A Tutorial on Uppaal 4.0

Updated November 28, 2006

Gerd Behrmann, Alexandre David, and Kim G. Larsen

Department of Computer Science, Aalborg University, Denmark
{behrmann,adavid,kgl}@cs.auc.dk.

Abstract. This is a tutorial paper on the tool Uppaal. Its goal is to be a short introduction on the flavour of timed automata implemented in the tool, to present its interface, and to explain how to use the tool. The contribution of the paper is to provide reference examples and modelling patterns.

1 Introduction

Uppaal is a toolbox for verification of real-time systems jointly developed by Uppsala University and Aalborg University. It has been applied successfully in case studies ranging from communication protocols to multimedia applications [35,55,24,23,34,43,54,44,30]. The tool is designed to verify systems that can be modelled as networks of timed automata extended with integer variables, structured data types, user defined functions, and channel synchronisation.

The first version of Uppaal was released in 1995 [52]. Since then it has been in constant development [21,5,13,10,26,27]. Experiments and improvements include data structures [53], partial order reduction [20], a distributed version of Uppaal [17,9], guided and minimal cost reachability [15,51,16], work on UML Statecharts [29], acceleration techniques [38], and new data structures and memory reductions [18,14]. Version 4.0 [12] brings symmetry reduction [36], the generalised sweep-line method [49], new abstraction techniques [11], priorities [28], and user defined functions to the mainstream. Uppaal has also generated related Ph.D. theses [50,57,45,56,19,25,32,8,31]. It features a Java user interface and a verification engine written in C++ . It is freely available at `http://www.uppaal.com/`.

This tutorial covers networks of timed automata and the flavour of timed automata used in Uppaal in section 2. The tool itself is described in section 3, and three extensive examples are covered in sections 4, 5, and 6. Finally, section 7 introduces common modelling patterns often used with Uppaal.

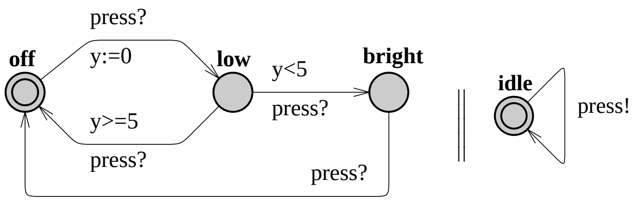
2 Timed Automata in Uppaal

The model-checker Uppaal is based on the theory of timed automata [4] (see [42] for automata theory) and its modelling language offers additional features such as bounded integer variables and urgency. The query language of Uppaal, used to specify properties to be checked, is a subset of TCTL (timed computation tree logic) [39,3]. In this section we present the modelling and the query languages of UPPAAL and we give an intuitive explanation of time in timed automata.

2.1 The Modelling Language

Networks of Timed Automata A timed automaton is a finite-state machine extended with clock variables. It uses a dense-time model where a clock variable evaluates to a real number. All the clocks progress synchronously. In UPPAAL, a system is modelled as a network of several such timed automata in parallel. The model is further extended with bounded discrete variables that are part of the state. These variables are used as in programming languages: They are read, written, and are subject to common arithmetic operations. A state of the system is defined by the locations of all automata, the clock values, and the values of the discrete variables. Every automaton may fire an edge (sometimes misleadingly called a transition) separately or synchronise with another automaton[1], which leads to a new state.

Figure 1(a) shows a timed automaton modelling a simple lamp. The lamp has three locations: `off`, `low`, and `bright`. If the user presses a button, i.e., synchronises with `press?`, then the lamp is turned on. If the user presses the button again, the lamp is turned off. However, if the user is fast and rapidly presses the button twice, the lamp is turned on and becomes bright. The user model is shown in Fig. 1(b). The user can press the button randomly at any time or even not press the button at all. The clock y of the lamp is used to detect if the user was fast ($y < 5$) or slow ($y >= 5$).



(a) Lamp. (b) User.

Fig. 1. The simple lamp example.

We give the basic definitions of the syntax and semantics for the basic timed automata. In the following we will skip the richer flavour of timed automata supported in UPPAAL, i.e., with integer variables and the extensions of urgent and committed locations. For additional information, please refer to the help

[1] or several automata in case of broadcast synchronisation, another extension of timed automata in UPPAAL.

menu inside the tool. We use the following notations: C is a set of clocks and $B(C)$ is the set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. A timed automaton is a finite directed graph annotated with conditions over and resets of non-negative real valued clocks.

Definition 1 (Timed Automaton (TA)). *A timed automaton is a tuple (L, l_0, C, A, E, I), where L is a set of locations, $l_0 \in L$ is the initial location, C is the set of clocks, A is a set of actions, co-actions and the internal τ-action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and $I : L \to B(C)$ assigns invariants to locations.* □

In the previous example on Fig. 1, `y:=0` is the reset of the clock y, and the labels `press?` and `press!` denote action–co-action (channel synchronisations here).

We now define the semantics of a timed automaton. A clock valuation is a function $u : C \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. Let $\mathbb{R}^C$ be the set of all clock valuations. Let $u_0(x) = 0$ for all $x \in C$. We will abuse the notation by considering guards and invariants as sets of clock valuations, writing $u \in I(l)$ to mean that u satisfies $I(l)$.

