Chapter 1
More Features in UPPAAL

Alexandre David, Kim G. Larsen

Abstract Following the introduction to the model checking tool UPPAAL of the previous chapter, this chapter presents a number of additional modeling and verification features offered by the tool. These features include in particular a C-like imperative language with user-defined types and functions, allowing for readable and compact models with reusable updates of discrete variables. Using an example of a Train Gate, we demonstrate the use(fulness) of these features. Also, the chapter presents the full query language of UPPAAL covering both safety, liveness and time-bounded liveness properties, again illustrated using the Train Gate example. Finally, directions are given on modelling choices and use of verification options that may improve time- and/or space-performance of the UPPAAL verifier

1.1 The Train Gate

In the previous chapter the basic modelling formalism of UPPAAL was presented: automata interacting over channels and extended with (integer) variables and clocks. For the ease of modeling, the full formalism of UPPAAL allows for structured variables (arrays and records) together with a C-like imperative language for their updates. To present and illustrate the use(fulness) of these features we use an example of a Train Gate. This example was originally presented in [27] as a number of trains running on seperate tracks, but – for economical reasons – having to cross a commong bridge. The challenge is to model the timing behaviour of the trains, as well as to design (and verify) a controller that will stop and (re)start trains in an appropriate manner, e.g. to avoid trains colliding on the common bridge.

Alexandre David and Kim G. Larsen
Department of Computer Science, Aalborg University, Selma Lagerlöfsvej 300, 9220 Aalborg, Denmark e-mail: `adavid,kgl@cs.aau.dk`

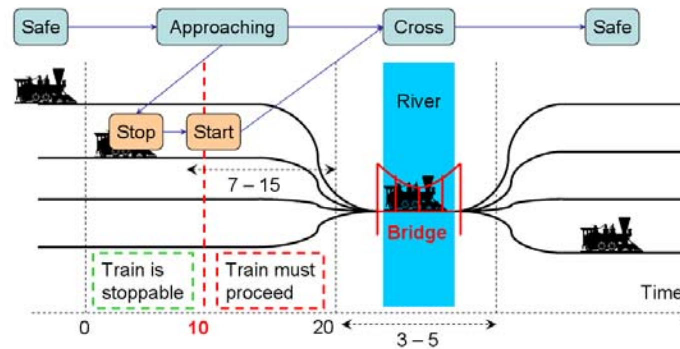

Fig. 1.1 The train gate problem. Trains run on their own tracks except on the bridge. Trains may be stopped before 10 time units, after which they must proceed to the bridge. If stopped a train will take some time to reach the bridge (7–15 time units). Crossing takes some time (3–5 time units).

A Simple Railway Control System [27]: We consider a railway control system to automatically control trains passing a critical point such as a bridge. The idea is to use a computer to guide trains from several tracks crossing a single bridge instead of building many bridges. Obviously, a safety property of such a system is to avoid the situation where more than one train are crossing the bridge at the same time.

Figure 1.1 depicts the problem of trains (here only 4) crossing a bridge. Initially trains are far enough from the bridge and are in a *Safe* state. At some point a train is approaching the bridge (state *Approaching*). The gate controller has then 10 time units to stop it. After this time the train has too much inertia to be stopped safely and must proceed to the bridge. It will take 20 time units to reach the bridge. If the train is stopped (state *Stop*) then it will be restarted again eventually (state *Start*) and it will take between 7 and 15 time units to reach the bridge. A train can be stopped at any time before 10 time units and so we model this non-determinism. When a train crosses the bridge it takes between 3 and 5 time units and we want as a safety property that only one train at a time has access to the bridge. After crossing, a train will go to its safe state again.

From the modelling point-of-view, this cyclic behaviour models different trains arriving on the same track. In addition, the behaviour of all the trains is the same and we only need a way to distinguish them, typically with a unique identifier. The model will then consist of a number of *train* instances derived from the same template and a *gate controller*.

1.2 User-Defined Types

In programming languages types allow for static checks to be performed, for ensuring that variables and expressions are used in a manner that will not lead to domain incompatibility in the sense that an operation is applied to a value that is *not* in its domain of arguments. The ability to define new types allow the user to identify and name value domains that are *not* primitive of the language being used, say "stacks" of characters equipped with operations for clearing, pushing, and popping a stack, selecting the top component, and testing for emptiness. For the modelling formalisms of UPPAAL a similar design decision has been taken.

The gate controller will typically need a queue to keep track of stopped trains and restart them. Trains are distinguished with a unique identifier whose range is defined by the total number of trains. It is natural and safe from a modelling point-of-view to declare a type for this identifier being a bounded integer. Furthermore we can structure the queue into one type that contains an array and a length. User-defined types are declared with the syntax

typedef type name;

where the type can be a bounded integer (*int*[min,max]) or a structure declared as

struct { type1; type2; ... }

In our example, the global declaration contains the following:

```
const   int N = 6;
typedef int[0,N-1] id_t;

chan        appr[N], stop[N], leave[N];
urgent chan go[N];
```

Here `id_t` is the type for identifiers that is used as argument for the template of trains. In addition, arrays of channels are declared for the communication between the gate and every train. Furthermore, the template of the gate has its own local declarations:

```
typedef struct {
        id_t list[N];
        int[0,N] len;
} queue_t;

queue_t q;
```

The type `queue_t` defines the domain of queues as records consisting of an array of identifiers and an associated length. The variable `q` over this type constitutes the actual queue to be used by the gate. Being declared locally, both `queue_t` and `q` are only visible within the gate template.

Figure 1.2 shows the templates used for the trains (a) and the gate (b). The train template has the argument `const id_t id` that defines its identifier. Its states

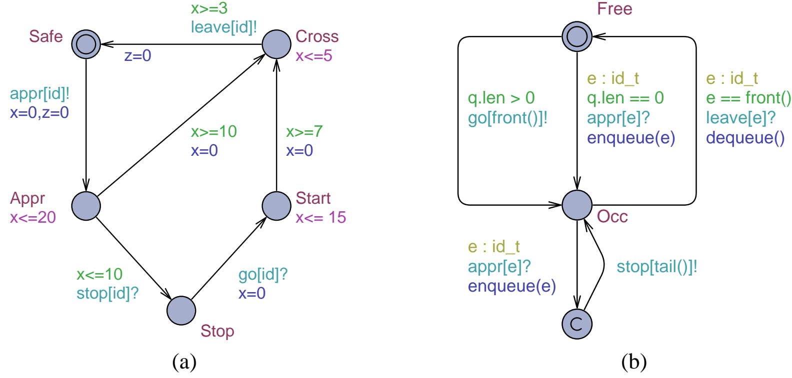

Fig. 1.2 Template for the trains (a) and the gate (b).

correspond to the states in Fig. 1.1. The communication with the gate is done with the channels with this identifier. When trains are approaching, the gate controller is notified with `appr[id]!` and when they leave the bridge they notify with `leave[id]!`. The gate controller can stop a train and then restart it. Trains listen on `stop[id]?` and `go[id]?` for this purpose. The different timing constraints of Fig. 1.1 are modelled with invariants on states and guards. Trains have a local clock `x` for this purpose (the purpose of the additional clock `z` will be explained later).

