, where the control locations are visualized with highlighted locations in the timed automata graphs and data is shown by means of variable valuation and clock constraints, as shown in Figure 1.

In the beginning, all processes are in the initial locations. The initial entry of simulator trace consists of processes locations, zone entry, variables as described in previous section. Subsequent entries consist of process locations, zone, variables and the transition taken to evolve to the next state. To represent the transitions in a simulation trace, the processes index with the edge number is present as $m\ n$ where $m \in processes$ and $n \in \varepsilon$.

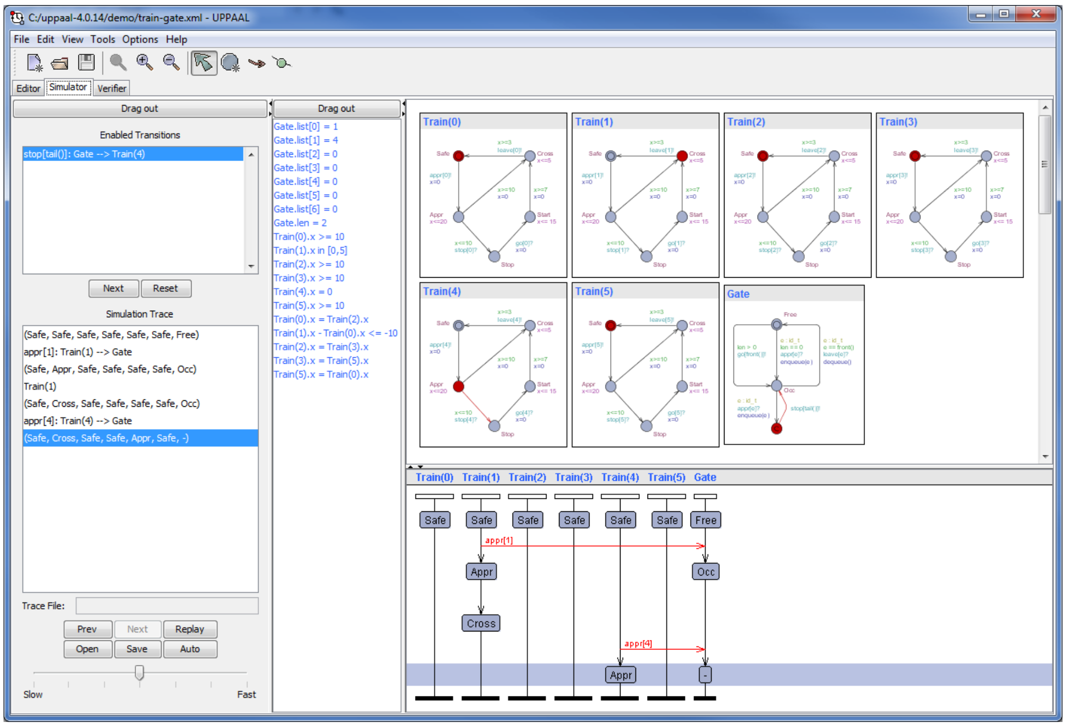
While simulating a model in UPPAAL, a trace is obtained consisting of:

1. Location(s) of the processes in model.

2. Zone (clock information).

3. Variables.

4. Transition taken by simulator.



```
stop[tail()]: Gate --> Train(4)
```

```
(Safe, Safe, Safe, Safe, Safe, Safe, Free)
appr[1]: Train(1) --> Gate
(Safe, Appr, Safe, Safe, Safe, Safe, Occ)
Train(1)
(Safe, Cross, Safe, Safe, Safe, Safe, Occ)
appr[4]: Train(4) --> Gate
(Safe, Cross, Safe, Safe, Appr, Safe, -)
```

```
Gate.list[0] = 1
Gate.list[1] = 4
Gate.list[2] = 0
Gate.list[3] = 0
Gate.list[4] = 0
Gate.list[5] = 0
Gate.list[6] = 0
Gate.len = 2
Train(0).x >= 10
Train(1).x in [0,5]
Train(2).x >= 10
Train(3).x >= 10
Train(4).x = 0
Train(5).x >= 10
Train(0).x = Train(2).x
Train(1).x - Train(0).x <= -10
Train(2).x = Train(3).x
Train(3).x = Train(5).x
Train(5).x = Train(0).x
```

Figure 1: UPPAAL Simulator GUI

The format of the simulation trace is defined based on the following **exteded BNF**.

$$
\begin{aligned}
\langle trace\rangle \quad ::= \; & \langle state\rangle \; (\langle state\rangle \; \langle transition\rangle)^* \\
& \langle state\rangle \; ::= \; \langle locvec\rangle \; \langle zone\rangle \; \langle varvec\rangle \\
& \langle locvec\rangle \; ::= \; (\langle location\rangle \; \langle nl\rangle)^* \; \langle dot\rangle \; \langle nl\rangle \\
& \langle zone\rangle \; ::= \; (\langle clock\rangle\langle nl\rangle\langle clock\rangle\langle nl\rangle\langle bound\rangle\langle dot\rangle)^* \; \langle dot\rangle \; \langle nl\rangle \\
& \langle varvec\rangle \; ::= \; (\langle NUM\rangle \; \langle nl\rangle)^* \; \langle dot\rangle \; \langle nl\rangle \\
& \langle transition\rangle \; ::= \; (\langle process\rangle \; \langle space\rangle \; \langle edge\rangle \; \langle nl\rangle)+ \; \langle dot\rangle \; \langle nl\rangle
\end{aligned}
$$

where ⟨nl⟩ is a newline character, ⟨space⟩ is a space character, ⟨dot⟩ is a dot character and ⟨NUM⟩ is an integer value. ⟨location⟩, ⟨clock⟩, ⟨process⟩ and ⟨edge⟩ are integers referring to a location, clock, process or edge, respectively.