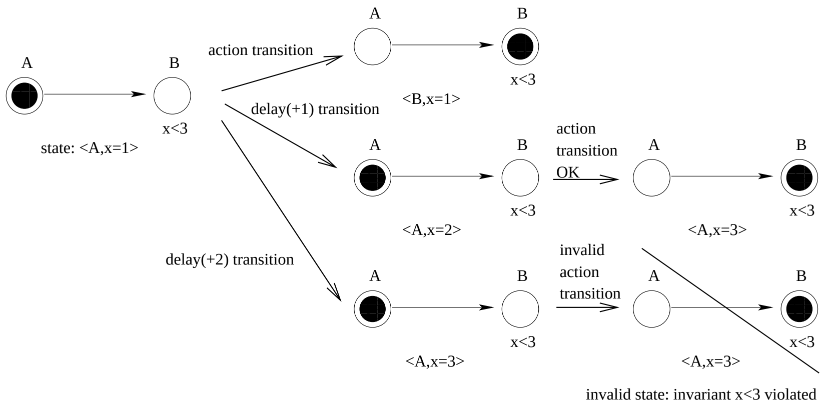


Fig. 2. Semantics of TA: different transitions from a given initial state.

Definition 2 (Semantics of TA). *Let (L, l_0, C, A, E, I) be a timed automaton. The semantics is defined as a labelled transition system $\langle S, s_0, \to \rangle$, where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\to \subseteq S \times (\mathbb{R}_{\geq 0} \cup A) \times S$ is the transition relation such that:*

- $(l, u) \xrightarrow{d} (l, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(l)$, *and*
- $(l, u) \xrightarrow{a} (l', u')$ *if there exists* $e = (l, a, g, r, l') \in E$ *s.t.* $u \in g$, $u' = [r \mapsto 0]u$, *and* $u' \in I(l')$,

where for $d \in \mathbb{R}_{\geq 0}$, $u + d$ maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $C \setminus r$. □

Figure 2 illustrates the semantics of TA. From a given initial state, we can choose to take an *action* or a *delay* transition (different values here). Depending of the chosen delay, further actions may be forbidden.

Timed automata are often composed into a *network of timed automata* over a common set of clocks and actions, consisting of n timed automata $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, I_i)$, $1 \leq i \leq n$. A location vector is a vector $\bar{l} = (l_1, \ldots, l_n)$. We compose the invariant functions into a common function over location vectors $I(\bar{l}) = \wedge_i I_i(l_i)$. We write $\bar{l}[l_i'/l_i]$ to denote the vector where the ith element l_i of $\bar{l}$ is replaced by l_i'. In the following we define the semantics of a network of timed automata.

Definition 3 (Semantics of a network of Timed Automata). *Let $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, I_i)$ be a network of n timed automata. Let $\bar{l}_0 = (l_1^0, \ldots, l_n^0)$ be the initial location vector. The semantics is defined as a transition system $\langle S, s_0, \rightarrow \rangle$, where $S = (L_1 \times \cdots \times L_n) \times \mathbb{R}^C$ is the set of states, $s_0 = (\bar{l}_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation defined by:*

- $(\bar{l}, u) \xrightarrow{d} (\bar{l}, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(\bar{l})$.
- $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l_i'/l_i], u')$ *if there exists* $l_i \xrightarrow{\tau g r} l_i'$ *s.t.* $u \in g$, $u' = [r \mapsto 0]u$ *and* $u' \in I(\bar{l}[l_i'/l_i])$.
- $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l_j'/l_j, l_i'/l_i], u')$ *if there exist* $l_i \xrightarrow{c?g_i r_i} l_i'$ *and* $l_j \xrightarrow{c!g_j r_j} l_j'$ *s.t.* $u \in (g_i \wedge g_j)$, $u' = [r_i \cup r_j \mapsto 0]u$ *and* $u' \in I(\bar{l}[l_j'/l_j, l_i'/l_i])$. □