The gate controller uses and manipulates its queue structure in the automaton. When trains are approaching the controller enqueues their identifiers and dequeues them when they leave the bridge. If trains are approaching when the bridge is occupied (state *Occ*) they are stopped and their identifiers enqueued. The model is made more compact by using the so-called *select* statement `e: id_t` to unfold the corresponding edge with `e` ranging over the type `id_t`. More details on this useful feature will be given in section 1.4. To keep the template readable, queueing and dequeuing are performed with the help of user-defined functions, as will be detailed in the next section 1.3.

We note that one location of the gate is marked with "C" indicating that it is *Committed*. If a location is *Committed* then this means that time can not elapse within this location, similar to the condition for *Urgent* locations (see section **??**). However, for *Committed* locations transitions are in addition restricted only to those leaving committed locations. This removes interleaving between processes and allows the user to model atomic sequences of actions to do, e.g., a multicast[1].

[1] Broadcast channels are supported and are declared by prefixing the channel declaration by *broadcast*.

1.3 User-defined functions

As a recommendation to the programmer, in its formulation by Benjamin C. Pierce [24], the *Abstraction Principle* reads: “Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts”.

With the availability of discrete variables (integer, boolean and even structured variables as well as variables over user-defined types), their updates quickly become more involved, e.g., inserting an element into a sorted array. Updates of discrete variables take place on transitions as sequences of simple assignments. Instead of using complex automata to encode an update (even using committed states for that purpose), it is far more convenient, compact, and efficient to use a function that can pack complex control flow constructs such as nested conditional statements and loops. Thus, concerning the imperative part of a model UPPAAL provides support for the principle of abstraction.

For managing the queue of train identifiers of the gate controller we use functions to queue, dequeue, and access the tail and front of the queue. The declaration is shown in figure 1.3. C-like syntax is used with the extension of references (like in C++) and without pointers. Enqueuing adds an element at the end of the queue and increases the length of the queue. Dequeuing removes the front element and shifts all the elements. Here we point out the final reset of the last element to zero. From a programming point of view this reset seems completely superfluos as this element is no longer part of the queue and will have no effect on the subsequent behaviour. In fact an optimizing compiler would most likely remove the reset. However, in a model checker this reset to a default value is key to limit the state-space explosion problem. If we were not reseting here, the queue would remember the last element that was there, even though it is no longer in the queue and has no relevance for the future behaviour of the system. Thus states that are behaviourally equivalent, would be different, thus impacting the performance of the model-checker (you may want to check the validity of this claim yourself!). The functions for reading the front and tail elements of the queue are straight-forward.

While writing these functions in C is simple, implementing the same functionality in “pure” timed automata with simple updates is complex and error prone (but possible).

```
// Put an element at the end of the queue
void enqueue(id_t element)
{
        q.list[q.len++] = element;
}

// Remove the front element of the queue
void dequeue()
{
        int i = 0;
        q.len -= 1;
        while (i < q.len)
        {
                q.list[i] = q.list[i + 1];
                i++;
        }
        q.list[i] = 0;
}

// Returns the front element of the queue
id_t front()
{
   return q.list[0];
}

// Returns the last element of the queue
id_t tail()
{
   return q.list[q.len - 1];
}
```

Fig. 1.3 Declaration of the functions used locally by the gate controller.

1.4 Select label

The gate controller of figure 1.2.(b) is using a *select* statement. This statement has the effect of duplicating the edge into several instances with the variable(s) used in the select taking values over the specified range(s), which in fact could be any type (here `id_t`). This construct is useful for models with a parameterised number of processes having to synchronise. Arrays of channels can be used for that purpose, e.g. allowing in our example the controller to know with which train it is synchronising and subsequently store its identifier in its queue. This can also be interpreted as message passing using (here) the `appr` channel that carries the identifier of the approaching train. Fig. 1.4 shows the window used to edit that edge. The variable `e` declared in the select label has its scope on all the labels of that edge, here it is used in both the guard and synchronisation labels.

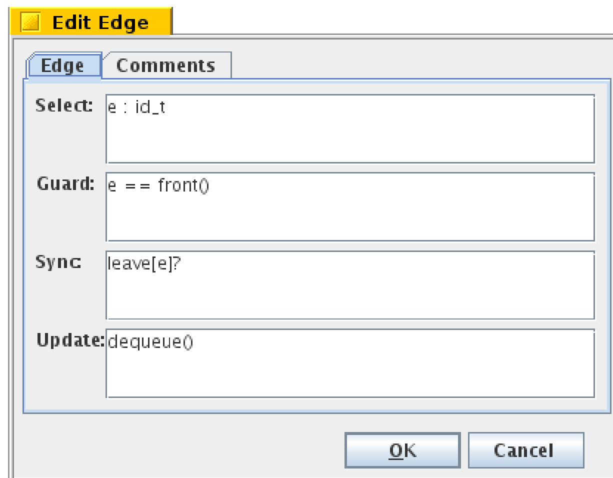

Fig. 1.4 Window for editing edges showing the different editable labels.

1.5 The Simulator Revisited

Having now completed the modeling of the Train Gate example, the first thing to examine the behaviour using the simulator of UPPAAL. Figure 1.5 shows a screenshot of the simulator, while simulating an instance of the Train Gate with six trains. From the simulation of the Jobshop example of the previous chapter, we recognize various parts of the simulator tab. The left part allows the user to control the simulation by choosing transitions and playing traces. The right part shows the automata with the active locations and the transitions that are taken, and below them a message sequence chart that shows the synchronisations between the automata.

However, whereas the center part of the simulator tab was completely empty for the Jobshop example, it now contains quite some information. In fact the center part provides information about the current value of variables and clocks. The user may select which information (s)he wants to pay attention to during a given simulation, by hiding (or viewing) processes and variables using the *View* menu.

Turning to the information offered for clocks, we see that the simulator does not exhibit concrete (real) values for the clocks but rather a collection of constraints on individual clocks, e.g. $x \leq 4$, or constraints on clock differences, e.g. $x - y < 10$. This reflects the fact, the model checking engine of UPPAAL does *not* perform state-space exploration based on concrete states with concrete values of clocks (which would be impossible due to the uncountably many such values) but rather based on *sets of clock values* described by such simple constraints. Sets of clock valuations described by constraints on clocks and clock differences are called *zones*, and may be represented in a canonical manner by so-called *difference bound matrices (DBM)*, where entries $b_{i,j}$ describe the upper bound on a clock difference $x_i - x_j$. In the *View* menu, the amount of information presented for clocks in the center part may be

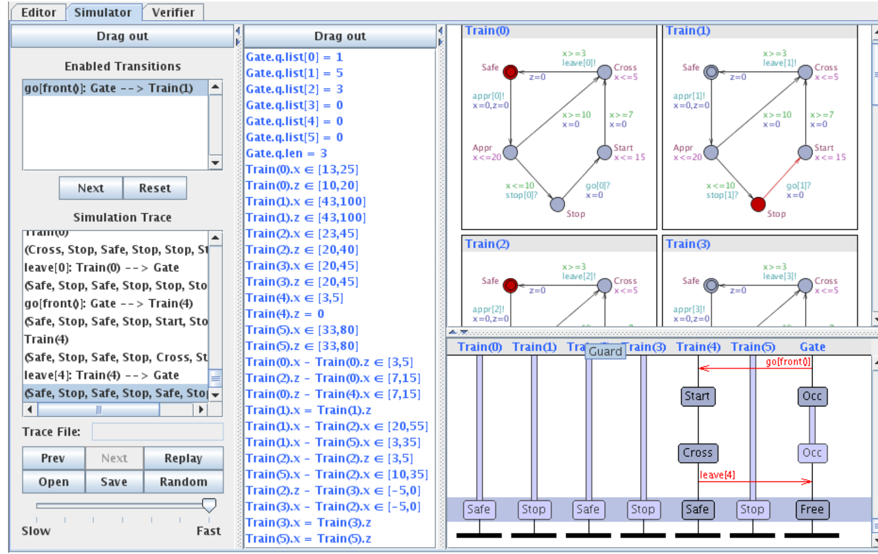

Fig. 1.5 Screenshot of the simulator running of the train-gate example.

affected. When the option *Full DBM* is selected, the all constraints present in the DBM will be shown. When the option is not selected, a reduced (but semantically equivalent) list of constraint is shown.

