Université Libre de Bruxelles

INFO-F410 - Embedded Systems Design

Electric-Bike Project (UppAal TiGa)

Student (MA1 Computer Science)
SWIRYDOWICZ Szymon
000477108

Lecturer / TAs
RASKIN Jean-François
BALACHANDER Mrudula
CHAKRABORTY Debraj
KHALIMOV Ayrat

April - May 2022

Contents

Chapter 1

UppAal TiGa

1.1 UppAal Tiga Introduction

This part of the project focuses on Uppaal and its extension Tiga. Uppaal is ”an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.)”.[1] TIGA is an extension of UPPAAL ”for solving games based on timed game automata with respect to reachability and safety properties”.[2]

1.2 Foreword

Tasks from U1 to U6 are implemented in the uppaal_project .xml and .q files, whereas U7 is available in uppaal_project2 .xml and .q as demanded. Note that to separate the Task U7 from the rest, Tasks U8 and U9 are stored in uppaal_project3 .xml and .q.

1.3 Task U1

This task requires us to two things. First, we modify the directed edge from the state Assist to Choose by adding the `timer1` equals 0 update property. This will put the edge to 0 each time this path is traversed.

Next, we need to actually define what timer1 is. Thus, we go to `Declarations`, and we add a line for *clock timer1.*

The clock variable, as its name suggest, serves to mesure the time progress. Clocks are usually useful to synchronise different system components

1.4 Task U2

In order to add the Boolean variable, we simply go to `Declarations` and add the int [0,1] brake variable, which is simply an integer that can take values 0 and 1.

Then, we insert a new and fresh template to model the braking system. We add three states: Pressed, Choose and Released, where the latest will have the `Initial` property checked. Additionally, we will add the Commited property to the Choose state, in order to force a transition (we do not want the braking system to remain in the Choose state for long).

Then, we add directed edges between all states except between Released and Pressed, as they will have to pass through the Choose state first.

We add the brake = 0/1 update from the Choose state respectively whether it goes to the Released or the Pressed state.

And finally we uncheck the Controllable property from all the edges.

[1]Source: https://uppaal.org/
[2]Source: https://uppaal.org/features/#tiga

1.5 Task U3

As before, we add a Boolean variable named batteryLevel, and we create a new template for the Battery system with the High (init), Low and Choose states.

As the battery can only discharge/charge when the motor is assisting/recharging, there will be two types of edges going from Choose to High/Low.

The first type allows to change the battery level (i.e go from High/Low to Low/High). This transition will update the `batteryLevel` accordingly if the corresponding `motorState` guards are respected.

The second type will allow to stay on the same battery level. To achieve this, additional edges were added from Chose to High/Low with guards that only check if the `batteryLevel` variable equals 1/0. That way, we will be able to do transitions such as High - Choose - High or Low - Choose - Low.

In case the `motorState` equals 0 (i.e neither assisting or recharging), system will only be able to transit using that second type of transition.

1.6 Task U4

We are supposed to execute a loop over the 4 states every 1 unit of time, that will allow to update the Battery, the Brake and the Motor in that exact order.

For this purpose, we start by creating for states: BatteryUpdate, BrakeUpdate, MotorUpdate and Wait. They will be connected by edges such that they for a loop: Battery to Brake to Motor to Wait and once again to Battery.

The Wait to Battery transition will be guarded with a timer called *schedule_timer* and it will check if it equals exactly 1, plus it will update the timer to 0 once the transition is taken. Additionally, the Wait state will have the *schedule_timer* $\leq$ *1* invariant so the Scheduler will not be able to remain the Wait state forever. That way, we will loop every one unit of time.

All the other transitions will have the Sync property, that thanks to Channels, will allow to send signals to other automata so force them to execute. What concerns the Battery, Brake and Motor update states, the will be checked as *urgent* so they are forced to transit one to another. Those will make the time freeze until we reach the Wait state, which will allow us to ensure we loop every one whole time unit. Note that those states are **not** marked as *committed*, in which case the Scheduler would first reach the Wait state of Scheduler before executing the *committed* states of the other automata.