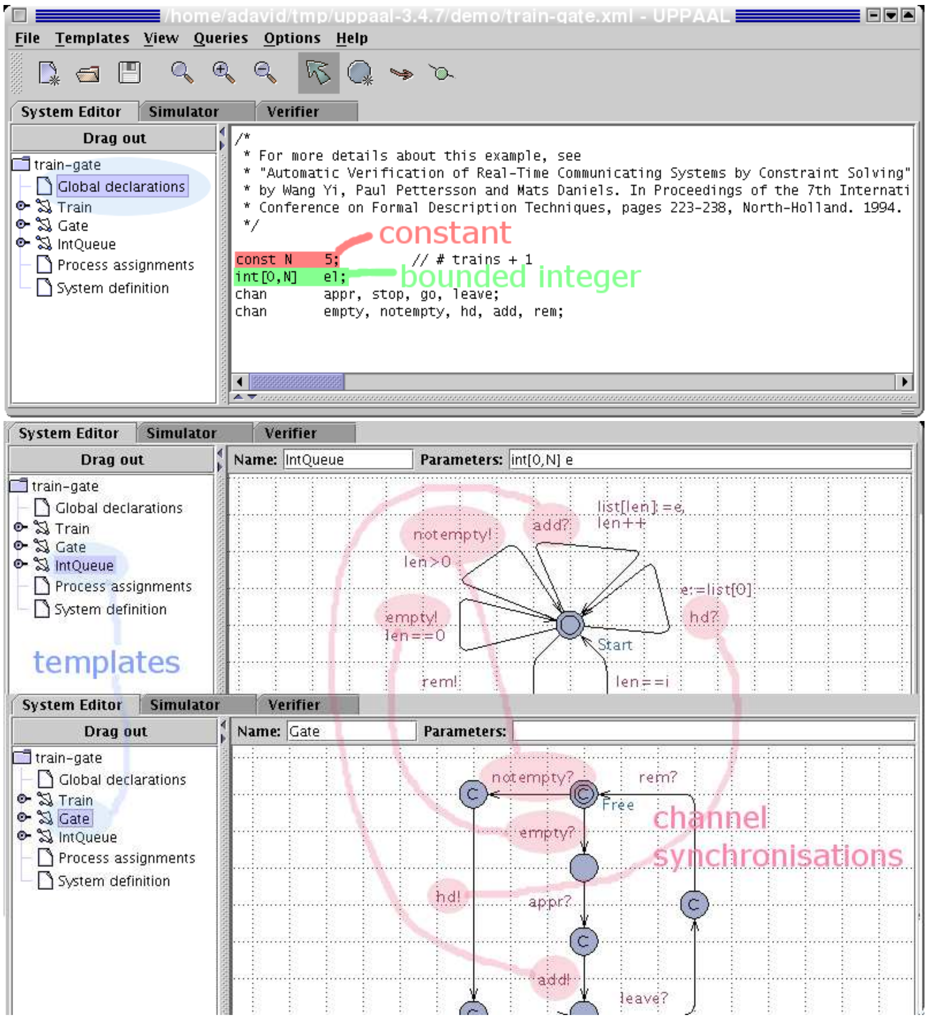
As an example of the semantics, the lamp in Fig. 1 may have the following states (we skip the user): $(\texttt{Lamp.off}, y = 0) \rightarrow (\texttt{Lamp.off}, y = 3) \rightarrow (\texttt{Lamp.low}, y = 0) \rightarrow (\texttt{Lamp.low}, y = 0.5) \rightarrow (\texttt{Lamp.bright}, y = 0.5) \rightarrow (\texttt{Lamp.bright}, y = 1000) \ldots$

Timed Automata in Uppaal The UPPAAL modelling language extends timed automata with the following additional features (see Fig. 3:

Templates automata are defined with a set of parameters that can be of any type (e.g., `int`, `chan`). These parameters are substituted for a given argument in the process declaration.

Constants are declared as `const name value`. Constants by definition cannot be modified and must have an integer value.

Bounded integer variables are declared as `int[min,max] name`, where `min` and `max` are the lower and upper bound, respectively. Guards, invariants, and assignments may contain expressions ranging over bounded integer variables. The bounds are checked upon verification and violating a bound leads to an invalid state that is discarded (at run-time). If the bounds are omitted, the default range of -32768 to 32768 is used.



```
/*
 * For more details about this example, see
 * "Automatic Verification of Real-Time Communicating Systems by Constraint Solving"
 * by Wang Yi, Paul Pettersson and Mats Daniels. In Proceedings of the 7th Internati
 * Conference on Formal Description Techniques, pages 223-238, North-Holland. 1994.
 */

const N     5;          // # trains + 1
int[0,N]    el;
chan        appr, stop, go, leave;
chan        empty, notempty, hd, add, rem;
```

Fig. 3. Declarations of a constant and a variable, and illustration of some of the channel synchronisations between two templates of the train gate example of Section 4, and some committed locations.

Binary synchronisation channels are declared as `chan c`. An edge labelled with `c!` synchronises with another labelled `c?`. A synchronisation pair is chosen non-deterministically if several combinations are enabled.

Broadcast channels are declared as `broadcast chan c`. In a broadcast synchronisation one sender `c!` can synchronise with an arbitrary number of receivers `c?`. Any receiver than can synchronise in the current state must do so. If there are no receivers, then the sender can still execute the `c!` action, i.e. broadcast sending is never blocking.

Urgent synchronisation channels are declared by prefixing the channel declaration with the keyword `urgent`. Delays must not occur if a synchronisation transition on an urgent channel is enabled. Edges using urgent channels for synchronisation cannot have time constraints, i.e., no clock guards.

Urgent locations are semantically equivalent to adding an extra clock `x`, that is reset on all incoming edges, and having an invariant `x<=0` on the location. Hence, time is not allowed to pass when the system is in an urgent location.

Committed locations are even more restrictive on the execution than urgent locations. A state is committed if any of the locations in the state is committed. A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations.

Arrays are allowed for clocks, channels, constants and integer variables. They are defined by appending a size to the variable name, e.g. `chan c[4]; clock a[2]; int[3,5] u[7];`.

Initialisers are used to initialise integer variables and arrays of integer variables. For instance, `int i = 2;` or `int i[3] = {1, 2, 3};`.

Record types are declared with the *struct* construct like in C.

Custom types are defined with the C-like *typedef* construct. You can define any custom-type from other basic types such as records.

User functions are defined either globally or locally to templates. Template parameters are accessible from local functions. The syntax is similar to C except that there is no pointer. C++ syntax for references is supported for the arguments only.