Thus, the model checking engine as well as the simulator of UPPAAL operates on *symbolic states* being tuples of the form (L,Z,V), where L is the location vector (active locations for all automata), Z is a zone, and V is the variable vector (values of all the integers). Figure 1.6 illustrates the sequence of symbolic states encountered during simulation of a simple timed automaton with two clocks x and y. As indicated in (a) simulation starts from the initial state where both x and y has the value 0.

- (a) From the intial state it is possible to delay and reach all states that respect the invariant of the initial location (i.e. $x \leq 4$). As the two clocks x and y increase in perfect synchrony, the resulting zone may be describe by the constraints: $x = y$ and $x \in [0,4]$.
- (b) Taking the transition that resets the clock y results in the zone described by $y = 0$ and $x \in [0,4]$
- (c) From each of these clock values (or points) delaying is again possible (except for $x = 4, y = 0$), which gives the zone described by the constraints $y \geq 0$ and $y \leq x$ and $x \in [0,4]$.
- (d) Finally, taking the transition guarded by $y >= 2$ adds this constraint resulting in the zone described by $y \geq 0$ and $y \leq x$ and $x \in [2,4]$.

Internally in the tool, these logical constraints are represented as a matrix with one extra special reference clock used for lower and upper bounds on individual clocks, e.g., $x \in [0,4]$. This representation also explains the limitation on the syntax for guards, namely they must be a conjunction of constraints of the form $x_i - x_j \leq b_{ij}$

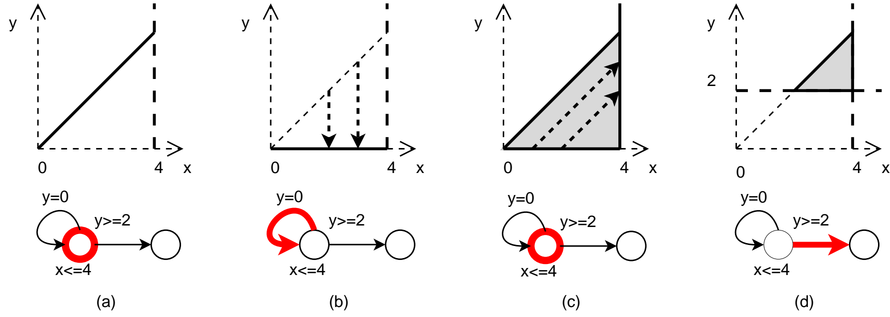

Fig. 1.6 Exploration of symbolic states. Starting from the origin $(0,0)$, (a) shows all the states reachable by delaying, (b) depicts the reset on the clock y, (c) shows a subsequent delay, and (d) shows the states that can take a transition guarded by $y \geq 2$.

(or a strict inequality). Within the simulator of UPPAAL, when selecting a transition, the variable view is updated to the symbolic state that *can* take that transition and the constraints change. When a transition is taken, the constraints are updated to reflect the resets and the delay that follows.

1.6 Queries Revisited

In the previous chapter, we considered basic UPPAAL queries of the types `A[]` ϕ and `E<>` ϕ for specifying safety and reachability properties of the job-shop example. As stated, these notations come from the field of temporal logic:

> In a temporal logic we can then express statements like "I am always hungry", "I will eventually be hungry", or "I will be hungry until I eat something".
>
> Temporal logic has found an important application in formal verification, where it is used to state requirements of hardware or software systems. For instance, one may wish to say that whenever a request is made, access to a resource is eventually granted, but it is never granted to two requestors simultaneously.

Two early contenders in formal verifications were Linear Temporal Logic, LTL (Amir Pnueli and Zohar Manna) and Computation Tree Logic, CTL (Edmund Clarke and E. Allen Emerson). In this section we will detail the full query language of UPPAAL, which is in fact a subset of CTL. Figure 1.7 illustrates the five formula-types supported by UPPAAL: `A[]` ϕ, `E<>`ϕ, `A<>`ϕ, `E[]` ϕ and $(\phi \leadsto \psi)$. The main restriction compared to full CTL is that UPPAAL does not allow nesting of formula, i.e. in the above ψ or ϕ must be state predicates refering only to locations, clocks and variables.

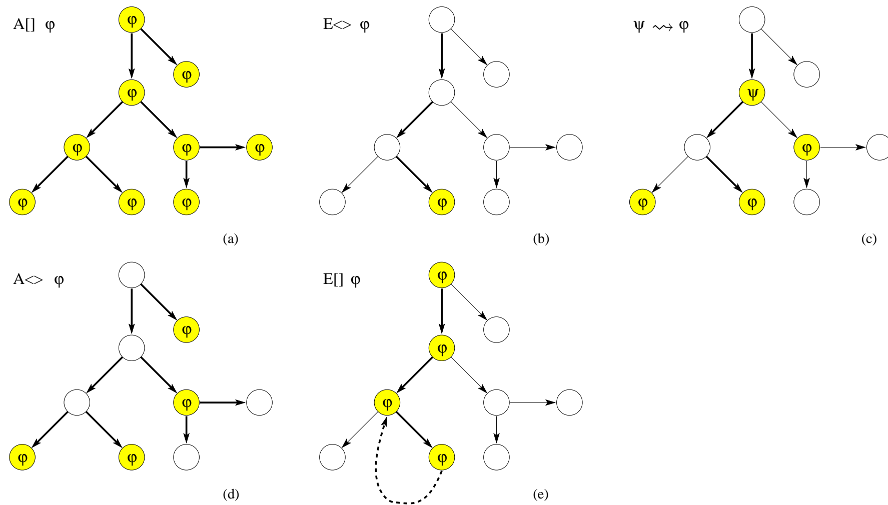

Fig. 1.7 The different types of logic formulas supported by UPPAAL.

1.6.1 Reachability

Reachability properties are of the form `E<>` ϕ and mean *there exists some path on which* ϕ *holds at some state* (Fig. 1.7.(b)). Reachability properties are useful for checking that models proposed at early design stages possess expected basic behaviours and to ask for diagnostic traces to confirm and study this more closely. For the Train Gate example such sanity properties could be:

```
E<> Gate.Occ
E<> Train(0).Cross
E<> Train(1).Cross
E<> Train(0).Cross and Train(1).Stop
E<> Train(0).Cross and (forall(i:id_t) i!=0 imply Train(i).Stop)
```

serving to check that the gate can be occupied, that trains 0 or 1 can cross, that train 0 can cross while train 1 is stopped, and that train 0 can cross while all *other* trains are stopped. In the last property – expected but maybe difficult to exhibit using manual or random simulation – *forall* is used over a range of indices.