1.7 Task U5

The observer is implemented using a one-state automaton. Every time the Brake systems chooses to go to the Pressed state, it emits a braking! broadcast, that tells the observer to reset the timer to 0. Since we do not want the timer to reset every time the Choose $\rightarrow$ Pressed, a guard condition was added to the Observer allowing the traversal if only the brake is set to 0. This means it will reset only if the previous state was Released. It is important to note that we use a broadcast channel and not a simple channel, in which case the Brake system would not be able to go to the Pressed state more than once in row (synchronisation).

1.8 Task U6

This task requires us to write a certain number of conditions that will have to be respected. The corresponding query formula has been written as follows:

```
control: A[] (Scheduler.Wait) imply
    (((Battery.Low and (timer2 >= 5 and Brake.Pressed)) imply not Motor.Idle) and (3.b)
    ((Brake.Released and Battery.High) imply not Motor.Idle) and                (3.a)
    ((timer1 <= 2) imply not Motor.Recharge) and                (2.c)
    (Battery.High imply not Motor.Recharge) and                 (2.b)
    (Brake.Released imply not Motor.Recharge) and               (2.a)
    (Battery.Low imply not Motor.Assist) and                    (1.b)
    (Brake.Pressed imply not Motor.Assist))                     (1.a)
```

Along side each line of the query have been added the corresponding conditions requested in project's instructions. The `control` prefix allows to execute the query in Uppaal Tiga instead of the usual Uppaal[3], and thus look for a winning strategy instead of verifying that those conditions always hold.

1.9 Task U7

1.9.1 Queries

In order to verify that Conditions 1, 2 and 3 hold, we will test a slightly different query from the one given in Task U6:

```
  A[] (Scheduler.Wait and scheduler_timer != 1) imply
    (((Battery.Low and (timer2 >= 5 and Brake.Pressed)) imply not Motor.Idle) and (3.b)
    ((Brake.Released and Battery.High) imply not Motor.Idle) and            (3.a)
    ((timer1 <= 2) imply not Motor.Recharge) and                            (2.c)
    (Battery.High imply not Motor.Recharge) and                             (2.b)
    (Brake.Released imply not Motor.Recharge) and                           (2.a)
    (Battery.Low imply not Motor.Assist) and                                (1.b)
    (Brake.Pressed imply not Motor.Assist))                                 (1.a)
```

The major difference is that we have removed the `control` prefix, which will assure that those conditions will always hold, instead of finding a winning strategy like we did in task U6.

Important assumption Please note the `schedule_timer != 1` added to the `Scheduler.Wait` check. It comes from the very design of the Scheduler and the following property from **Task U4** it must respect:

```
  At state Wait it should stay for exactly 1 time unit and go back to the state BatteryUpdate.
```

In the Scheduler, the system leaves the state **Wait** to **BatteryUpdate** only when the `schedule_timer` timer reaches 1 unit. In consequence, there might exist a really brief instant at which the `schedule_timer` equals 4, we are both in the Scheduler.Wait and in Motor.Idle (which is allowed), and just before leaving the **Wait** state at exactly `schedule_timer = 1`, the condition (3.b) instantly fails as `timer2` equals now 5 and technically we are still in the `Wait` state.

For this reason, `schedule_timer != 1` was added to the query.

What concerns the fourth Condition, it can be easily checked in Uppaal with the following query, which will check if for all possible branches we never reach a deadlock:

```
  A[] not deadlock
```

1.9.2 Guards

In order to make sure the conditions are respected, we will simply put the negation of those conditions on the corresponding motor states. For example, the condition "The motor should not assist when the brake is pressed, or the battery is low", can be implemented by adding the guard:

```
brake == 0 and batteryLevel == 1
```

[3]Source: https://people.cs.aau.dk/~adavid/tiga/manual.pdf

on the edge going from Choose to Assist.