Expressions in Uppaal Expressions in UPPAAL range over clocks and integer variables. The BNF is given in Fig. 33 in the appendix. Expressions are used with the following labels:

Select A select label contains a comma separated list of *name : type* expressions where *name* is a variable name and *type* is a defined type (built-in or custom). These variables are accessible on the associated edge only and they will take a non-deterministic value in the range of their respective types.

Guard A guard is a particular expression satisfying the following conditions: it is side-effect free; it evaluates to a boolean; only clocks, integer variables, and constants are referenced (or arrays of these types); clocks and clock differences are only compared to integer expressions; guards over clocks are essentially conjunctions (disjunctions are allowed over integer conditions). A guard may call a side-effect free function that returns a bool, although clock constraints are not supported in such functions.

Synchronisation A synchronisation label is either on the form *Expression*`!` or *Expression*`?` or is an empty label. The expression must be side-effect free, evaluate to a channel, and only refer to integers, constants and channels.

Update An update label is a comma separated list of expressions with a side-effect; expressions must only refer to clocks, integer variables, and constants and only assign integer values to clocks. They may also call functions.

Invariant An invariant is an expression that satisfies the following conditions: it is side-effect free; only clock, integer variables, and constants are referenced; it is a conjunction of conditions of the form `x<e` or `x<=e` where `x` is a clock reference and `e` evaluates to an integer. An invariant may call a side-effect free function that returns a bool, although clock constraints are not supported in such functions.

2.2 The Query Language

The main purpose of a model-checker is verify the model w.r.t. a requirement specification. Like the model, the requirement specification must be expressed in a formally well-defined and machine readable language. Several such logics exist in the scientific literature, and UPPAAL uses a simplified version of TCTL. Like in TCTL, the query language consists of path formulae and state formulae.[2] State formulae describe individual states, whereas path formulae quantify over paths or traces of the model. Path formulae can be classified into *reachability*, *safety* and *liveness*. Figure 4 illustrates the different path formulae supported by UPPAAL. Each type is described below.

State Formulae A state formula is an expression (see Fig. 33) that can be evaluated for a state without looking at the behaviour of the model. For instance, this could be a simple expression, like `i == 7`, that is true in a state whenever i equals 7. The syntax of state formulae is a superset of that of guards, i.e., a state formula is a side-effect free expression, but in contrast to guards, the use of disjunctions is not restricted. It is also possible to test whether a particular process is in a given location using an expression on the form `P.l`, where `P` is a process and `l` is a location.

In UPPAAL, deadlock is expressed using a special state formula (although this is not strictly a state formula). The formula simply consists of the keyword `deadlock` and is satisfied for all deadlock states. A state is a deadlock state if there are no outgoing action transitions neither from the state itself or any of its delay successors. Due to current limitations in UPPAAL, the deadlock state formula can only be used with reachability and invariantly path formulae (see below).

Reachability Properties Reachability properties are the simplest form of properties. They ask whether a given state formula, φ, *possibly* can be satisfied

[2] In contrast to TCTL, UPPAAL does not allow nesting of path formulae.

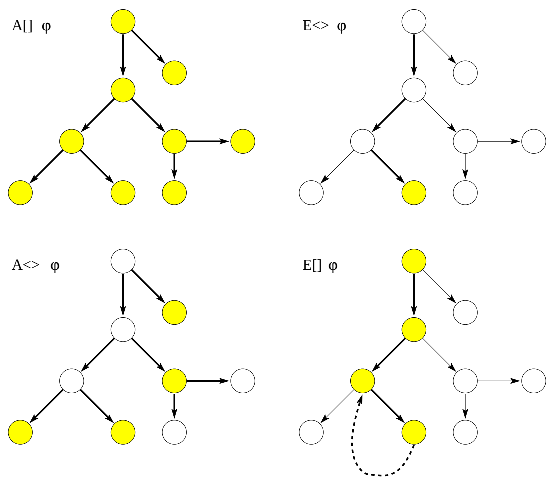


Fig. 4. Path formulae supported in UPPAAL. The filled states are those for which a given state formulae ϕ holds. Bold edges are used to show the paths the formulae evaluate on.

by any reachable state. Another way of stating this is: Does there exist a path starting at the initial state, such that φ is eventually satisfied along that path.

Reachability properties are often used while designing a model to perform sanity checks. For instance, when creating a model of a communication protocol involving a sender and a receiver, it makes sense to ask whether it is possible for the sender to send a message at all or whether a message can possibly be received. These properties do not by themselves guarantee the correctness of the protocol (i.e. that any message is eventually delivered), but they validate the basic behaviour of the model.

We express that some state satisfying φ should be reachable using the path formula $E\Diamond\ \varphi$. In UPPAAL, we write this property using the syntax `E<>` φ.

Safety Properties Safety properties are on the form: “something bad will never happen”. For instance, in a model of a nuclear power plant, a safety property might be, that the operating temperature is always (invariantly) under a certain threshold, or that a meltdown never occurs. A variation of this property is that “something will possibly never happen”. For instance when playing a game, a safe state is one in which we can still win the game, hence we will possibly not loose.

In UPPAAL these properties are formulated positively, e.g., something good is invariantly true. Let φ be a state formulae. We express that φ should be true in all reachable states with the path formulae $A\Box\ \varphi$,[3] whereas $E\Box\ \varphi$ says that

[3] Notice that $A\Box\ \varphi = \neg E\Diamond\ \neg\varphi$

there should exist a maximal path such that φ is always true.[4] In UPPAAL we write `A[]` φ and `E[]` φ, respectively.