1.6.2 Safety

Safety properties are of the form `A[]` ϕ and mean that *for all paths and for all states on those paths* ϕ *holds* (Fig. 1.7.(a)). We note that `E<>` $\neg\phi = \neg$ `A[]` ϕ, which means that `E<>` $\neg\phi$ gives a counter example in terms of a trace to a state that does not satisfy ϕ. For the Train Gate example expected safety properties are:

```
A[] forall (i:id_t) forall (j:id_t) \
    Train(i).Cross && Train(j).Cross imply i==j
```

```
A[] not deadlock
```

Here the first safety property expresses that the gate controller correctly implements mutual exclusion of the bridge, in that no two different trains can be in the crossing simultaneously. The nested usage of the *forall* construct ranging over `id_t`, ensures that the formula correctly (and conviniently) expresses mutual exclusion regardless of the number of trains.

Other properties that also fall within the category of safety properties are of the form `E[]` ϕ, that means *there exists a path on which* ϕ *always holds* (Fig. 1.7.(e)).

1.6.3 Liveness

Whereas safety properties are usefull for expressing "that something bad will never happen", they are not sufficient for ensuring that a designed system is adequate. Given the Train Gate example it is utterly simple to obtain a safe system guaranteeing no crashes on the bridge: simply use a gate controller that will stop all trains! Clearly, this is not satisfactory.

What is needed is the additional ability to express liveness properties of a system in the sense "that something good is guaranteed to eventually happen". The first liveness property has the form `A<>` ϕ expressing that *for all paths* ϕ *eventually holds* (Fig. 1.7.(d)). We note that `E[]` $\neg\phi = \neg$ `A<>` ϕ, which means that `E[]` $\neg\phi$ gives a counter example to the liveness property `A<>` ϕ in the form of an infinite path (witnessed as a loop) or a path that ends on a deadlock on which ϕ does not hold.

The second, and particularly useful, liveness property has the form ϕ `-->` ψ and should be read as ϕ *leads to* ψ. In fact this property is equivalent to (and a shorthand for) the formula `A[]` `(`ϕ `imply A<>` ψ`)`, and means that *whenever* ϕ *holds for a state, then* ψ *will always hold eventually for all paths starting from that state* (Fig. 1.7.(c)). More interestingly is its usage as a *time bounded liveness* property with the help of an observer as shown in Fig. 1.16.(a) of Section 1.9.

In our Train Gate example, we may want to ensure that whenever a train is approaching it eventually will be at crossing. This will clearly rule out the inadequate solution of a controller which (purposely) stops all train, or wrongly implemented controllers under which some trains might get stuck in the queue.

```
Train(0).App --> Train(0).Cross
Train(1).App --> Train(1).Cross
Train(2).App --> Train(2).Cross
...
```

1.6.4 Bounded Liveness and Performance Evaluation

Having studied the correctness of the model, we may be interested in its performance. Though it is essential to know that trains approaching will eventually reach

the crossing, we may additionally want to obtain lower and upper bounds on the time between trains being in the *Appr* and *Cross* locations. For this, we add a clock `z` to the model as shown in Fig. 1.2. The clock is reset whenever a train is approaching[2]. To determine the relevant time-bounds we perform a binary search using properties of the type

```
A[] Train(0).Cross imply Train(0).z <= UP
A[] Train(0).Cross imply Train(0).z >= LOW
```

with `UP` and `LOW` being constants used in the binary search, e.g., 1000, 500, 250, 125, ... Adding an extra clock to the model that is reset when we want to measure some time and asking a safety or reachability property on a specific state is a technique used for *bounded liveness*. We note that this is cheaper than liveness.

As an attractive alternative to performing the (manual) binary search, we may use the queries `inf{Pr}:exp` and `sup{Pr}:exp`, which returns the infimum (supremum) of the expression `exp` over all reachable states satisfying the state predicate `Pr`. For the Train Gate example the queries

```
inf{Train(0).Cross}: Train(0).z
sup{Train(0).Cross}: Train(0).z
```

will give us the desired time-bounds directly for this clock in the state *Train(0).Cross*. The state predicate is optional and not putting the brackets at all is the same as having *true* as the predicate. The bounds are here $7 \leq z \leq 125$ for 6 trains.

We could also have used a *stop-watch*, which is sometimes simpler, i.e., by adding the invariant `z'==0` to the state *Safe* to stop the clock and reseting it when entering this state. Then we would ask `sup: Train(0).z`, which gives the same upper bound. We note that using stop-watches makes the reachability problem undecidable but UPPAAL uses an over-approximation technique so the result is reliable. In this example, the bound is exact but it could have been looser.

1.7 Verification Options

The model checking technology comes with the curse of state space explosion, i.e. the number of states to be explored tend to grow exponentially with the number of components (automata, clocks, variables, etc.) of the model. Development of techniques for state-space representation and exploration for making model checking efficient in practice is an extremely active area. Benefitting from (and contributing to) this research, UPPAAL offers a number of verification options to affect the representation and exploration of the verirication engine. These are available in the GUI under the *Options* menu.

[2] The reset to *Safe* is to reduce the state-space and has an impact because as the clock is used in the property, it is always active.

1.7.1 Search Order

The classical search ordering depth-first and breadth-first search are supported, as well as random depth-first (Fig. 1.8). In scheduling models where one solution is wanted, depth-first (or random depth-first) will typically be the most efficient option. There is another option available in the 4.1.x versions, closest to target first. This is an experimental heuristic used to guide the search.

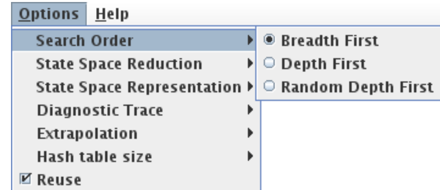

Fig. 1.8 Search order options.

1.7.2 State Space Reduction

These options are used to reduce the size of the *stored* state-space (Fig. 1.9). No optimisation can be chosen (none), committed states may not be stored unless they start a loop (conservative), only states starting loops will be stored (aggressive), or no state at all will be stored (extreme). The last option should be used with caution and is useful only when the model guarantees progress or is acyclic.

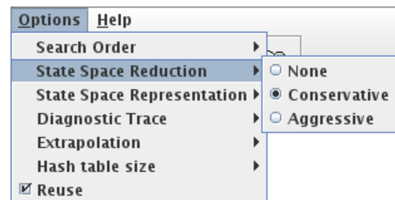

Fig. 1.9 State space reduction options.

1.7.3 State Space Representation

These options specify how to store individual state (Fig. 1.10.(a)). The option *DBM* uses the canonical matrix representation of constraints known as *difference bound matrix*. The option *Compact Data Structure* computes a reduced set of necessary constraints to store, which costs time but reduces memory footprint. The option *Under approximation* activate the *bit-state hashing* technique where every state is stored as one bit in a big hash table whose size is specified in the *Hash table size* option (Fig. 1.10.(b)). Finally the option *Over approximation* merges states together internally using an over-approximation technique (known as *convex-hull*), which reduces the number of states.

