INFO-F-410 Embedded Systems Design
Tools for Controller Synthesis

Gilles Geeraerts

Academic year 2012–2013

1 GAVS+

GAVS+ (*Game Arena Visualization and Synthesis, Plus!*) is an ongoing project, developed by Chih-Hong Cheng at the University of Münich. It is an open-source tool that groups several game solvers under a common GUI. GAVS+ is an academic project, but it provides a good idea of what can be done with an actual tool for solving game. GAVS+ can be downloaded from `http://www6.in.tum.de/~chengch/gavs/`. A concise tutorial is available on the web page. GAVS+ is written in Java.

The features of GAVS+ are numerous. In this practical, we will focus on the solver for safety and reachability games.

Installation Download the archive from the webpage, and extract it. Run the tool by launching `java -jar GAVS+.jar` in the `GAVS_dist` directory.

Exercise 1 – warm-up

In the tutorial, do the exercise described in part II – A on reachability games. For safety games, the nodes to avoid must be marked in red (see III – B).

Exercise 2

In this exercise, you will model a railway level crossing example. In this example, two tracks cross and we want to avoid that two trains are in the crossing at the same time. For simplicity, we will model only two trains, one on each track.

High speed trains travel on the first track. These trains cannot be stopped, but a sensor on the track allows to detect them two time units before they enter the crossing. The behaviour of the train on the first track is depicted in Fig. 1 (left). Initially, the train is out of the crossing. It can non-deterministically start to approach the crossing, by moving to the far state. The choice of staying in out or to move to far is thus *a decision of the environment*. When it is in far, it moves to 'approaching', at the next step, then gets in the crossing, and finally moves out of the crossing.

On the second track, local trains run (see Fig. 1, right). These trains are detected only when they approach the crossing. At that point they can either enter the crossing (in state) o be slowed down, if need be, by moving first to the wait state before entering in. The choice of the successor of app. is a *decision of the controller*.

We will model this system as a two players turn-based game. *Turn based* means that the two players will play in turn, alternating strictly.

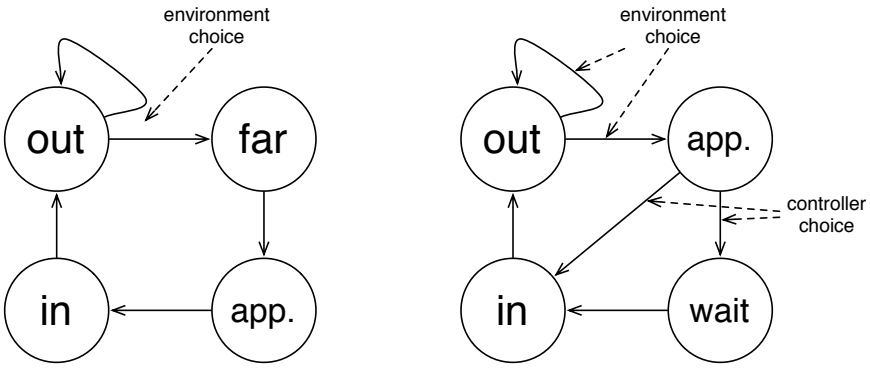


Figure 1: The behaviours of the two trains.

- The positions controlled by the environment are labeled by pairs (q_1, q_2) where q_1 is a state of the first train, and q_2 is a state of the second train. For instance $(\mathsf{far}, \mathsf{app})$ means that the first train is in the far state, and that the second is in the app state. In those states, the choice of the environment will be to decide which train(s) leave(s) the out state (if there is a state in the out state).

- The positions controlled by the controller are labeled by pairs (b_1, b_2), where b_1 and b_2 are two bits that indicate which train leaves the out state, when relevant. The successor of those states are obtained by letting one time unit elapse, and thus by letting the trains move to their next state. When the second train is in the app. train, there will be two successors, reflecting the two possible choices of the controller. Otherwise, there will be only one successor.

For instance, the environment state $(\mathsf{out}, \mathsf{app})$ has two successors labeled respectively by $(0, 0)$ (the second train stays in out) and $(1, 0)$ (the second train moves to far). The controller state labeled by $(1, 0)$ has two successors, labeled by $(\mathsf{far}, \mathsf{wait})$ and $(\mathsf{far}, \mathsf{in})$, that both take into account the choice of the environment (it has moved from out to far), and reflects the two possible choices for the controller.