Liveness Properties Liveness properties are of the form: something will eventually happen, e.g. when pressing the *on* button of the remote control of the television, then eventually the television should turn on. Or in a model of a communication protocol, any message that has been sent should eventually be received.

In its simple form, liveness is expressed with the path formula $A\Diamond\ \varphi$, meaning φ is eventually satisfied.[5] The more useful form is the *leads to* or *response* property, written $\varphi \leadsto \psi$ which is read as whenever φ is satisfied, then eventually ψ will be satisfied, e.g. whenever a message is sent, then eventually it will be received.[6] In UPPAAL these properties are written as `A<>` φ and φ `-->` ψ, respectively.

2.3 Understanding Time

Invariants and Guards UPPAAL uses a continuous time model. We illustrate the concept of time with a simple example that makes use of an *observer*. Normally an observer is an add-on automaton in charge of detecting events without changing the observed system. In our case the clock reset (`x:=0`) is delegated to the observer for illustration purposes.

Figure 5 shows the first model with its observer. We have two automata in parallel. The first automaton has a self-loop guarded by `x>=2`, `x` being a clock, that synchronises on the channel `reset` with the second automaton. The second automaton, the observer, detects when the self loop edge is taken with the location `taken` and then has an edge going back to `idle` that resets the clock `x`. We moved the reset of `x` from the self loop to the observer only to test what happens on the transition before the reset. Notice that the location `taken` is committed (marked `c`) to avoid delay in that location.

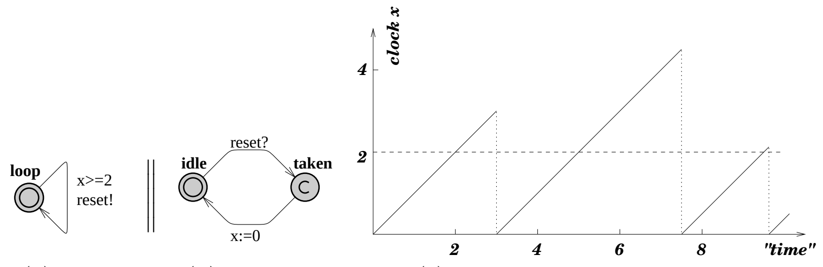
The following properties can be verified in UPPAAL (see section 3 for an overview of the interface). Assuming we name the observer automaton `Obs`, we have:

- `A[] Obs.taken imply x>=2` : all resets off `x` will happen when `x` is above 2. This query means that for all reachable states, being in the location `Obs.taken` implies that `x>=2`.
- `E<> Obs.idle and x>3` : this property requires, that it is possible to reachable state where `Obs` is in the location `idle` and `x` is bigger than 3. Essentially we check that we may delay at least 3 time units between resets. The result would have been the same for larger values like 30000, since there are no invariants in this model.

[4] A maximal path is a path that is either infinite or where the last state has no outgoing transitions.

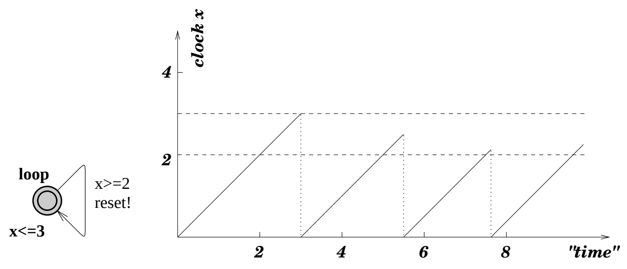
[5] Notice that $A\Diamond\ \varphi = \neg E\Box\ \neg\varphi$.

[6] Experts in TCTL will recognise that $\varphi \leadsto \psi$ is equivalent to $A\Box\ (\varphi \implies A\Diamond\ \psi)$



(a) Test. (b) Observer. (c) Behaviour: one possible run.

Fig. 5. First example with an observer.



(a) Test. (b) Updated behaviour with an invariant.

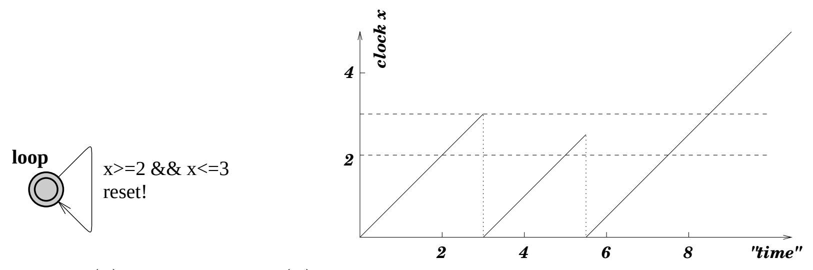
Fig. 6. Updated example with an invariant. The observer is the same as in Fig. 5 and is not shown here.