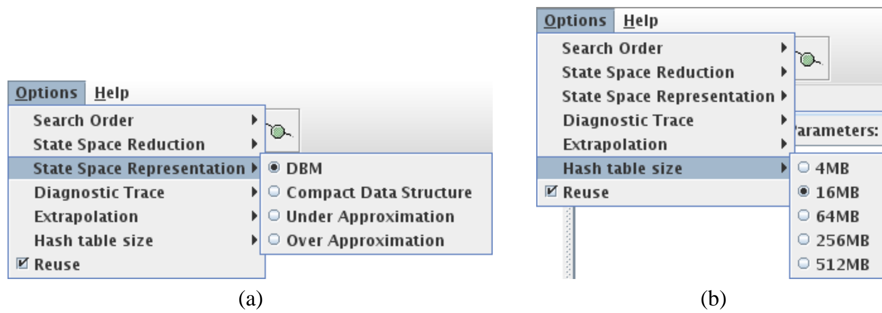

Fig. 1.10 State space representation (a) and hash table size (b) options.

1.7.4 Diagnostic Trace

To obtain a trace, a different option than "none" should be selected. When this is done, only one property at a time may be checked. Some trace may be obtained, or the shortest possible trace w.r.t. the number of steps, or the fastest w.r.t. time.

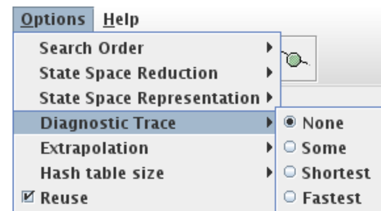

Fig. 1.11 Diagnostic trace options.

1.7.5 Extrapolation

Our verifier is using a symbolic technique to explore the state-space: rther than operating on concrete states with concrete values of clocks, the symbolic technique operates on *sets of clock values* (so-called zones) represented by constraints on clocks, $x \leq c$, and constraints on clock differences ,$x - y \leq c$. When analysing a particular timed automaton, static analysis will reveal that for each clock x there is a threshold value c_x above which the exact value of x is irrelevant for the behaviour of the timed automaton. This observation is crucial for abstractions (widening) of zones to obtain a finite symbolic state-space. This is in essence what the extrapolation does. It turns out there are different proposals for exptrapolation, that will be either exact or an over-approximation depending on the type of constraints used in the model. It is recommended to leave that option to *automatic*. The other settings are *none* (may prevent termination), *difference* (a more expensive operation useful when constraints of

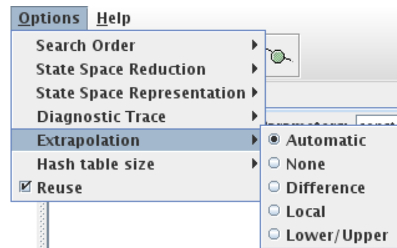

Fig. 1.12 Extrapolation options.

the form $x - y \leq c$ are used), *local* (the default), and *lower/upper* (an optimisation applicable for certain models).

1.7.6 Reuse

When checking consecutive reachability or safety properties, the model-checker may reuse the generated states if that option is selected. We note that alternating reachability and liveness property will cancel the benefit of that option.

1.7.7 Impact

Table 1.1 shows the impact of some of these options on a few examples that can be found on *http://www.uppaal.org* under examples/benchmarks. The example csma is a collision detection protocol, here with 10 nodes. Then we experiment with Fischer's protocol, a mutual exclusion protocol with 10 nodes here. Finally we use a token ring FDDI (fiber distributed data interface) protocol with 25 nodes. The properties checked here are safety properties and the depth-first search option makes the search a lot slower. This is explained by inclusion of symbolic states that will not be effective with that order. However, in other cases where a simple schedule is wanted, this order will work well. Deactivating compact data-structures will incur a small speed improvement and a large loss in memory compared to the default setting. The aggressive state-space representation stores fewer states and can sometimes (in these cases) give speed improvements. This is explained by the inclusion check that is done on fewer states.

	def	*dfs*	*S2*	*-C*
csma-10	5.8s	115s	5.6s	5.1s
	15.4M	16.5M	15.4M	24.6M
fischer-10	24.7s	111s	21.2s	20.6s
	23.9M	21.2M	15M	30.6M
fddi-25	4.4s	*	2.5s	3.8s
	13.7M	*	10.9M	29M

Table 1.1 Performance comparison with different options. The option *def* corresponds to the default setting, which is breadth-first search, compact data structure, and conservative state-space representation. The option *dfs* only changes the search order to depth-first. The option *S2* is the default setting changed with aggressive state-space representation. Finally the option *-C* is the default setting without the compact data-structure. We show results in seconds and MBytes obtained on a PentiumD running at 2.8GHz. The entries '*' mark an experiment that was stopped because it was taking more than 3 minutes.

1.8 Gossipping Girls: A Case-Study for Efficient Modelling

In this section we iterate over different versions of models to solve the gossipping girls problem (mentioned in the previous chapter), a notoriously difficult combinatorial problem. The goal is to expose the inherent limits of the model-checking technique, known as state-space explosion, and see how to change a model to improve performance.

The gossiping girls problem. Let n girls have each a private secret they wish to share with each other. Every girl can call another girl and after a conversation, both girls know mutually all their secrets. The problem is to find out how many calls are necessary so that all the girls know all the secrets. A variant of the problem is to add time to conversations and ask how much time is necessary to exchange all the secrets, allowing concurrent calls.

The basic formulation of the problem is not timed and is typically a combinatorial problem with a string of n bits that may take (at most) 2^n values for every girl. That means we have in total a string of n^2 bits taking 2^{n^2} values (in product with other states of the system).

1.8.1 Modelling in UPPAAL

We face choices regarding the representation of the secrets and where to store them. Every girl keeps track of her known secrets. The natural encoding would be to use an array of booleans. One could think of using a more compact encoding by choosing to use one integer to do so and to manage the bits manually as booleans. This limits the model to the number of bits available but as we have seen from the complexity, the state-space explodes too quickly for this to be a limiting factor. We will explore both encodings to see that the "optimized" version using integers is in fact not convenient at all for further refinements of the model. The second choice is where to store the messages, in one shared table or locally with every girl. The models described here can be found at *http://www.cs.aau.dk/~adavid/GossippingGirls/*.

In the following sub-sections we present different versions of the model. They will all use these common global declarations:

```
const int GIRLS = 4;
typedef int[0,GIRLS-1] girl_t;
chan phone[girl_t], reply[girl_t];
```

The declaration of the constant GIRLS allows us to scale the model easily. Notice that it is possible to declare that arrays of channels are indexed by a given type, which implicitly gives them the right size. This is useful for an optimization seen later.

The girl template is named Girl and has girl_t id as parameter. A first version of the the template is shown in figure 1.13. Every girl has a different ID. The *system* declaration is simply: system Girl;. This makes use of the *auto-instantiation* feature of UPPAAL. All instances of the template Girl ranging over its parameters are generated. The number of instances is controlled by the constant GIRLS.

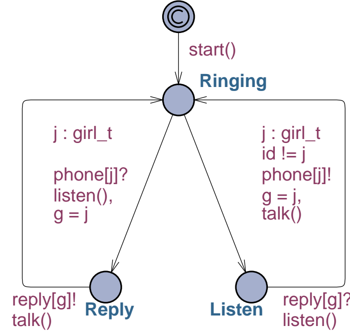

Fig. 1.13 First attempt for modelling the gossipping girls. The model consists in different instances of the same template with different identifiers to differentiate the girls. This is the template of a girl, taking as argument an identifier of type `girl_t`.