The zone entry consist of clock and values represented in DBM format i.e. $x - y \leq n$ where x, y is the clock number in DBM and $n \in \mathbb{Z}$. In order to save space in memory, active clock reduction filter is applied to rectify the clocks which are not active or equals. The algorithm to reduce number of clock was presented in [11]. "*a clock is active at some control location if its value at the location may influence the future evolution of the system. This may happen whenever the clock appears in the invariant condition of the location, it is tested in the condition of some of the outgoing edges, or an active clock takes its value when moving through an outgoing edge.*"

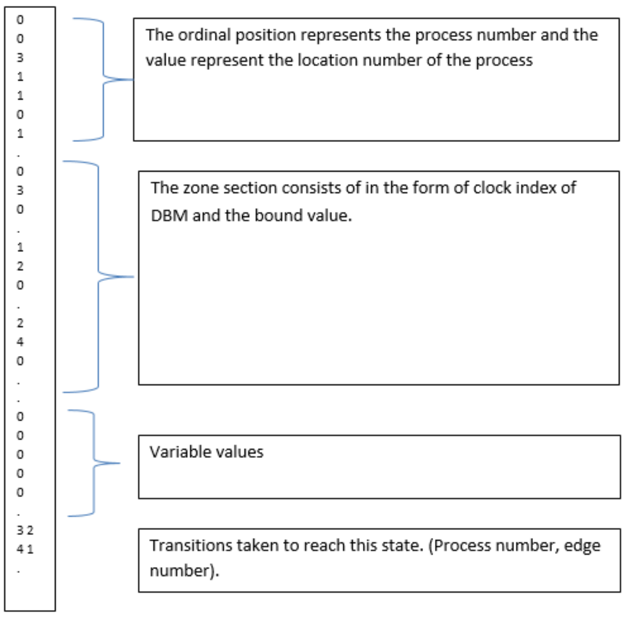

0
0
3
1
1
0
1
.
0
3
0
.
1
2
0
.
2
4
0
.
.
0
0
0
0
0
.
3 2
4 1
.
The ordinal position represents the process number and the value represent the location number of the process
The zone section consists of in the form of clock index of DBM and the bound value.
Variable values
Transitions taken to reach this state. (Process number, edge number).

Figure 2: Example for Trace format generated by the UPPAAL Simulator

For example, the UPPAAL simulation trace shown in Figure 2 can be divided into four major sections. The initial section contains location index of the process identified by its ordinal position. The zone section contains the clock constraints consist of clock index of DBM and the bound value. The zone section only contains active clocks and one of the equal clocks (see [11] for details) at a given state-set. The variable vector section is used to store the variable's values. The ordinal position of the values relates to the variable which appears in layout section of the intermediate format.

2.2.2 UPPAAL Intermediate format Structure:

The Intermediate format was introduced in UPPAAL 3.6. During the verification process, the UPTA model is transformed into an intermediate format as shown in Figure 5. The intermediate format is then used to identify processes, locations, edges, guards, updates, synchronizations and variables. The TRON follows the same approach during online testing. The intermediate format can be explicitly generated with UPPAAL verification engine called *verifyta* and stored in a file for further use. Figure 5 provides a summary of the *intermediate format (IF)*, based on the following structure.

Layout: This section contains information about constants, clocks, variables, meta-data, cost, locations and static. The clock information contains clock name and number in DBM. Variables and meta information is supplemented by minimum and maximum value, initial value and its ordinal

```
layout
#index:const:value
#index:clock:nr:name
#index:var:min:max:init:nr:name
#index:meta:min:max:init:nr:name
#index:cost
#index:location:flags:name
#index:static:min:max:name
26:clock:1:interface.x
27:location::idle
28:location::ignoring
29:location::alert
30:location::touched
31:location::releasing
32:location::holding
33:const:17
34:const:13
35:location::idle
36:location:committed:goingOn
37:location:committed:goingOff.
.
.
68:clock:5:levelAdapter.x
69:var:-32767:32767:0:4:
    ↪ levelAdapter.data
70:location:committed:_id16
71:location::idle
72:location::signal
.
.
.

instructions
#address:  opcode
0:  0 1        push 1
2:  33         halt
3:  0 0        push 0
5:  33         halt
6:  0 1        push 1
8:  33         halt
9:  0 1        push 1
11:  33        halt
.
.
.

processes
#index:initial:name
0:27:interface
1:35:switcher
2:45:dimmer
3:49:user
4:55:graspAdapter
5:61:releaseAdapter
6:71:levelAdapter
.
.
```

```
locations
#index:process:invariant
27:0:6
28:0:21
29:0:42
30:0:63
31:0:84
32:0:99
35:1:237
70:6:838
71:6:847
72:6:862
.
.
edges
#process:source:target:guard:sync:
    ↪ update
0:27:28:102:114:108
0:28:27:117:126:123
0:29:30:129:141:135
0:30:27:144:153:150
0:28:29:156:177:174
0:29:32:180:201:198
0:32:31:204:216:210
0:31:27:219:228:225
1:35:36:258:279:267
1:35:37:282:309:291
1:36:35:312:321:318
1:37:35:324:333:330
2:42:44:405:414:411
2:44:43:417:441:423
.
.
6:71:72:883:901:889
6:72:70:904:916:910
.
.
expressions
#address:reads:writes:text
0:::1
6:::1
9:26::x
15:2,22::epsilon + tolerance
21:2,22,26::x <= epsilon +
    ↪ tolerance
30:26::x
36:2,23::delta + tolerance
42:2,23,26::x <= delta + tolerance
51:26::x
57:2::tolerance
63:2,26::x < tolerance
72:26::x
78:2::tolerance
84:2,26::x < tolerance
93:::1
.
.
```

Figure 3: Intermediate format example

number in a given template and its name. The cost is represents the cost of delay in certain situations or the cost of particular actions. UPPAAL supports cost annotations of the model and can do minimal cost reachability analysis [20]. The location information consists of location flag *i.e.* normal, urgent and committed. The static label somehow related to return value of the functions used in the UPTA model.

Instructions: A list of instructions used by the model. Describes how expressions are calculated, including functions used in the model. The instructions are written in a form of assembly language.

Processes: The Processes section (lines 30-38) contains information about the index, the initial location and the name of the process. For example, first entry on line 32 shows that the index of process *interface* is 0 and the index of the initial location is 27. By referring the index, it can be easily identified that index 27 is for location "idle".

Locations: Location section (lines 41-54) mainly consists of the index, the process index , the location index and the expression index for the invariant in expression section (lines 79 - onwards). For example on line 44, the invariant index is 21. By referring to expression section, we can identify that the location invariant expression is: $x <= epsilon + tolerance$.

Edges: All information about edges is present in this section in the following format :

$$process : source : target : guard : sync : update$$

Process is the index number from the processes section. Source and target value is the index number which belongs to location section. Guard, sync and update value belong to the index number in expression section.

Expression section: The last section of intermediate contains a textual representation of the expression, and lists of the variables the expression concerns. Expressions such as guards, updates and invariants in use by the model.

2.3 UPPAAL TRON

The on-line testing algorithm presented in [22] is implemented as a tool named UPPAAL TRON (Testing Real-time systems ONline) in short TRON. The IUT is attached to TRON via a test-adapter and considered as a black-box since its state cannot be directly observed (see Figure 4). Only communication events via input/output channels are visible. The user supplies TRON with the model of the IUT that consists of closed timed automata network in parallel composition with assumptions on the environment.