In the same way, we add the following guard for the 2nd Condition:

```
brake == 1 and batteryLevel == 0 and timer1 > 2
```

and the below one for the 3rd Condition:

```
(brake==1 or batteryLevel==0) and (batteryLevel == 1 or (timer2 < 5 or brake == 0))
```

Note: As Uppaal can be quite capricious with negations, the guards had to be slightly rewritten and could not be wrote using the simple form *not(Condition).*

1.9.3 Results

Using the Uppaal's verifier tool, we obtain that the two queries/properties above are successfully satisfied after adding the motor guards.

1.10 Task U8

In this task, we have to check if "Always, if the brake has been pressed for more than 1 time unit, the motor engage recharge".

This can be checked using the following query:

```
(Brake.Pressed and timer2 > 1) --> (Motor.Recharge)
```

When Verified by Uppaal, it is stated as unsatisfied.

We can find two easy counter-examples, that are due to the previously defined conditions:

2.c As first counter-example, it can be blocked by the `timer1 > 2` guard. Since the Assist state can only be reached when the cyclist is not braking, we might get a trace during which timer1 of the motor and timer2 of might evolve at the same pace (i.e `timer1 == timer2`). In consequence, we might reach Motor.Choose, and despite reaching timer2 with brake == 2 (first part of the property), we might not be able to reach enter Motor.Recharge because timer2 also equals 2, and thus we are blocked by the Condition 2.c

2.b The second counter-example, is obtained thanks to the `batteryLevel == 0` guard. No matter for how long the cyclist has been braking, the system will not be able to enter Motor.Recharge if the battery is high.

As a reminder, the battery can go to Low only if the motor is assisting, but there exist a scenario in which the battery remains indefinitely High. Additionally, no matter how long we will be braking, if the battery is High, we will not be able to enter Motor.Recharge since we would need to enter Motor.Assist to lower the battery, which in turn would require to stop braking.

1.11 Task U9

1.11.1 Rim brakes

We start by inserting a new template we will call Rimbrake. It contains two states Released (initial) and Pressed. Next we define a binary channel as `chan rimUpdate[2]`. All transitions are done by synchronisation with the Motor automaton as described in the project's instructions. Whether Motor signals `rimUpdate[0]!` we either go from Pressed to Released or we stay on Released, and similarly for `rimUpdate[1]!` and the Pressed state.

To signal `rimUpdate[0]!`, we simply added the sync property on edges Choose → Assist and Choose → Recharge. However, we can **not** only add `rimUpdate[1]!` to Choose → Idle, as we do not want to signal anything if the automaton goes to Motor.Idle while it could have gone to the Motor.Recharge state; the signal shall only be sent if we the Motor.Idle state has been reached while Motor.Recharge could have not.

A first naive solution would be to copy the guard from Choose → Idle, and add another transition that respects that guard, but that additionally makes sure that:

```
and (brake == 1)
and (batteryLevel == 1 or timer1 <= 2)
```

This however, would not make the two transitions exclusive, and so the Motor could always take the initial transition without emitting the signal. Thus, we add the following to the initial Choose → Idle to make it exclusive with the new one:

```
and ((brake == 0) or (batteryLevel == 0 and timer1 > 2))
```

1.11.2 Property check

Finally, we have to check the following property: "Always, if the brake has been pressed for more than 1 time unit, the motor either engage recharge or the rim-brakes are pressed".

It can be tested using the following query:

```
(Brake.Pressed and timer2 > 1) --> (Motor.Recharge or Rimbrake.Pressed)
```

But as expected from the Rim brakes definition, the property is not satisfied. This is due to the fact that the transitions Choose → Idle and Choose → Recharge are not exclusive. For instance, with

- `brake == 1`
- `timer1/2 == 3`
- `batteryLevel == 0`

it is allowed to reach both Motor.Idle and Motor.Recharge from Motor.Choose. In consequence, the property is not satisfied. Additionally, since Motor.Choose → Motor.Idle transition can not be taken anymore if braking happens for more than 5 time units, one might think that the Motor will eventually reach Motor.Recharge. However, after reaching Motor.Idle, the cyclist can at any moment stop braking which will restart the whole process.

Note that if we took look at this property from a strategy game point of view, it would probably be satisfied. Since the edges of the Motor automaton are controllable, we would have the choice to go to the Recharge state instead of the Idle in order to win the game.