The template of a girl uses functions on its transitions to handle communication. It is good practice to use such function to improve readability of the model and to make it more flexible. We will change the internal data-structures and implementation of these functions but still keep the same automaton. Since the identifier parameter is a template argument, its scope is the whole template, which means it can be used directly in any local function. Here the start() function will initialize the girl with her unique secret (which depends on her identifier). The functions `talk()` and `listen()` aure used to send and receive secrets to and from other girls. The synchronization is done with the channels corresponding to other girls. We note that for replying, a girls needs to remember who she talked to so the model keeps track of that with a local variable `girl_t g`.

1.8.2 Representing Secrets With Boolean Arrays

We need to encode message passing between different processes, which is not directly supported by UPPAAL. To do so, the standard way is to declare a temporary shared variable. In addition, this variable is prefixed with the keyword *meta*, which means that it is a special temporary variable that will *not* be part of the states. This means that users should never refer to it between two states. Its value is only reliable on one given transition (possibly involving several edges in case of a synchronization).

We add to the global declarations meta bool tmp[girl_t]; to encode message passing. The functions mentioned previously are implemented as follows:

```
bool secrets[girl_t];
void start()  { secrets[id] = true; }
void talk()   { tmp = secrets; }
void listen() { for(i:girl_t) secrets[i] |= tmp[i]; }
```

In this version we use assignment between arrays for `talk()`. The function `listen()` uses an iterator. The template automaton is given in figure 1.13 and is common for both versions gossip1 (boolean encoding) and gossip0 (integer encoding). This first attempt captures the fact that we want the model to be symmetric with respect to sending and receiving and is quite natural with symmetric uses of `talk()` and `listen()`.

Initialisation is done by setting secret *id* to true. The initial committed location ensures all girls are initialised before they start to exchange secrets. Then we have a standard message passing using a shared variable with the receiver merging the secrets sent with her own (logical or).

1.8.3 Representing Secretes With Integers

This time we add `meta int tmp;` to pass integer messages between the girls. The functions are now implemented as follows to manage the integers' bits:

```
int secrets;
void start()  { secrets = 1 << id; }
void talk()   { tmp = secrets; }
void listen() { secrets |= tmp; }
```

Initialisation is done here by setting bit *id* to one. The other functions are similar to the boolean encoding but manipulating all bits at once this time.

1.8.4 Basic Improvements

Basic optimisations of a model.

1. Avoid useless interleavings by using committed locations.
2. Make sure to model exactly what you need and not more.
3. Use *active variable reduction*, which is to reset a variable to a fixed known value, whenever its value is not relevant to the current state.

Let us now apply and illustrate the above basic optimization rules using the Gossiping Girls example:

1. The intermediate state Listen should be made committed otherwise all interleaving of half-started and complete calls will occur.
2. One select statement is enough because we are modelling something else here, namely girl id selects a channel indexed by j and *any other* girl that selects the same channel index can communicate with girl id.
3. The local variable g contributes badly to the state-space when its value is not relevant, i.e., the previous communication does not need to be kept. We can set it in a symmetric manner upon the start and reset it after communication to id.

The updated model is shown in Fig. 1.14. This update is for both boolean (gossip3) and integer (gossip2) encodings. The template keeps as an invariant that the variable g is always equal to id whenever it is not sending. In addition, when a channel j is selected, then it corresponds to exactly girl j. Only one committed location is enough but it is a good practice to mark them both. It is more explicit when we read the model. The previous versions could only be checked up to 4 girls, now we can check 5 within roughly the same time. This is a very good improvement considering the exponential complexity of the problem.

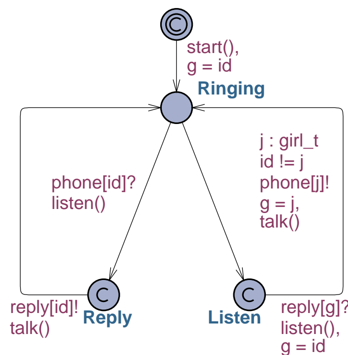

Fig. 1.14 Improved model of the gossipping girls.

1.8.5 Abstracting The Communication Protocol

We can abstract which communication line is used by declaring only one channel chan call. Since the semantics says that any pair of enabled edges (call!,call?) can be taken, we do not need to make an extra select. In addition, processes cannot synchronise with themselves so we do not need this check either. The downside is that we lose the information on the receiver from the sender point of view. We do not need this in our case. We can get rid of the local variable g as well. We can also simplify the communication protocol by merging the sequence listen()-talk() into one function and simplify listen() to a simple assignment since we know that the

message already contains the sent secrets. The global declaration is updated with only chan call; for the channel. The updated automaton is depicted in Fig. 1.15.

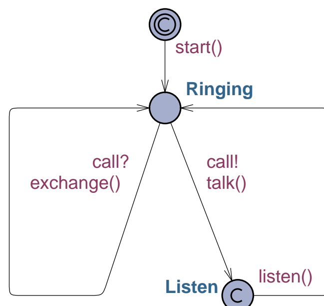

Fig. 1.15 Abstract model of the gossipping girls.

The *integer* version of the model (gossip4.xml) has the following local functions:

```
int secrets;
void start()    { secrets = 1 << id; }
void talk()     { tmp = secrets; }
void exchange() { secrets = (tmp |= secrets); }
void listen()   { secrets = tmp; }
```

The *boolean* version of the model (gossip5.xml) is changed with the functions:

```
bool secrets[girl_t];
void start()    { secrets[id] = true; }
void talk()     { tmp = secrets; }
void exchange() { for(i:girl_t) tmp[i] |= secrets[i];
                  secrets = tmp; }
void listen()   { secrets = tmp; }
```

The exchange function could have been written as follows:

```
void exchange() {
    for(i:girl_t) secrets[i] = (tmp[i] |= secrets[i]);
}
```

which is almost the same. The difference is that the number of interpreted instructions is lower in the first case. It is possible to further optimise the model by having one parameterised shared table and avoid message passing all-together. We leave this as an exercise for the reader but we notice that this change destroys the nice design with the local secrets to each process.

1.8.6 Verification

We check the reachability property that all girls will eventually know all secrets. For the integer version of the model, the property is:

```
E<> forall(i:girl_t) forall(j:girl_t)
     (Girl(i).secrets & (1 << j))
```

We can write a shorter but less obvious equivalent formula that takes advantage of the fact that $2^{GIRLS} - 1$ generates a bit mask with the first GIRLS bits set to one:

```
E<> forall(i:girl_t)
     Girl(i).secrets == ((1 << GIRLS)-1)
```

The formula for the boolean version is:

```
E<> forall(i:girl_t) forall(j:girl_t)
     Girl(i).secrets[j]
```

The formulas use the *for-all* construct, which gives compact formulas that automatically scale with the number of girls in the model. The version with the integers checks with a bit mask that the bits are set.

Table 1.2 shows the resource consumption for the different models with different number of girls. Experiments are run on an AMD Opteron 2.2GHz with UPPAAL rev. 2842. The results show how important it is to be careful with the model and to optimise the model to reduce the state-space whenever possible. We notice that the model is not even using clocks. The model with integers is faster due to its simplicity but consumes marginally less memory. The two last models (gossip6 and gossip7) are discussed in the next paragraph.