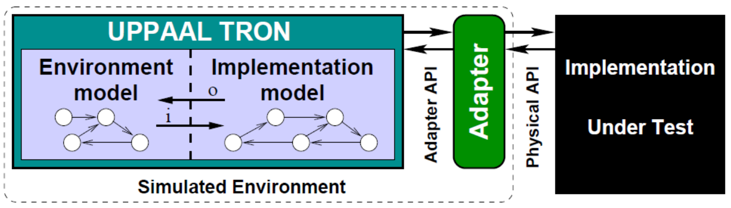


Figure 4: UPPAAL TRON framework [2]

Test primitives are generated directly form the model, executed and the system responses are checked at the same time *(on-the-fly)* while connected to IUT, thus avoiding huge intermediate test suites. During the testing, the tool follows state changes available in the model or can be driven by environment model [2].

2.3.1 Log format:

During the test session, the changing of state evolves the testing session into a new state along with clocks and variable valuation. The evolved state can be logged to investigate weather it conforms to the locations, clocks and variable valuations of the model. TRON can produce logs on different verbosity levels depending on the required information. The default level 9, contains information about the current state set before applying delay and output events, and also enables some diagnostic information when test verdict is failed or inconclusive. Compared to level 9, level 22 also contains information about events applied by the UPPAAL engine, choices considered when emulating the environment, and the current reachable states set before each update. The structure of the TRON log on verbosity level 22 is shown in Listing 1.

Processes and locations	$(p_0.l_0, p_1.l_1, ...p_{n-1}.l_m)$ where m, n is the maximum number of processes and locations respectively.
Clock Constraints	$x \sim n$ *and* $x - y \sim n$ where $n \in \mathbb{N}$, and $\sim \in \{\leq, \geq, <, >\}$, $x, y \in \mathcal{C}_0$
Variables	$var = n$ where $var \in \{variables\}$ and $n \in \mathbb{N}$

Inputs, outputs, internal events and test : The available inputs, outputs and internal choices are shown as:

$$ch@ < n : m >$$
$$where\ ch \in \mathcal{P}\{channels\},\ < \in \{[, (\},\ > \in \{),]\}\ and\ m, n \in \mathbb{R}_{\geq 0}.$$

Line 1 in Listing 1 shows the currently enabled locations, line 2 shows clock constraints, line 3 shows available inputs and the time interval until TRON will wait for input. Internal events are listed on line 4 and output events are listed on line 5. The selection of a channel or an event is shown with the clause “Choosing form”at line 6 with channel and the time interval. If TRON has to wait for some reason, the “wait for”clause shown at line 7 shows the allowed interval for which TRON can wait. On line 8, the “Chose to wait until”shows the time until TRON waited and “Test”clause at line 9 shows the time unit when Test event is received form IUT.

```
( interface.idle switcher.idle dimmer.PassiveUp user.idle graspAdapter.idle
    ↪ releaseAdapter.idle levelAdapter.idle )
interface.x<=0, interface.x-dimmer.x<=0, interface.x-graspAdapter.x<=0, interface.x-
    ↪ releaseAdapter.x<=0, interface.x-levelAdapter.x<=0, interface.x-#t<=0, dimmer.x
    ↪ <=0, dimmer.x-interface.x<=0, dimmer.x-graspAdapter.x<=0, dimmer.x-releaseAdapter
    ↪ .x<=0, dimmer.x-levelAdapter.x<=0, dimmer.x-#t<=0, graspAdapter.x<=0,
    ↪ graspAdapter.x-interface.x<=0, graspAdapter.x-dimmer.x<=0, graspAdapter.x-
    ↪ releaseAdapter.x<=0, graspAdapter.x-levelAdapter.x<=0, graspAdapter.x-#t<=0,
    ↪ releaseAdapter.x<=0, releaseAdapter.x-interface.x<=0, releaseAdapter.x-dimmer.x
    ↪ <=0, releaseAdapter.x-graspAdapter.x<=0, releaseAdapter.x-levelAdapter.x<=0,
    ↪ releaseAdapter.x-#t<=0, levelAdapter.x<=0, levelAdapter.x-interface.x<=0,
    ↪ levelAdapter.x-dimmer.x<=0, levelAdapter.x-graspAdapter.x<=0, levelAdapter.x-
    ↪ releaseAdapter.x<=0, levelAdapter.x-#t<=0, #t<=0, #t-interface.x<=0, #t-dimmer.x
    ↪ <=0, #t-graspAdapter.x<=0, #t-releaseAdapter.x<=0, #t-levelAdapter.x<=0 on=0
    ↪ iutLevel=0 OL=0 envLevel=0 levelAdapter.data=0
Inps: grasp@[0;+inf)
Ints: (empty)
Outs: (empty)
Choosing from inputs: grasp@[0;+inf)
Wait for [0;+inf)
Chose to wait until 43078us
TEST: grasp()@[50000us;50000us] at [5;5] on 1
```

Listing 1: TRON log example

3 Approach

For back-traceability and debugging purposes, we developed an approach to convert the log produced by TRON into a simulation trace for UPPAAL. The approach is not trivial, since it requires the reconstruction of the state space in order to identify the timed transitions information and calculating the next symbolic state of the simulation trace. As presented in Section 2.1, during the test execution, an UPTA can evolve by taking discrete or time transitions. Both types of transitions are encountered in the TRON log. In contrast, in the symbolic execution done in UPPAAL, the time transitions are hidden. The transformation (depicted in Algorithm 1) takes as input a TRON log file (on verbosity level 22) and applied the following steps which are discussed later in this section.

- Identify active (available) locations.
- Extract clock information from constraints.
- Re-calculate clock constraints.
- Extract variable values.
- Calculate next transition(s) taken by TRON.

3.1 Identify active (available) locations

The first entry in state set contains information about the processes and locations as shown in Listing 1. In order to calculate location vector of the trace, state-set processes and their locations are searched in the *layout* section of intermediate format as described in Subsection 2.2.2. When the ordinal position of the state-set process entry corresponds to the process index in the intermediate format, the location index is then resolved by collecting all locations of the process and by assigning them value from 1 to n. For