We update the first model and add an *invariant* to the location `loop`, as shown in Fig. 6. The invariant is a progress condition: the system is not allowed to stay in the state more than 3 time units, so that the transition has to be taken and the clock reset in our example. Now the clock `x` has 3 as an upper bound. The following properties hold:

– `A[] Obs.taken imply (x>=2 and x<=3)` shows that the transition is taken when `x` is between 2 and 3, i.e., after a delay between 2 and 3.
– `E<> Obs.idle and x>2` : it is possible to take the transition when `x` is between 2 and 3. The upper bound 3 is checked with the next property.
– `A[] Obs.idle imply x<=3` : to show that the upper bound is respected.

The former property `E<> Obs.idle and x>3` no longer holds.

Now, if we remove the invariant and change the guard to `x>=2 and x<=3`, you may think that it is the same as before, but it is not! The system has no progress condition, just a new condition on the guard. Figure 7 shows what happens: the system may take the same transitions as before, but deadlock may also occur. The system may be stuck if it does not take the transition after 3 time units. In fact, the system fails the property `A[] not deadlock`. The property `A[] Obs.idle imply x<=3` does not hold any longer and the deadlock can also be illustrated by the property `A[] x>3 imply not Obs.taken`, i.e., after 3 time units, the transition is not taken any more.



(a) Test. (b) Updated behaviour with a guard and no invariant.

Fig. 7. Updated example with a guard and no invariant.

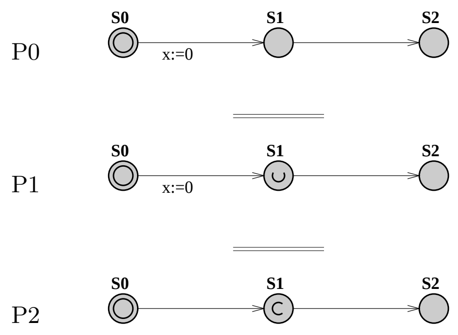


Fig. 8. Automata in parallel with normal, urgent and commit states. The clocks are local, i.e., `P0.x` and `P1.x` are two different clocks.

Committed and Urgent Locations There are three different types of locations in Uppaal: normal locations with or without invariants (e.g., `x<=3` in the previous example), urgent locations, and committed locations. Figure 8 shows 3 automata to illustrate the difference. The location marked `u` is urgent and the one marked `c` is committed. The clocks are local to the automata, i.e., `x` in `P0` is different from `x` in `P1`.

To understand the difference between normal locations and urgent locations, we can observe that the following properties hold:

- `E<> P0.S1 and P0.x>0` : it is possible to wait in `S1` of `P0`.
- `A[] P1.S1 imply P1.x==0` : it is not possible to wait in `S1` of `P1`.

An urgent location is equivalent to a location with incoming edges reseting a designated clock `y` and labelled with the invariant `y<=0`. Time may not progress in an urgent state, but interleavings with normal states are allowed.

A committed location is more restrictive: in all the states where `P2.S1` is active (in our example), the only possible transition is the one that fires the edge outgoing from `P2.S1`. A *state* having a committed location active is said to be committed: delay is not allowed and the committed location must be left in the successor state (or one of the committed locations if there are several ones).

3 Overview of the Uppaal Toolkit

Uppaal uses a client-server architecture, splitting the tool into a graphical user interface and a model checking engine. The user interface, or client, is implemented in Java and the engine, or server, is compiled for different platforms (Linux, Windows, Solaris).[7] As the names suggest, these two components may be run on different machines as they communicate with each other via TCP/IP. There is also a stand-alone version of the engine that can be used on the command line.

3.1 The Java Client

The idea behind the tool is to model a system with timed automata using a graphical editor, simulate it to validate that it behaves as intended, and finally to verify that it is correct with respect to a set of properties. The graphical interface (GUI) of the Java client reflects this idea and is divided into three main parts: the editor, the simulator, and the verifier, accessible via three “tabs”.

The Editor A system is defined as a network of timed automata, called processes in the tool, put in parallel. A process is instantiated from a parameterised template. The editor is divided into two parts: a tree pane to access the different templates and declarations and a drawing canvas/text editor. Figure 9 shows the editor with the train gate example of section 4. Locations are labelled with names and invariants and edges are labelled with guard conditions (e.g., `e==id`), synchronisations (e.g., `go?`), and assignments (e.g., `x:=0`).

The tree on the left hand side gives access to different parts of the system description:

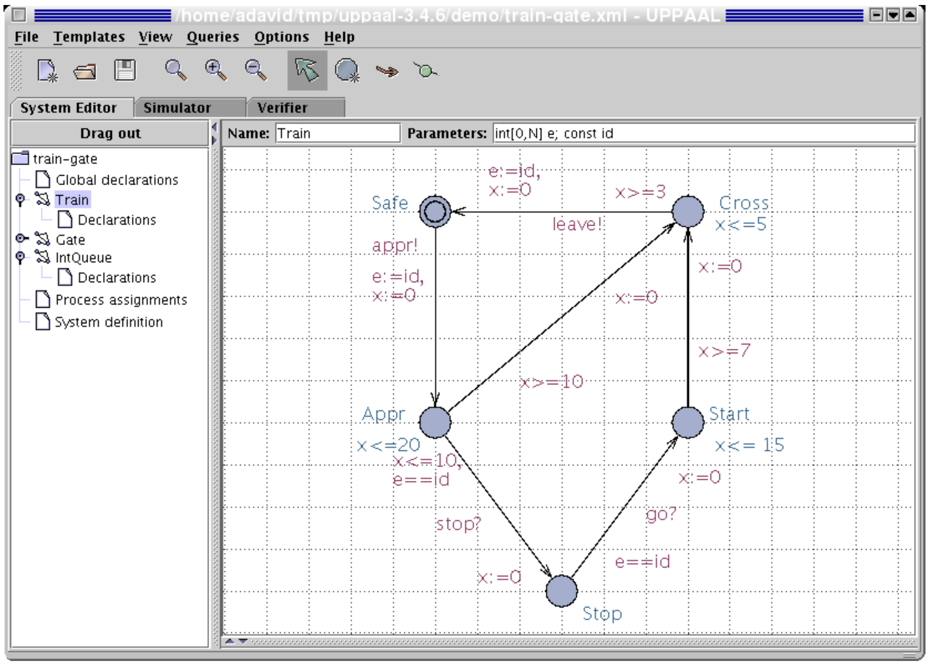
Global declaration Contains global integer variables, clocks, synchronisation channels, and constants.

