INFO-F-410 Embedded Systems Design
Uppaal TIGA and Prism games

Gilles Geeraerts, Nikita Veshchikov

This exercise session will teach you to use the tools UppAal TiGa and Prism games, that we have reviewed during the lectures.

1 UppAal TiGa

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 infinite games we have studied during the lectures:

- Unlike infinite 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 uncotrollable 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 -w0 ../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.

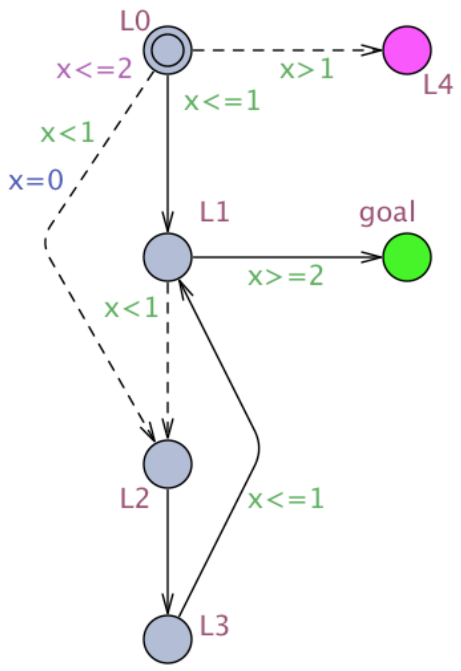
Next, further observe examples `toy01.xml` through `toy04.xml`: they are all variations on the same model. Observe how these slight changes modify the result of the verification step.

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

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 plane. . .

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.

Prism games

Prism games is a tool which is developed at Oxford University. You can download it from: `http://www.prismmodelchecker.org/games/`. The website of Prism games is pretty comprehensive, so we will use the tutorials available online.

Exercise 1

Model the Markov chain in `http://www.prismmodelchecker.org/tutorial/die.php` and run experiments to check that the probability to reach each side of the dice is the same and equal to $\frac{1}{6}$.

Exercise 2

Follow the example on `http://www.prismmodelchecker.org/games/` to model a communication protocol and check that Player 1 has a strategy to reach a state where $c = 2$ within 5 steps, and with probability ≥ 0.99. Generate this strategy and interpret its text version in the log.

Additionnal exercise on UppAal TiGa

Model a Chinese juggler who jungles 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 ;
```