Algorithm 1 Transformation

```
1: procedure TRANFORM(Log)
2:     $\sigma := \emptyset$
3:     for each $s \in$ $Log$ do                                → for each state-set in log
4:         $\sigma := \sigma \cup \{\rho(s)\}$                   → parse log, extract state-set
5:     $\Gamma := \emptyset$
6:     for each $s = \{l \in \mu, z \in \Psi, v \in \nu, t \in \tau\} \in \sigma$ do
7:         $\Gamma := \Gamma \cup \{s\}$
8:     $T := \emptyset$        → $Traceset$
9:     for each $(s, s') \in \Gamma$ do
10:        if $(\mu \in s \neq \mu \in s') \vee$ (TEST $event\ has\ some\ channel(ch)$) then
11:            $t := \emptyset$
12:            $t := t \cup \{s'\}$                              → Add location Vector to trace
13:            $t := t \cup \{calculateZone(z)\}$
14:            $t := t \cup \{v\}$
15:            $t := t \cup \{TransitionVector(s, s', ch)\}$
16:            $T := T \cup t$
17:        else (TEST : delay to)
18:            $Skip$
19:    for each $t \in T$ do
20:        $write\ t$
21: function CALCULATEZONE($z$)
22:    $\psi := \emptyset, l := \emptyset, u := \emptyset$
23:    for each $(x > n) \in z$ do                               → $where\ x \in \mathcal{C}_0\ and\ n \in \mathbb{N}$
24:        $l := l \cup \{x > n\}$
25:    for each $(x < n) \in z$ do                               → $where\ x \in \mathcal{C}_0\ and\ n \in \mathbb{N}$
26:        $u := u \cup \{x < n\}$
27:    for each $x \in l$ do                                     → $where\ x \in \mathcal{C}_0\ and\ n \in \mathbb{N}$
28:        for each $y \in u$ do
29:            if $(x < y)$ then
30:                $\psi \cup \{(x - y) \sim n\}$                → $where\ x, y \in \mathcal{C}_0\ and\ n = \{x - y : x, y \in u, n \in \mathbb{N}\}, \sim \in \{<, \leq\}$
31:            if $(x > y)$ then
32:                $\psi \cup \{(x - y) \sim n\}$                → $where\ x, y \in \mathcal{C}_0\ and\ n = \{x - y : x, y \in l, n \in \mathbb{N}\}, \sim \in \{<, \leq\}$
33:    return $(\psi)$
34: function TRANSITIONVECTOR($s_s, s_t, ch$)
35:    $t := \emptyset$
36:    for each $e = \{s, L, \psi, \rho, s'\}$ do
37:        if $(s_s = s \wedge s_t = s')$ then
38:            $t := t \cup e$
39:        else($ch \in \mathcal{L}$)
40:            $t := t \cup \{e\}$
41:    return $(t)$
```

example, line 26 in Figure 5 contains *27:0:6*. The first value (*i.e.* 27) identifies the global index for location (*i.e.* 27:location::idle). The next value is the process index (*i.e.* 0:27:interface) and the last number identifies the invariant global index (*i.e.* 6). The entry *27:0:6* in location section will results in 1, because *idle* is the first location in the layout section of *IF* file. Subsequently, *ignoring, alert* and *touched* correspond to location 2,3 and 4 respectively in the *interface* process.

```
1   layout
2   #index:const:value
3   #index:clock:nr:name
4   #index:var:min:max:init:nr:name
5   #index:meta:min:max:init:nr:name
6   #index:cost
7   #index:location:flags:name
8   #index:static:min:max:name
9   26:clock:1:interface.x
10  27:location::idle
11  28:location::ignoring
12  29:location::alert
13  30:location::touched
14  .
15  .
16  .
17  .
18
19
```

```
20  processes
21  #index:initial:name
22  0:27:interface
23  1:35:switcher
24  .
25  .
26  .
27  locations
28  #index:process:invariant
29  27:0:6
30  28:0:21
31  29:0:42
32  30:0:63
33  31:0:84
34  32:0:99
35  35:1:237
36  .
37  .
38
```

Figure 5: Layout, processes and locations section

3.2 Extracting clock information and calculating clock constraints

Location information is followed by clock constraints, which are in the form of zones as described in Subsection 2.1.

```
interface.x>=50
graspAdapter.x>=50
releaseAdapter.x>105
#t>105

interface.x<=60
interface.x-dimmer.x<=55
interface.x-graspAdapter.x<=0
interface.x-releaseAdapter.x<-55
interface.x-levelAdapter.x<=55
interface.x-#t<-55

dimmer.x<=5
dimmer.x-interface.x<=-50
dimmer.x-graspAdapter.x<=-50
dimmer.x-releaseAdapter.x<-105
dimmer.x-levelAdapter.x<=0
dimmer.x-#t<-105

graspAdapter.x<=65
graspAdapter.x-interface.x<=5
graspAdapter.x-dimmer.x<=60
graspAdapter.x-releaseAdapter.x<-55
graspAdapter.x-levelAdapter.x<=60
graspAdapter.x-#t<-55

releaseAdapter.x<121
releaseAdapter.x-interface.x<61
releaseAdapter.x-dimmer.x<116
releaseAdapter.x-graspAdapter.x<56
releaseAdapter.x-levelAdapter.x<116
releaseAdapter.x-#t<=0

levelAdapter.x<=5
levelAdapter.x-interface.x<=-50
levelAdapter.x-dimmer.x<=0
levelAdapter.x-graspAdapter.x<=-50
levelAdapter.x-releaseAdapter.x<-105
levelAdapter.x-#t<-105

#t<121
#t-interface.x<61
#t-dimmer.x<116
#t-graspAdapter.x<56
#t-releaseAdapter.x<=0
#t-levelAdapter.x<116
```

Figure 6: TRON clock constraints example

Clock information: The clock information can be understood by taking the following example from Figure 6 into account. We applied some formatting to enhance the visualization.

$$\text{Block 1}\begin{cases}\text{lower-bound}\begin{cases}interface.x >= 50\\ graspAdapter.x >= 50\\ releaseAdapter.x > 105\\ \#t > 105\end{cases}\end{cases}$$

$$\text{Block 2}\begin{cases}\text{upper-bound}\begin{cases}interface.x <= 60\end{cases}\\ \text{clock difference}\begin{cases}interface.x - dimmer.x <= 55\\ interface.x - graspAdapter.x <= 0\\ interface.x - levelAdapter.x <= 55\\ interface.x - levelAdapter.x < 5\\ interface.x - \#t < -55\end{cases}\end{cases}$$

$$\vdots$$

$$\text{Block n}\begin{cases}\text{upper-bound}\begin{cases}\#t < 121\end{cases}\\ \text{clock difference}\begin{cases}\#t - interface.x < 61\\ \#t - dimmer.x < 116\\ \#t - graspAdapter.x < 56\\ \#t - releaseAdapter.x <= 0\\ \#t - levelAdapter.x < 116\end{cases}\end{cases}$$

Block 1 of the clock information contains lower limits of the clocks. The clocks present in this block are the only clock which are active and unequal as proposed in [11]. The upper-bound of the clock is available in the beginning of corresponding block. By using these clock constraints from Figure 6, we can populate a DBM like matrix to visualize the clock constraints.