Girls	4	5	6	7
gossip0	0.6s/24M	498s/3071M	-	-
gossip1	1.0s/24M	809s/3153M	-	-
gossip2	0.1s/1.3M	0.3s/22M	71s/591M	-
gossip3	0.1s/1.3M	0.5s/22M	106s/607M	-
gossip4	0.1s/1.3M	0.2s/22M	37s/364M	-
gossip5	0.1s/1.3M	0.3s/22M	63s/381M	-
gossip6	0.1s/1.3M	0.1s/1.3M	3.4s/29M	399s/1115M
gossip7	0.1s/1.3M	0.1s/1.3M	0.3s/21M	29s/108M

Table 1.2 Resource consumption for the different models with different number of girls. Results are in seconds/M-bytes.

1.8.7 Improving Verification with Symmetry and Progress Measures

UPPAAL features two major techniques to improve verification. These techniques concern directly verification and are orthogonal to model optimisation. The first is *symmetry reduction*. Since we designed our model to be symmetric from the start, taking advantage of this feature is done by using a *scalar set* for the type `girl_t`. The second feature is the generalised *sweep-line* method. We need to define a progress measure to help the search. Furthermore, only the model with booleans is eligible

for symmetry reduction since we cannot access individual bits in an integers in a symmetric manner (using scalars).

Symmetry reduction. This technique exploits the structure of states in order to identify symmetries that occur during verification, in order to minimize the states-space that needs to be considered. The intuition behind symmetry reduction is that the order in which state components (automata, variables, clocks, etc) are stored in a state-vector does not influence the future behaviour of the system. Ideally, the reduced state-space will have only one state representing each symmetry equivalence class. As an example consider a protocol with n functionally identical nodes to implement a mutual exclusion. The only difference between the nodes is their respective identifier. It is not relevant to distinguish configurations where node 1 is in state A, node 2 in state B, and node 3 in state C from configurations where node 2 is in state A, node 3 in state B, and node 1 in state C. What matter is the number of nodes being in state A, state B and state C (respectively). This technique has the potential of giving an exponential gain in both time and memory.

In UPPAAL [17] symmtry reduction is activated whenever a *scalar* set is declared. A scalar set is a set of different scalar values that can only be compared for equality. A variable of a given scalar set type can only be set to another variable of the same scalar set type. Arithmetic operations that would break symmetry are not supported.

Sweep line method [11]. In models where it is possible to define some progress in the exploration, UPPAAL can save memory by "forgetting" past states if it knows it is progressing forward in the exploration. The technique works by declaring some progress measures that are used by the model-checker to delete states and save memory when it knows that it is making progress.

The only change required to take advantage of symmetry reduction is for the definition of the type `girl_t`. We use a *scalar* set for the new model (gossip6):

```
typedef scalar[GIRLS] girl_t;
```

To activate the sweep-line optimisation, we need to define a progress measure that is cheap to compute and relevant to help the search. It is important that it is cheap to compute since it will be evaluated for every state. To do so, we add `int m;` to the global declarations, we add the progress measure definition *after* the system declaration:

```
progress { m; }
```

Finally, we compute `m` in the exchange function as follows:

```
void exchange() {
    m = 0;
    for(i:girl_t) {
```

```
        m += tmp[i] ^ secrets[i];
        tmp[i] |= secrets[i];
    }
}
```

This measure counts the number of new messages exchanged per communication. The two last experiments in table 1.2 show that these features give gains with another order of magnitude both in speed and memory. The model still explodes exponentially, which we cannot avoid given its nature.

1.9 Modelling Tips

In this last section we summarize some of the modelling patterns that we have been using in our examples as well as present some additional ones.

1.9.1 Active variables

When the value of a variable is not important for the model, one should alwaysreset it to a default value, e.g., 0. This is what `list[i] = 0;` does in the `dequeue` function of the Train Gate example. If this statement is not here the model still works but states will keep memory of the last train that was in the queue, thus increasing the state-space needlessly. The same principle should be applied to all integers.

1.9.2 Value passing

Sometimes it is useful to have one process send a value to another. There are two ways to do that. The first is to use a *meta* variable. Such a variable is not part of the state and can be used only as a temporary place-holder on one transition. Its value is not reliable between two states. Typically the sender process synchronises with a `c!` and writes on a meta variable. Then the receiver process reads it when synchronising with a `c?`. Here `c` is a channel. In practice the instructions are executed first on the `c!` side, which explains why this trick works. A variant of this is to write directly the value into the destination variable on the sender or receiver side if the variables are shared. The drawback is that this makes for less modular modelling. The second way to encode value passing is to use arrays of channels. This is recommended for small ranges. One would declare `chan c[5];` and then send `i` with `c[i]!`. The trick is on the receiver where *select* is used with `i : int[0,4]`. The receiver can then receive with `c[i]?`.

1.9.3 Multi-cast

In cases when we want to synchronise from one process to several processes either from one central sender or in a chain, the following pattern should be used. Every step of the synchronisation should use a different channel and every intermediate location should be *committed*. Committed locations forbid interleaving with other non-committed locations. In a given committed state (i.e., a location vector, a variable vector, and a zone), only the transitions from its committed locations are allowed. In case of synchronisation, leaving one committed location (as part of the synchronisation) is enough.

1.9.4 Urgent transitions

The only way to model urgent transitions in UPPAAL is to use urgent channels. The question remains how to model a simple transition that we want to be urgent by itself? Simply add a dummy process with a self loop synchronising with `go!` on an urgent channel. The transition we want to be urgent synchronises with `go?`. We note that if the guard on that transition evaluates to false then delay is allowed (unless the state itself is urgent or committed).

1.9.5 Model Decoration and Monitors

Sometime the temporal formulas supported by the query language of UPPAAL are too limited. In such cases, an automaton acting as a monitor with an accepting or error state can help checking more complex properties involving causality between values and variables and time.

> Using model decoration and monitor is a general technique that consists in adding to the original model variables or a whole automaton to monitor the state of the system. This can be used in different ways to measure delays or to detect error states with some complex causality relationship.

As an example, let us suppose that we want to check a bounded liveness property of the form $\phi \leadsto_{\leq t} \psi$, i.e., whenever ϕ holds for a state then ψ will eventually hold on all paths starting from that state within t time units. It is not possible to check this property directly using only the formulas supported. Instead we use a monitor automaton following the pattern shown in figure 1.16.(a). The states marked ϕ are those for which ϕ should hold, similarly for ψ. The boolean `b` is set to true or false in the monitor. Not shown in the automaton are the guards to go between the states that should monitor the conditions ϕ and ψ. Those transitions should be made urgent.

The bottom (dark) states are the states for which the check is active (`b` is true). The property to check is then `A[](b imply z<= t)`.

A model can also be checked for non-*zenoness* with the help of an observer. In timed automata, it is allowed to have behaviours that will let an infinite amount of transitions to be fired (take actions) within a finite amount of time. This can be done by looping or blocking time with invariants and not using resets. Unless proper guards are used, this may happen but it is seldom a desirable property of the model. By using the observer of figure 1.16.(b) (let us call it `Obs`) in parallel with the reset of the model, one can check the property `Obs.A --> Obs.B`. In the automaton `C` is a constant set to a good value w.r.t. the rest of the model. The value itself is not very important as long as it is strictly positive.