First draw the game graph, then enter it in GAVS+, and check for the existence of a strategy that always avoids $(\mathsf{in}, \mathsf{in})$ using the safety game solving engine.

2 UppAal TiGa

This part of the practical will teach you to use the UppAal TiGa tool, that is developed by the universities of Uppsala (Sweden) and Aalborg (Denmark). UppAal TiGa allows to model synthesis problems with *timed games*, and to analyse those games. A short user manual is available.

Timed games The user manual gives a complete formal description of a timed game, we will briefly introduce them here, by comparing them to the Muller games we have studied during the lectures:

- Unlike Muller games, transitions (and not states) are controlled by the players in UppAal TiGa games. From the same state, there can be two types of transitions: controllable transitions that belong to the controller (solid lines) and uncontrollable transitions that belong to the environment (dashed lines).

- The game is *timed*, i.e., the elapsing of time is explicitly taken into account. The model boasts one or several *clocks* which are continuous variables that evolve with time elapsing, all at the same speed. Constraints on the values of the clocks and their reset can be specified in the game:
  - Each transition has a *guard*, which is a constraint on the clocks that has to be fulfilled for the transition to fire. For instance $1 \leq x \leq 5$, where x is a clock.
  - Each transition can reset clocks.
  - Each state is labelled by an invariant, that must be fulfilled during all the time spent in the state. When the invariant is about to be falsified, a transition must be taken to leave the state. For instance $x \leq 5$ forces to leave the state before $x > 5$.

  Clocks must be declared in the model, with a C-like syntax: `clock x;`

  Since the game is *timed*, players must now decide *when to play* in addition to *what to play*. In particular, when both a controllable and an uncontrollable transitions are active in the same state, with both guards satisfied, it is not possible to guarantee that the controller will always be able to play its controllable transition: the environment can always play faster.

Installation First download UppAal TiGa from the web page `http://www.cs.aau.dk/~adavid/tiga/`. You can then unzip it and launch the `tiga` script.

GUI The GUI has three parts:

1. An editor to build the model, graphically (by drawing the automata) and textually (by declaring variables and constants).
2. A simulator to play the game step by step.
3. A property editor to enter properties that have to be checked.

Exercise 1

Load the `toy01.xml` example from the `demo` directory. That simple example has two states and two transitions. The game must reach the *safe* state for the controller to win. First observe the syntax of the model: in the left pane of the editor, the 'Project' contains three components:

- 'Declarations' (empty in this case). This allows to make global declarations.
- 'Test': a timed arena, that, itself contains a 'Declaration' section. That section declares a single clock `x` referred to in the guards of 'Test'.
- 'System declarations', that declares the whole system thanks to the `system` keyword. Here the system contains only one instance of 'Test'.

Then,try to devise a winning strategy for the controller by looking at the example. Next, launch the simulator and simulate an execution where the controller wins, and one where the controller looses. The simulator can be reset by pressing the F5 key. Finally, go to the property editor and observe the properties (see the user manual for reference):

- `E<> Test.safe`: there exists (`E`) an execution that reaches (`<>`) the state safe.

- `A<> Test.safe`: all executions (`A`) eventually reach safe.
- `control: A<> Test.safe`: there exists a strategy that guarantees `A<> Test.safe`.

Which properties are verified by the model ? Try to find the answer by yourself before running the verifier !

To obtain the strategy (when it exists), one has to run a command-line tool, called `verifytga`, found in the `bin-Linux` directory. Go to that directory and run:

```
verifytga -t0 ../demo/toy01.xml
```

The tool will load the XML file as well as the *toy01.q* file containing the properties that we have observed in the property editor. Compare the generated strategy to your strategy.