$$\begin{array}{c|cccccc|}
 & t & interface & dimmer & graspAdapter & releaseAdapter & levelAdapter \\
t & (105,121) & 61) & 116) & 56) & 0] & 116) \\
interface & (-55 & [50,60] & 55] & 0] & -55) & 55] \\
dimmer & [5 & [-50 & ?,5] & -50] & -105) & 55] \\
graspAdapter & (-55 & [5 & [60 & [50,65] & -55) & 60] \\
releaseAdapter & [0 & (61 & (116 & (56 & (105,121) & 116) \\
levelAdapter & (-105 & [-50 & [0 & [-50 & (-105 & ?,5]
\end{array}$$

Figure 7: DBM like structure containing clock constraints

3.3 Calculating clock constraints

Clock reduction for UPPAAL has been originally discussed in [11]. The original approach used an iterative algorithm to reduce the number of clocks in the timed automata. In our approach, we used the information present in the TRON log to calculate clock constraints *zones* of a state-set which consist of active and unequal clocks. The resulting clock constraints are in the format which is used by UPPAAL simulator to store trace. In our approach, we only consider those clocks which have both upper and lower bonds (see Figure 7) due to their *boundedness* property. The boundedness of the i^{th}clock can be determined by observing the lower and upper bound at the intersection of the i^{th} row and the i^{th} column. Additionally, due to boundedness property of the clocks, one can calculate the entire matrix by subtracting the corresponding bounds of the clocks.

For example, the matrix elements in Figure 7 shows the clock constraints extracted from Figure 6. Neither *dimmer* nor *levelAdapter* has the lower-bound available in the TRON log entry (see Figure 7). Due to infiniteness, both of these clocks were excluded during the transformation which indicates that the clocks are not active or that they are equal to some other clock. Additionally, the entries above the diagonal elements *i.e, the upper-bounds*, can be calculated by subtracting the lower-bound of clocks. Similarly, the elements below diagonal of the matrix *i.e, lower-bound* can be calculated by subtracting the upper-bound of the clocks.

For example, to calculate the matrix entries of $\#t - interface.x$ and $interface.x - \#t$, clock t = (50, 121) and clock *interface.x* =[50,60]. By subtracting the upper-bounds of the clocks, we can obtain the upper-bound of interface *i.e.* $121 - 60 = 21$ and similarly, by subtracting lower-bounds, we can obtain the lower-bound element of the matrix *i.e.* $50 - 105 = -55$.

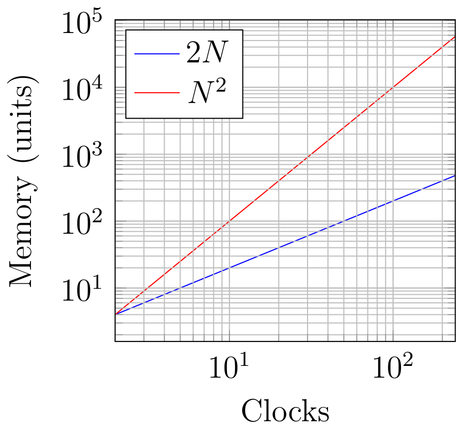


Figure 8: Memory required to store a zone of N clocks

Our approach reduced the memory usage for calculating the state-set reachability by using the lower bond and the upper bond of a clock. The memory required to store the clocks information is twice as the number of clocks present in a zone *i.e,* for N clocks only $2N$ units of memory are required. Moreover, the original approach in DBM requires N^2 units of memory to store the clock information. However, our approach requires $N^2 - 2N$ units of memory for a zone composed of N clocks. Equation 1 computes the overall memory reduction $M_{(red)}$ for n zones. Figure 8 shows the memory required for both N^2 and $2N$ where N includes all the clocks including the reference clock $\mathbf{0}$.

$$M_{(red)} = \sum_{z=1}^{n} N_z^2 - 2N_z \tag{1}$$

On-the-fly (run-time) calculation of the clock constraints may result in extra CPU usage and increased over-all turn-around time.

3.4 Constructing variable vector

The UPPAAL simulation trace also contains values for discrete variables [2] for a given symbolic state. Extracting variable values from TRON log is a straight forward process. The variable values are in the form of $x = n\ where\ x \in \{variables\}\ and\ n \in \mathbb{N}$. The ordinal position of variable in TRON log corresponds to the variable position in the UPPAAL simulation trace. For example, Listing 1 contains variables and their values *i.e,* $on = 0$ $iutLevel = 0$ $OL = 0$ $envLevel = 0$ $levelAdapter.data = 0$. The corresponding vector for UPPAAL simulation trace will be as follow:

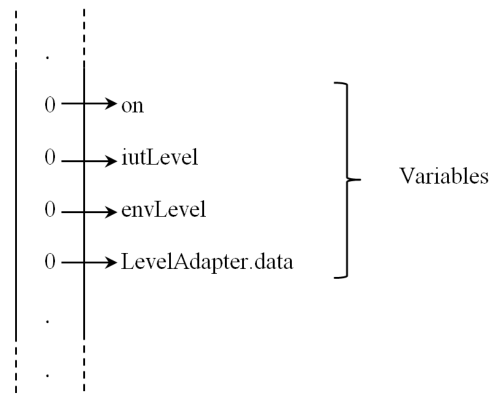


Figure 9: Variable vector with corresponding variable names (see Figure 2)

3.5 Calculate next transition(s)

The last part of the UPPAAL simulation trace for a given state consists of transitions taken by simulator. Each transition consists of a process index and a edge number which was taken to evolve to the current state. In the TRON log, we encountered two variants of transitions namely *time transition* and *discrete transitions*. We also classified discrete transition into three categories *i.e, discrete transition without synchronization*, *discrete transition with synchronization*, and *discrete transitions with select*.

3.5.1 Time Transition (delay):

It is possible that TRON is applying delay at some state-set and does not take any transition (see Sec.2.1). However, whenever TRON takes *time transitions*, the log entry has the following format: *TEST : delay to ⟨time unit ⟩on ⟨number of states⟩*. This implies that no transition has been taken and delay applied to the last n number of states.

3.5.2 Discrete transition without synchronization:

These transition are taken when an edge does not have a synchronization, but it is enabled by its guard and the invariant conditions of the destination location. These transitions can be identified by observing state location vector. The evolved state location vector has at least one location different compared to the previous location vector. The edge connecting the source and target locations can be identified by querying the intermediate format file. One limitation of our approach is that it is able to identify self-looping edges without synchronization. Except this limitation, the approach works well because of the clock and variable values are calculated and extracted separately. In case of the modification of variables and/or clock resets occurred by the self-looping edge, the calculated clock values and the extracted variable vector rectifies the discrepancies.

