INFO-F410:
Electric-Bike Project

April 2022

Abstract

This note describes the project for INFO-F410. The project is to be made on a strict individual basis.[1]

Report. The structure of the report folder is the following:

```
your_name/
  lustre/
    lustre_report.pdf
    main.lus
  strix/
    strix_report.pdf
    <strix_project_files>
  uppaal/
    uppaal_report.pdf
    <uppaal_project_files>
  xcos/
    xcos_report.pdf
    <xcos_project_files>
```

1 Introduction

You are the chief safety engineer at SuperVolt Corp., and lead a project on developing the world-safest electric bike. Only you can stop the company from releasing the bike with critical safety issues, saving it from billion-euro fines, and protect people lives. Your aids are: formal verification, synthesis, testing, and simulation. ;-)

The main goal of this project is to allow you to experience the use of formal models supported by verification and synthesis and simulation techniques as aid to the design of a piece of reactive software. This reactive software is part of a representative embedded system. We ask you to design and verify a part of a motor controller of an electric bike, on different abstraction levels. You will synthesise automatically and design manually models, and write logical specifications, from the informal descriptions given in this note.

[1]As a consequence any breach of this rule will trigger the application of the general rules of the university to combat plagiarism, which a.o. include exclusion. Those rules will apply both for the copying side as well as the source side.

The project consists of four parts. In the part on synthesis, you will automatically synthesise a part of the motor controller using STRIX [5]. In the part on verification of timed automata, you will design, verify a timed model and fins strategy using UPPAAL TIGA [2]. In the part on reactive programming, you will implement the controller manually in the synchronous data-flow language LUSTRE [3]. Finally, the part on simulation requires designing a physical dynamics model of a bike and then simulating it in XCOS [4].

2 High-level description of the bike components

The electric bike is schematically depicted in Fig. 2.

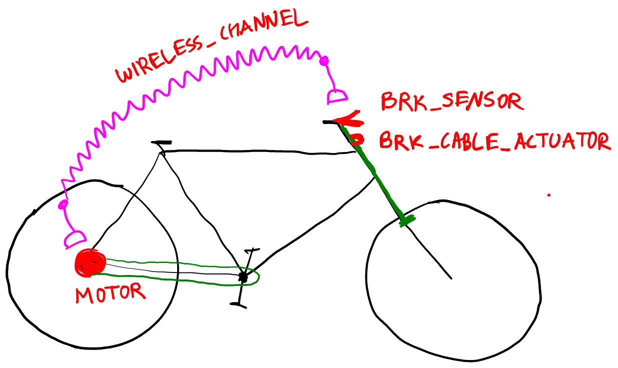


Figure 1: Components of the bike: motor controller, brake sensor, brake-cable actuator, wireless channel.

The bike comes equipped with an electric motor and a braking system. The braking system consists of a brake sensor and actuator. The brake sensor `BRK_SENSOR` outputs a signal whenever the rider presses on the brake lever. There are two press levels of the braking: soft braking and hard braking (and, of course, no braking at all). This signal is read by the brake actuator `BRK_ACT`. The braking level is also sent wirelessly to the motor. The wireless communication is not reliable. When the connection is active, on receiving the soft or hard braking command, the motor controller should engage the regenerative braking using the motor. Such a regenerative braking charges the battery. On the other hand, when the communication between the braking sensor and motor controller is lost, the regenerative braking should not be engaged. In this case, the braking actuator should clinch the brake cable leading to the rim brake on the front wheel, engaging the manual brake. The actuator should also clinch the cable and engage the manual braking when the rider presses hard on brakes.

The motor is also used to assist the cyclist. To this end, the motor controller reads the signal from the power sensor measuring the current effort by the rider, the current speed, and the battery level. Finally, note that the motor should not assist the cyclist on speeds above 25 km/h nor when the braking is active nor when the battery level is 0. The motor should not engage in assisting nor braking on losing communication with the braking sensor.

The rules above are a subset of operational rules of the braking system and the motor controller. We now look at them in more detail.

Model decomposition

Figure 2 depicts the components that compose the system and their interaction. Those pieces of equipment communicate with each other by emitting and reading events[2]. We adopt here the vocabulary of the synchronous approach to reactive systems as used in LUSTRE [3]. In the following description, we assume that the *execution* of the system can be represented as a time line of linearly ordered logical time points $t_0, t_1, \ldots, t_n, \ldots$. In each time point, each signal (Boolean or valued) can be present or absent. If a valued signal is present, then it carries a value. For example, if there are two signals: one Boolean $A : \mathbb{B}$, and one integer-valued $B : \mathbb{Z}$, then the following sequence:

$$(t_0, \{A, B(1)\}), (t_1, \{B(2)\}), (t_2, \{A\}), \ldots$$

denotes a prefix of an infinite execution. In the initial logical time point, the Boolean signal A is *true*, the integer signal B carries the value 1. In the second logical time point, the Boolean signal A is *false* and the integer signal B carries the value 2. Finally, in the third logical time point, the Boolean signal A is *true* and the integer signal B is not active.

We now describe how the components interact in detail.