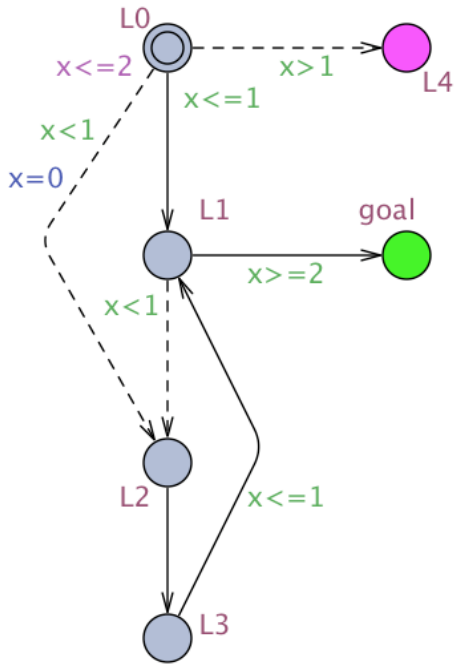
Next, further observe examples `toy01.xml` through `toy04.xml`: they are all variations on the same model. Observe how these slight changes modify the result of the verification step.

Is the example `toy06.xml` controllable (i.e., is there a winning strategy for the controller ?) Why ?

Exercise 2

Model the above game in UppAal TiGa and determine whether there exists a winning strategy

- To reach goal
- To avoid goal
- To reach goal while avoiding L4



Exercise 3

Model the railway crossing example of exercise 2 in UppAal TiGa.

1. The high speed train is modeled by one automaton with four states: out, far, app and in. It circles between the four states, and spends exactly one time unit in each state, except for the out state were it can spend *at least* one time unit.

2. The local train is modeled by a second automaton and has four states too, called out, app, wait and in. It runs as follows:

   (a) out is the initial state, the train spends at least one time unit there and moves to app,

   (b) the train spends exactly 2 time units in app and moves to wait,

   (c) the train spends between 0 and 3 time units in wait then moves to in,

   (d) the train spends exactly one time unit in in and goes back to out.

The only control option for the controller is to choose the time the local train spends in the wait state (between 0 and 3 time units). All the rest is uncntrollable (in particular, the controller cannot choose when either of the trains leaves the out state).

After modeling this system in UppAal TiGa, first check that you can reach a state where both trains are *in the crossing*. Otherwise, the control problem is trivial (all strategies for the controller would be winning...) Then, check whether there exists a winning strategy for the controller. What happens if you change the lattitude of the controller in the state wait of the local train ? Try by allowing at most 2 time units or at most 4 four time units.

Exercise 4

Load the example *rescue1.xml* which is made up of several automata. The example models a super-hero who has to rescue a crashing plane. An automaton describes the plane and is composed essentially of uncontrollable transitions (the super-hero is the controller in the game and can't control when the plane takes off for instance). The model of the super-hero contains mainly controllable transitions: he can decide when to go have a sip of coffee, or take a nap. When the super-hero is sleeping, he can be awaken by his alarm clock (and then heads first for coffee), or sooner (by an uncontrollable transition) and fly to the plance...

The automata communicate through a channel called `save`, which is declared in the global variables. A channel is a mean of communications between the automata. One can send a message on a canal with `nomCanal!`, and another automaton can read the message with `nomCanal?`. For instance, when the super-hero saves the plane, he sends a message `save!`, which allows the plane to move to the saved state by reading `save?`.

Observe where the canal is declared and how it is used for communication. Use the simulator to fire transitions where this happens.

Exercise 5 (optional)

Model a Chinese juggler who juggles by spinning plates on top of poles:

- There are n plates in the system that can't fall at any time. Initially, all the plates are spinning and stable.

- The juggler can, at any time, decide to spin a plate, by spinning the pole. Spinning takes time, the juggler can decide how long he spins the plates.
- There are several parameters in the system `SHORT`, `STABLONG` et `STABSHORT`. If the juggler spins a pole more than `SHORT` time units, that plate will stay stable exactly `STABLONG` time units from the moment the juggler stops spinning. Otherwise, the place stays stable `STABSHORT` time units.
- A mosquito can walk on one of the plates, and diminish the 'stability time' of the plate. The mosquito can fly from one plate to another, but this takes at least `WAITMOSQUITO` (another parameter) time units.

Model that system with one plate by creating an automaton for the plate, one for the juggler and one for the mosquito. Check that it is possible to reach (regardless of any strategy) a global state of the system where the plate falls, that such a situation can be avoided, and that there exists a winning strategy for the juggler to avoid that. Then add a second plate.

Here are some suggested values for the parameters (you can try other values):

```
const int STABCOURT = 2 ;
const int STABLONG = 4 ;
const int ATTENTEMOUSTIQUE = 9 ;
const int COURT = 2 ;
```