3.5.3 Discrete transitions with synchronization:

The edges labelled with synchronization can be easily identified by the label present in *Inps, Ints, Outs* clauses in the TRON log. The synchronization might be an input event, an internal event or a output event. The *Choosing from :* clause identifies the synchronizations (channels) that were used to evolve to the given state.

3.5.4 Discrete transition with select:

The *select* operator implements the non-deterministic assignment *i.e.* in order to provide a non-deterministic way to select a value from the set of values. The non-deterministic behaviour of select results in additional edge for each value in the set.

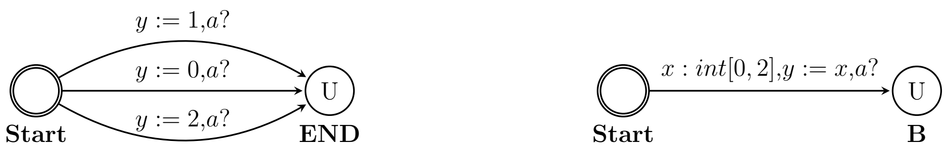


Figure 10: Edges generated by TRON for every value in select (left) and corresponding UPPAAL model with *select* (right)

Figure 10 (left) shows how TRON generates edges for every value for select in symbolic simulator and Figure 10 (right) shows how UPPAAL internally interprets the select operator in the symbolic simulation.

4 Tool support

In order to automate the proposed transformation approach, we have developed a tool called “Back-Tracer”in Java. The tool uses similar approach as *tracer* utility in *Uppaal Timed Automata Parser Library* [9] for parsing the UPTA model. The transformation process and the artefacts used by the tool are shown in Figure 11. The tool take as an input : a TRON log file, an intermediate format file, UPTA model, and a sample trace file.

The process of transformation consist of following steps:

1. Run a test session with TRON and obtain a log file.

2. Generate *if* file from the UPTA model by using *verifyta* (it only has to be done once per model, as long as the model does not change).

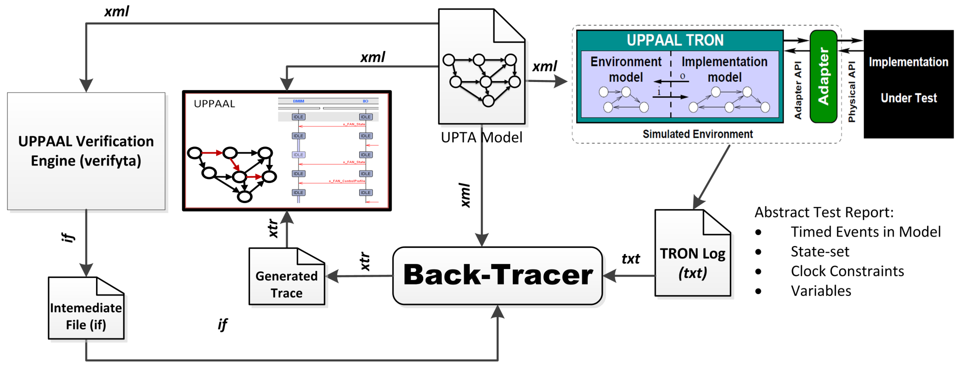


Figure 11: Back Tracer setup

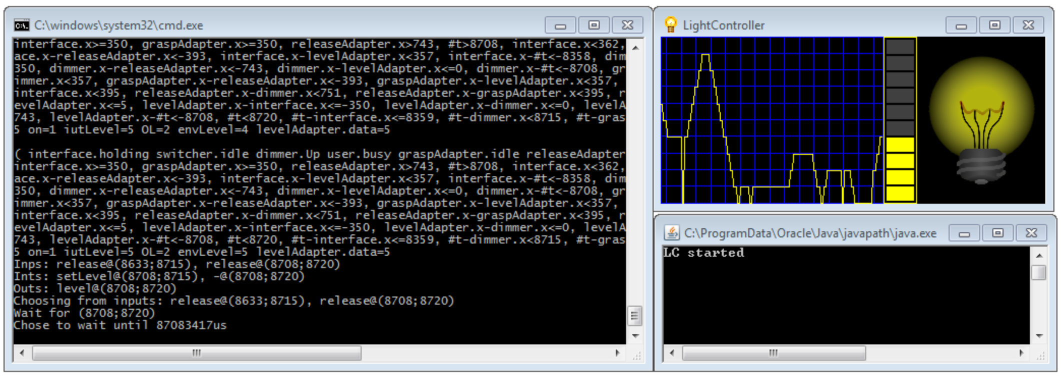
3. Transform TRON log to UPPAAL simulator trace by executing **Back-Tracer** with required parameters (i.e. an intermediate format file *(if)*, an UPTA model *(xml)* and a TRON log).

The intermediate format (*if*) contains the information about the UPTA model as described in Subsection 2.2.2. The user must generate the *if* file initially and after any modification in UPTA model (see [9] for how to generate *if file*). The UPTA model is required for parsing the model and identifying the indexes of the processes, locations, edges, invariants, guards, and synchronizations *etc.* The TRON log is then used to construct the UPPAAL simulation trace with the help of identified indexes. The binaries of Back-Tracer tool are available at: `http://users.abo.fi/jiqbal/back-tracer/`.

5 Example and evaluation

The applicability of our approach is exemplified by using smart lamp controller available with the TRON distribution. Figure 12 shows that the example contains a Light controller simulator with light-level history, light level bar and a lamp with colour-encoded level. A console window with events at light controller side is also present at the bottom of light-level display. The light controller is an essential component of smart lamp where the level of light is changed by the time interval between consecutive user grasps and releases. The TRON is behaving like a smart lamp user by issuing grasps and releases events. At the same time it observes that the level of light of the lamp is correct according to the lamp specification. We modified the start-up command so that TRON produces test logs with the verbosity level 22 (-v22).