Templates `Train`, `Gate`, and `IntQueue` are different parameterised timed automata. A template may have local declarations of variables, channels, and constants.

Process assignments Templates are instantiated into processes. The process assignment section contains declarations for these instances.

System definition The list of processes in the system.

The syntax used in the labels and the declarations is described in the help system of the tool. The local and global declarations are shown in Fig. 10. The graphical syntax is directly inspired from the description of timed automata in section 2.


/home/adavid/tmp/uppaal-3.4.6/demo/train-gate.xml - UPPAAL

Name: Train

Parameters: int[0,N] e; const id

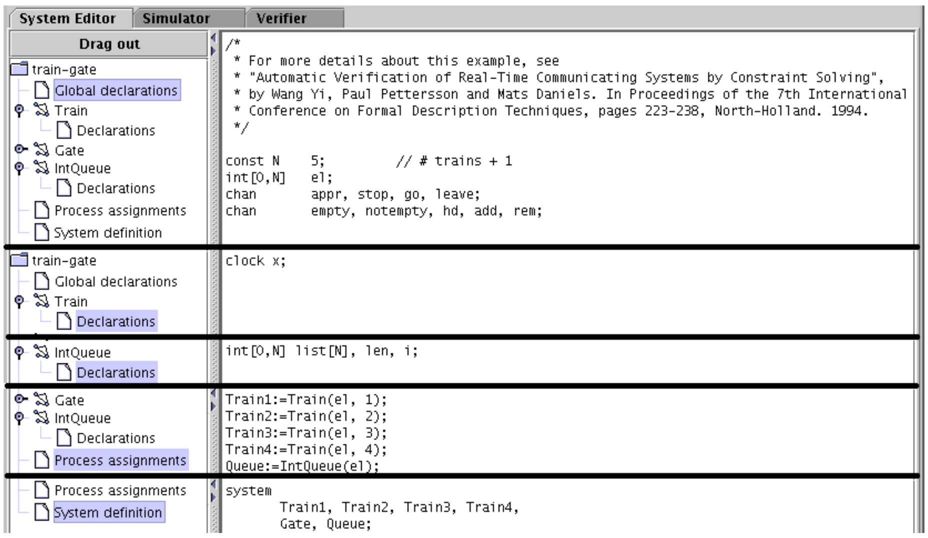
Fig. 9. The train automaton of the train gate example. The *select* button is activated in the tool-bar. In this mode the user can move locations and edges or edit labels. The other modes are for adding locations, edges, and vertices on edges (called nails). A new location has no name by default. Two text fields allow the user to define the template name and its parameters. Useful trick: The middle mouse button is a shortcut for adding new elements, i.e. pressing it on the canvas, a location, or edge adds a new location, edge, or nail, respectively.

The Simulator The simulator can be used in three ways: the user can run the system manually and choose which transitions to take, the random mode can be toggled to let the system run on its own, or the user can go through a trace (saved or imported from the verifier) to see how certain states are reachable. Figure 11 shows the simulator. It is divided into four parts:

The control part is used to choose and fire enabled transitions, go through a trace, and toggle the random simulation.

The variable view shows the values of the integer variables and the clock constraints. UPPAAL does not show concrete states with actual values for the clocks. Since there are infinitely many of such states, UPPAAL instead shows sets of concrete states known as symbolic states. All concrete states in a symbolic state share the same location vector and the same values for discrete variables. The possible values of the clocks is described by a set of con-

[7] A version for Mac OS X is in preparation.



```
/*
 * For more details about this example, see
 * "Automatic Verification of Real-Time Communicating Systems by Constraint Solving",
 * by Wang Yi, Paul Pettersson and Mats Daniels. In Proceedings of the 7th International
 * Conference on Formal Description Techniques, pages 223-238, North-Holland. 1994.
 */

const N     5;          // # trains + 1
int[0,N]    el;
chan        appr, stop, go, leave;
chan        empty, notempty, hd, add, rem;
```

```
clock x;
```

```
int[0,N] list[N], len, i;
```

```
Train1:=Train(el, 1);
Train2:=Train(el, 2);
Train3:=Train(el, 3);
Train4:=Train(el, 4);
Queue:=IntQueue(el);
```

```
system
       Train1, Train2, Train3, Train4,
       Gate, Queue;
```

Fig. 10. The different local and global declarations of the train gate example. We superpose several screen-shots of the tool to show the declarations in a compact manner.

straints. The clock validation in the symbolic state are exactly those that satisfy all constraints.