A model is *zeno* if it allows an infinite number of discrete transitions to take place in a finite amount of time. In other words, it contains a loop where time does not diverge. This is an undesirable behaviour for a real system but it is very easy to obtain with timed automata. It is sometimes useful to check that a model does not allow such behaviours.

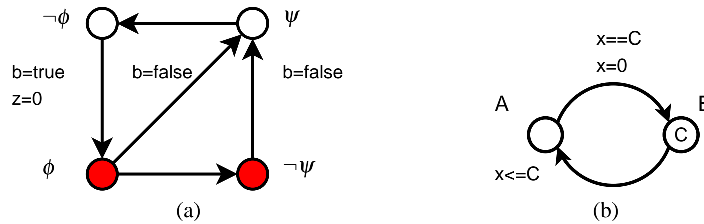

Fig. 1.16 Patterns for checking bounded liveness (a) and non-zeno behaviours (b).

1.10 Extensions of the Formalism

The UPPAAL tool suite has been specialised to different application domains. Here we mention its different variants.

CORA is a specialised version of UPPAAL that implements guided and minimal cost reachability algorithms [4, 5, 21]. It is suitable in particular for solving cost-optimal schedulability problems [2, 6]. The extension consists in adding a special `cost` variable to the model. The variable is put on location in conjunction with existing invariants in a *cost rate* expression of the form `cost' == expr` where `expr` is an expression that evaluates to a non-negative integer. In addition transition updates may have discrete cost increases with expressions of the form `cost += expr` with the same kind of expression.

TRON [20, 22] is a testing tool suited for black-box conformance testing [25, 19] of timed systems. It is mainly targeted for embedded software commonly found in various controllers. Testing is done online in the sense that that tests are derived, executed, and checked while maintaining the connection to the system in real-time.

TIGA [3] is a specialisation of UPPAAL designed to verify systems modelled as timed game automata where a controller plays against an environment. The tool synthesises code represented as a strategy to meet control objectives [14, 1, 23, 26]. The control objectives are specified as *until* or *weak-until* properties that are the more general forms of reachability and safety. The tool is based on an on-the-fly algorithm [9] and has be applied to industrial case studies [18, 10]. The tool can also handle timed games with partial observability [8] and has been extended [7] to check for simulation of timed automata and timed game automata.

PORT [15] is a version targeted to component-based modelling and verification. Its interface is developed as an Eclipse plug-in. The tool supports graphical modelling of internal component behaviour as timed automata and hierarchical composition of components. It is able to exploit the structure of such systems and apply partial order reduction techniques successfully [16].

ECDAR [13, 12] is a specialisation of TIGA that implements a recent specification theory that combines notions of specifications with a satisfaction relation, a refinement relation and a set of operators supporting stepwise design. The operators supported are composition, conjunction, and quotient. Specifications and implementations are given as timed I/O automata.

References

1. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller Synthesis for Timed Automata. In *Proc. IFAC Symp. on System Structure & Control*, pages 469–474. Elsevier Science, 1998.
2. G. Behrmann, E. Brinksma, M. Hendriks, and A. Mader. Scheduling lacquer production by reachability analysis – a case study. In *Workshop on Parallel and Distributed Real-Time Systems 2005*, pages 140–. IEEE Computer Society, 2005.
3. Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. UPPAAL-TIGA: Time for playing games! In *Proceedings of the 19th International Conference on Computer Aided Verification*, number 4590 in LNCS, pages 121–125. Springer, 2007.
4. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient guiding towards cost-optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer, 2001.
5. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits Vaandrager. Minimum-cost reachability for priced timed automata. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Proceedings of the 4th International Workshop on Hybris Systems: Computation and Control*, number 2034 in Lecture Notes in Computer Sciences, pages 147–161. Springer, 2001.
6. Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *ACM SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.

7. Peter Bulychev, Thomas Chatain, Alexandre David, and Kim G. Larsen. Efficient on-the-fly algorithm for checking alternating timed simulation. In *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems*, number 5813 in LNCS, pages 73–87. Springer, 2009.
8. F. Cassez, A. David, K. G. Larsen, D. Lime, and J.-F. Raskin. Timed control with observation based and stuttering invariant strategies. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis*, volume 4762 of *LNCS*, pages 192–206. Springer, 2007.
9. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR'05*, volume 3653 of *LNCS*, pages 66–80. Springer–Verlag, August 2005.
10. Franck Cassez, Jan J. Jessen, Kim G. Larsen, Jean-François Raskin, and Pierre-Alain Reynier. Automatic synthesis of robust and optimal controllers. In *Proceedings of the 12th International Conference on Hybrid Systems: Computation and Control*, volume 5469 of *LNCS*, pages 90–104. Springer, 2009.
11. Søren Christensen, Lars Michael Kristensen, and Thomas Mailund. A sweep-line method for state space exploration. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2001.
12. Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. ECDAR: An environment for compositional design and analysis of real time systems. In *Proceedings of ATVA 2010*, page to appear, 2010.
13. Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In *HSCC*, pages 91–100. ACM, 2010.
14. L. De Alfaro, T. A. Henzinger, and R. Majumdar. Symbolic Algorithms for Infinite-State Games. In *Proc. 12th Conf. on Concurrency Theory (CONCUR'01)*, volume 2154 of *LNCS*, pages 536–550. Springer, 2001.
15. John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. Component-Based Design and Analysis of Embedded Systems with UPPAAL PORT. In *ATVA*, pages 252–257, 2008.
16. John Håkansson and Paul Pettersson. Partial Order Reduction for Verification of Real-Time Components. In *Proc. of the 5th Int. Conf. on FORMATS*, LNCS, pages 211–226. Springer-Verlag, 2007.
17. Martijn Hendriks, Gerd Behrmann, Kim Guldstrand Larsen, Peter Niebert, and Frits W. Vaandrager. Adding symmetry reduction to uppaal. In Kim Guldstrand Larsen and Peter Niebert, editors, *FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2003.
18. Jan Jakob Jessen, Jacob Illum Rasmussen, Kim G. Larsen, and Alexandre David. Guided controller synthesis for climate controller using UPPAAL-TIGA. In *Proceedings of the 19th International Conference on Formal Modeling and Analysis of Timed Systems*, number 4763 in LNCS, pages 227–240. Springer, 2007.
19. Moez Krichen and Stavros Tripakis. *Model Checking Software*, volume 2989 of *LNCS*, chapter Black-Box Conformance Testing for Real-Time Systems, pages 109–126. Springer-Verlag, 2004.
20. K.G. Larsen, M. Mikučionis, and B. Nielsen. Online Testing of Real-time Systems Using UPPAAL. In *FATES'04*, LNCS, pages 79–94, Linz, Austria, September 2004.
21. Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV 2001*, number 2102 in Lecture Notes in Computer Science, pages 493–505. Springer, 2001.
22. Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Testing real-time embedded software using uppaal-tron: an industrial case study. In *the 5th ACM international conference on Embedded software*, pages 299 – 306. ACM Press New York, NY, USA, September 18–22 2005.

23. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proc. 12th Symp. on Theoretical Aspects of Computer Science (STACS'95)*, volume 900, pages 229–242. Springer, 1995.
24. Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
25. Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, 1992.
26. S. Tripakis and K. Altisen. Controller synthesis for discrete and dense-time systems. In *Proc. World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *LNCS*, pages 233–252. Springer, 1999.
27. Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7*th *Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.