When a simulation is terminated, we used our BackTracer tool to transform the TRON log into a UPPAAL simulation trace. The obtained trace is then loaded to UPPAAL simulator to be analysed. Figure 13 shows a



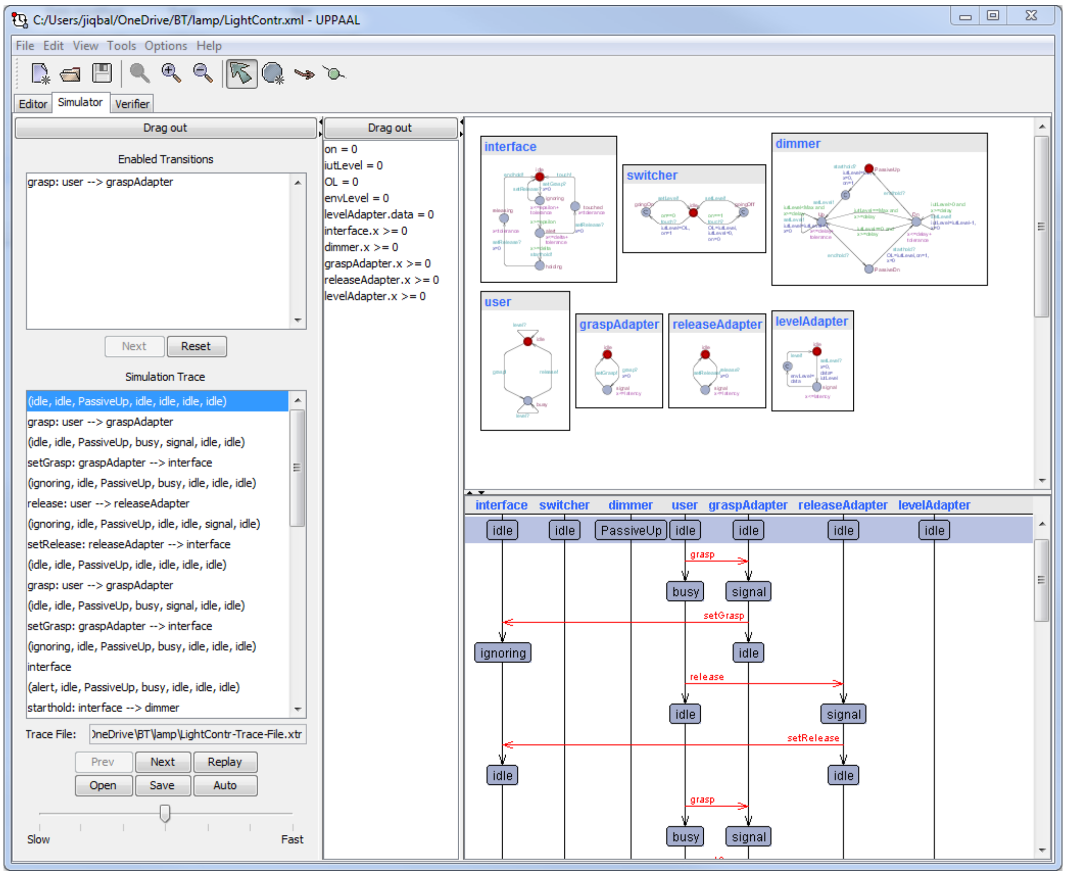
```
interface.x>=350, graspAdapter.x>=350, releaseAdapter.x>743, #t>8708, interface.x<362,
ace.x-releaseAdapter.x<-393, interface.x-levelAdapter.x<357, interface.x-#t<-8358, dim
350, dimmer.x-releaseAdapter.x<-743, dimmer.x-levelAdapter.x<=0, dimmer.x-#t<-8708, gr
immer.x<357, graspAdapter.x-releaseAdapter.x<-393, graspAdapter.x-levelAdapter.x<357,
interface.x<395, releaseAdapter.x-dimmer.x<751, releaseAdapter.x-graspAdapter.x<395, r
evelAdapter.x<=5, levelAdapter.x-interface.x<=-350, levelAdapter.x-dimmer.x<=0, levelA
743, levelAdapter.x-#t<-8708, #t<8720, #t-interface.x<=8359, #t-dimmer.x<8715, #t-gras
5 on=1 iutLevel=5 OL=2 envLevel=4 levelAdapter.data=5

( interface.holding switcher.idle dimmer.Up user.busy graspAdapter.idle releaseAdapter
interface.x>=350, graspAdapter.x>=350, releaseAdapter.x>743, #t>8708, interface.x<362,
ace.x-releaseAdapter.x<-393, interface.x-levelAdapter.x<357, interface.x-#t<-8358, dim
350, dimmer.x-releaseAdapter.x<-743, dimmer.x-levelAdapter.x<=0, dimmer.x-#t<-8708, gr
immer.x<357, graspAdapter.x-releaseAdapter.x<-393, graspAdapter.x-levelAdapter.x<357,
interface.x<395, releaseAdapter.x-dimmer.x<751, releaseAdapter.x-graspAdapter.x<395, r
evelAdapter.x<=5, levelAdapter.x-interface.x<=-350, levelAdapter.x-dimmer.x<=0, levelA
743, levelAdapter.x-#t<-8708, #t<8720, #t-interface.x<=8359, #t-dimmer.x<8715, #t-gras
5 on=1 iutLevel=5 OL=2 envLevel=5 levelAdapter.data=5
Inps: release@(8633;8715), release@(8708;8720)
Ints: setLevel@(8708;8715), -@(8708;8720)
Outs: level@(8708;8720)
Choosing from inputs: release@(8633;8715), release@(8708;8720)
Wait for (8708;8720)
Chose to wait until 87083417us
```

```
LC started
```

C:\windows\system32\cmd.exe
LightController
C:\ProgramData\Oracle\Java\javapath\java.exe

Figure 12: TRON test Session with Smart lamp

trace loaded successfully in UPPAAL simulator to visualize the test execution trace, locations, edges and the synchronization between processes in a graphical manner.



C:/Users/jiqbal/OneDrive/BT/lamp/LightContr.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator Verifier
Drag out
Enabled Transitions
Next Reset
Simulation Trace
Trace File: OneDrive\BT\lamp\LightContr-Trace-File.xtr
Prev Next Replay
Open Save Auto
Slow Fast
interface switcher dimmer user graspAdapter releaseAdapter levelAdapter

```
grasp: user --> graspAdapter
```

```
on = 0
iutLevel = 0
OL = 0
envLevel = 0
levelAdapter.data = 0
interface.x >= 0
dimmer.x >= 0
graspAdapter.x >= 0
releaseAdapter.x >= 0
levelAdapter.x >= 0
```

```
(idle, idle, PassiveUp, idle, idle, idle, idle)
grasp: user --> graspAdapter
(idle, idle, PassiveUp, busy, signal, idle, idle)
setGrasp: graspAdapter --> interface
(ignoring, idle, PassiveUp, busy, idle, idle, idle)
release: user --> releaseAdapter
(ignoring, idle, PassiveUp, idle, idle, signal, idle)
setRelease: releaseAdapter --> interface
(idle, idle, PassiveUp, idle, idle, idle, idle)
grasp: user --> graspAdapter
(idle, idle, PassiveUp, busy, signal, idle, idle)
setGrasp: graspAdapter --> interface
(ignoring, idle, PassiveUp, busy, idle, idle, idle)
interface
(alert, idle, PassiveUp, busy, idle, idle, idle)
starthold: interface --> dimmer
```

Figure 13: UPPAAL simulator after loading trace file obtained by Backtracer

5.1 Evaluation of tool support

During the development of the *Back-tracer* tool, we encountered many challenges including calculating the timing constraints and reducing the non-active and equal clocks. The traditional approach [12] would imply the reconstruction of the entire symbolic state space. However, since we are interested in only a specific trace through the symbolic state space, we adopted a simplified approach which only calculates the timing constraints. This allows us to reduce the effort and time needed to implement our tool. Another challenge was to understand the trace format of the UPPAAL simulator in the absence of proper documentation.