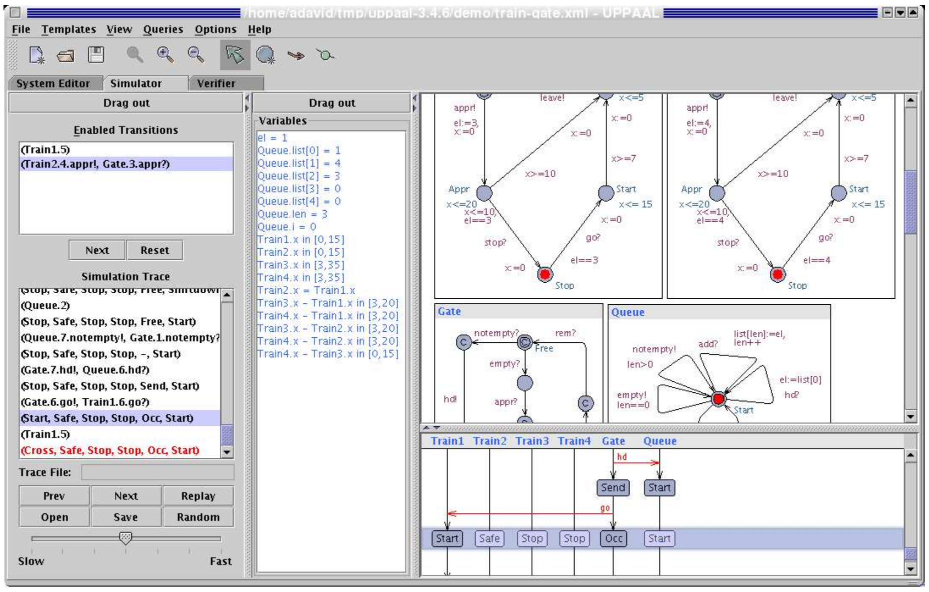
The system view shows all instantiated automata and active locations of the current state.

The message sequence chart shows the synchronisations between the different processes as well as the active locations at every step.

The Verifier The verifier "tab" is shown in Fig. 12. Properties are selectable in the *Overview* list. The user may model-check one or several properties,[8] insert or remove properties, and toggle the view to see the properties or the comments in the list. When a property is selected, it is possible to edit its definition (e.g., `E<> Train1.Cross and Train2.Stop` . . .) or comments to document what the property means informally. The *Status* panel at the bottom shows the communication with the server.

When trace generation is enabled and the model-checker finds a trace, the user is asked if she wants to import it into the simulator. Satisfied properties are marked green and violated ones red. In case either an over approximation or an under approximation has been selected in the options menu, then it may happen that the verification is inconclusive with the approximation used. In that case the properties are marked yellow.

[8] several properties only if no trace is to be generated.



```
(Train1.5)
(Train2.4.appr!, Gate.3.appr?)
```

```
(Stop, Safe, Stop, Stop, Free, Shutdown
(Queue.2)
(Stop, Safe, Stop, Stop, Free, Start)
(Queue.7.notempty!, Gate.1.notempty?
(Stop, Safe, Stop, Stop, -, Start)
(Gate.7.hd!, Queue.6.hd?)
(Stop, Safe, Stop, Stop, Send, Start)
(Gate.6.go!, Train1.6.go?)
(Start, Safe, Stop, Stop, Occ, Start)
(Train1.5)
(Cross, Safe, Stop, Stop, Occ, Start)
```

```
el = 1
Queue.list[0] = 1
Queue.list[1] = 4
Queue.list[2] = 3
Queue.list[3] = 0
Queue.list[4] = 0
Queue.len = 3
Queue.i = 0
Train1.x in [0,15]
Train2.x in [0,15]
Train3.x in [3,35]
Train4.x in [3,35]
Train2.x = Train1.x
Train3.x - Train1.x in [3,20]
Train4.x - Train1.x in [3,20]
Train3.x - Train2.x in [3,20]
Train4.x - Train2.x in [3,20]
Train4.x - Train3.x in [0,15]
```

Fig. 11. View of the simulator tab for the train gate example. The interpretation of the constraint system in the variable panel depends on whether a transition in the transition panel is selected or not. If no transition is selected, then the constrain system shows all possible clock valuations that can be reached along the path. If a transition is selected, then only those clock valuations from which the transition can be taken are shown. Keyboard bindings for navigating the simulator without the mouse can be found in the integrated help system.

3.2 The Stand-alone Verifier

When running large verification tasks, it is often cumbersome to execute these from inside the GUI. For such situations, the stand-alone command line verifier called `verifyta` is more appropriate. It also makes it easy to run the verification on a remote UNIX machine with memory to spare. It accepts command line arguments for all options available in the GUI, see Table 3 in the appendix.

4 Example 1: The Train Gate

4.1 Description

The train gate example is distributed with Uppaal. It is a railway control system which controls access to a bridge for several trains. The bridge is a critical shared resource that may be accessed only by one train at a time. The system is defined as a number of trains (assume 4 for this example) and a controller. A train can not be stopped instantly and restarting also takes time. Therefor, there are timing constraints on the trains before entering the bridge. When approaching,