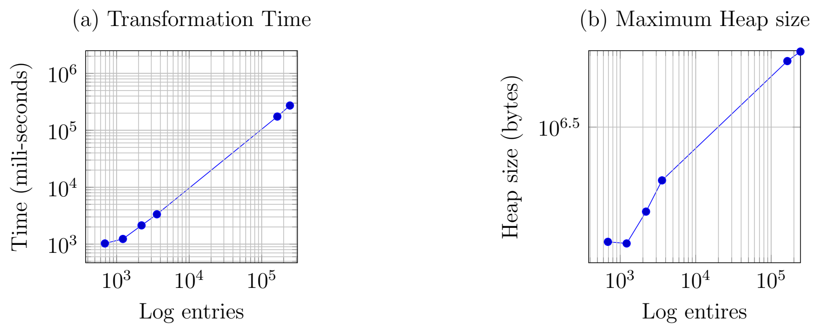


Figure 14: Graphs showing (a)Transformation time, (b) Maximum Heap size

With respect to our smart lamp controller example, the SUT and the environment consisted of seven timed automata. To evaluate our tool, we used the provided Java implementation of the lamp. We ran the on-line testing session multiple times with different length of the test session, in order to obtain log files with different lengths. We used these log files for evaluating the time, CPU utilization and memory used by the tool. The VisualVM profiler available on-line at [8] has been used for this purpose. For time calculation, we used the Java *time* library to measure the transformation time. Figures 14 (a) and (b) show that the transformation time and memory used varies proportionally with the TRON log size. Further optimizations can be applied to the tool in order to reduce the memory usage during the transformation process which is beyond the scope of this paper. This shows that the approach has the potential to scale up for larger log files.

6 Conclusions

The proposed approach has been deployed in transformation from TRON log to UPPAAL simulator trace, with the purpose of analysing external behaviour of SUT against formal specifications and visualizing the trace during the test session. The proof of concepts and applicability of our transformation tool have been demonstrated in an example called smart lamp controller.

The research study has faced several practical challenges in the transformation step of the approach. First challenge was to interpret and abstract the execution log of TRON tool with different levels of verbosity. Secondly, the selection of log verbosity level which provide adequate information for transformation process. Thirdly, how to deal with select operator when it was introduce in the model, which increased number of edges from source to target location. Another important challenge was to calculate the clock constraints based on the information present in the log. The solution to that challenge was to apply our proposed simplified approach to identify and calculate clock constrains instead of applying the clock reduction algorithm presented in [11].

Our proposed approach has successful transformed the TRON execution to log UPPAAL simulation trace. Apparently, the proposed approach provide a sophisticate tool support to analyse and visualize the black-box (conformance) testing trace in UPPAAL which was difficult to understand in long lasting test sessions and inconclusive or failed test results.

References

[1] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, PedroR. DArgenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, KimG. Larsen, M.Oliver Mller, Paul Pettersson, Carsten Weise, and Wang Yi. Uppaal - now, next, and future. In Franck Cassez, Claude Jard, Brigitte Rozoy, and MarkDermot Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 99–124. Springer Berlin Heidelberg, 2001.

[2] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004. online; last accessed : 18.05.2015.

[3] Axel Belinfante, Jan Feenstra, RenG. de Vries, Jan Tretmans, Nicolae Goga, Loe Feijs, Sjouke Mauw, and Lex Heerink. Formal test automation: A simple experiment. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Testing of Communicating Systems*, volume 21 of *IFIP The International Federation for Information Processing*, pages 179–196. Springer US, 1999.

[4] Johan Bengtsson. *Clocks, dbms and states in timed systems.* Acta Universitatis Upsaliensis, 2002.

[5] Victor Braberman, Miguel Felder, and Martina Marr. Testing timing behavior of real-time software. In *In International Software Quality Week*, 1997.

[6] Rachel Cardell-oliver and Tim Glover. A practical and complete algorithm for testing real-time systems. In *In Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 1486*, pages 251–261. Springer-Verlag, 1998.

[7] D. Clarke and Insup Lee. Automatic test generation for the analysis of a real-time system: Case study. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 112–124, Jun 1997.

[8] Oracle Corporation. Visualvm. `http://visualvm.java.net/`, 2015. [Online; accessed 13-May-2015].

[9] Alexandre David. Uppaal timed automata parser library. `http://people.cs.aau.dk/~adavid/utap/`. [Online; accessed on 11-March-2015].

[10] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 66–75, Dec 1995.

[11] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 73–81, Dec 1996.

[12] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 313–329, London, UK, UK, 1998. Springer-Verlag.

[13] Evert de Kock. Control and performance analysis of wafer flow in wafer handling systems. Master's thesis, Eindhoven University of Technology, 2015. [Online at `https://www.tue.nl/fileadmin/content/faculteiten/wtb/Onderzoek/Onderzoeksgroepen/Manufacturing_Networks/MN_Pdfs/MSc/2014_MN420766_Kock_EGJ_de_afstudeerverslag.pdf` ;accessed 10-June-2015].

[14] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer Berlin Heidelberg, 1990.

[15] Jean-Claude Fernandez, Claude Jard, Thierry Jron, and Csar Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and ThomasA. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer Berlin Heidelberg, 1996.

[16] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 394–406, Jun 1992.

[17] Anders Hessel, Kim G Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using uppaal. In *Formal Approaches to Software Testing*, pages 114–130. Springer, 2004.

[18] Thierry Jéron. Symbolic model-based test selection. *Electronic Notes in Theoretical Computer Science*, 240(0):167 – 184, 2009. Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008).

[19] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 109–126. Springer Berlin Heidelberg, 2004.

[20] Kim Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Effcient cost-optimal reachability for priced timed automata. In Grard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505. Springer Berlin Heidelberg, 2001.

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

[22] KimG. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin Heidelberg, 2005.

[23] M. Mikucionis, Kim G. Larsen, and B. Nielsen. T-uppaal: online model-based testing of real-time systems. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 396–397, Sept 2004.

[24] Marius Mikuĉionis. Uppaal tron. `http://people.cs.aau.dk/~marius/tron/`, 2012. [Online; accessed 20-April-2015].

[25] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[26] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012. online; last accessed : 26.05.2015.

ISBN 978-952-12-3243-5
ISSN 1239-1891