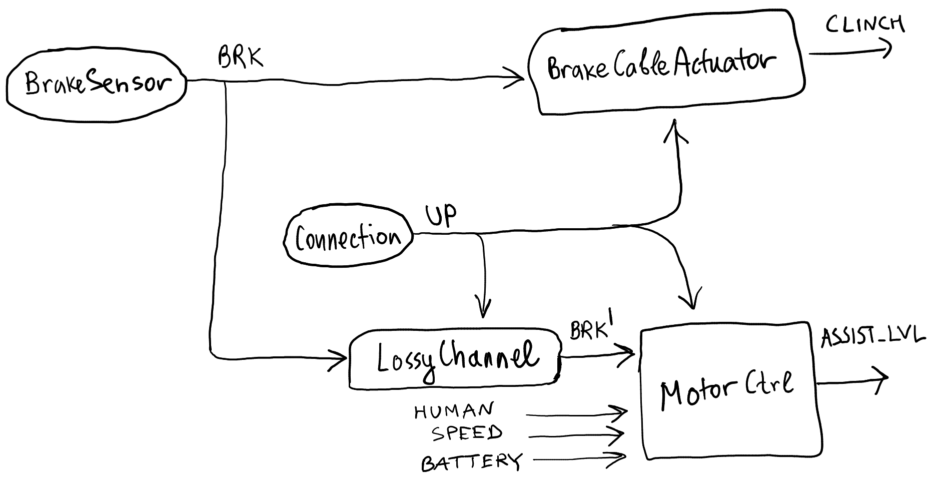


Figure 2: Interaction of the components.

Brake sensor. The brake sensor models human braking behaviour by outputting signal $\mathsf{BRK} : \mathbb{Q}_{\geq 0}$. We distinguish three cases: when $\mathsf{BRK} = 0$ no braking happens, when $\mathsf{BRK} \in (0, BRK\_SOFT]$ the soft braking mode is active, and when $\mathsf{BRK} > BRK\_SOFT$ the hard braking mode is active, where BRK_SOFT is a certain constant. As this component models the environment (human) behaviour, it is nondeterministic.

Lossy channel. This component models a wireless transmission of data. It reads the values from the braking sensor and the current status UP of the channel. When UP is

[2]Throughout this document, we use both the terms *event* and *signal* interchangeably. The term *event* is an intuitive notion that we have used in the formal verification course while *signal* is more used in the context of control or LUSTRE.

true, the component truthfully relays the value on its input to the output. When UP is false, the output may be anything. This component is nondeterministic.

Brake cable actuator. The behaviour of the actuator depends on BRK and on status UP of the connection with the motor. First, consider the case when the connection is up.

- When the human brakes softly, the brake cable actuator does not engage (thus CLINCH is false). At the same time, the motor should activate its regenerative braking. In the regenerative braking mode, the motor uses the kinetic energy of the bicycle to charge the battery.
- When the cyclist applies hard braking, the cable actuator engages the rim brake (CLINCH becomes true). Note that the motor should still apply the regenerative braking.

Now consider the case when the connection is down (UP is false). As there is no reliable connection with the motor, the brake cable actuator should activate the rim brakes to ensure manual braking. This should happen in both cases of soft and hard braking.

Motor controller. The motor controller is responsible for regenerative braking and for assisting the cyclist. The component reads the braking signal through the wireless channel, the amount of force the human applies, the current speed, the battery level, and outputs the amount of assistance $\mathsf{ASST\_LVL} : \mathbb{Q}$ that the motor should apply. When $\mathsf{ASST\_LVL} > 0$, we say that the motor should assist the cyclist, when $\mathsf{ASST\_LVL} < 0$, we say that the motor should apply the regenerative braking, and when $\mathsf{ASST\_LVL} = 0$, the motor should not be engaged. The motor controller behaves as follows. When the connection with the braking sensor is down, the motor should not be engaged. Consider the case when the connection is up. If the braking occurs, the motor controller activates the regenerative braking whose level is proportionate to the amount of braking BRK'. Otherwise, if the speed is below 25 km/h, the motor controller should provide the assistance that equals the current human force. When the speed is above 25 km/h, the motor should not provide assistance. On top of that, the motor should not provide assistance if the battery level is 0 (but the regenerative braking can still occur).

In the next sections, we design and verify (some of) these components on different levels of abstraction.

3 Lustre: design and verification of discrete-time systems

In this part we focus on the discrete-time abstraction of the system. Your task is to implement a few components in language Lustre and verify them using model checker kind2. This model checker is capable of verifying safety properties of systems working with integer and real values, and although such a model checking problem is undecidable in general, the tool is capable of verifying many interesting cases.

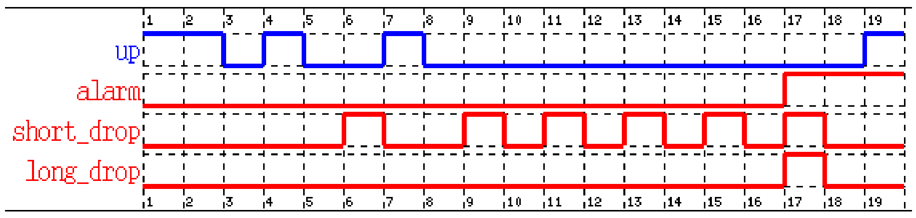
You need to implement the components `MotorController`, `BrakeCableActuator`, `LossyChannel`, and `ConnectionAlarm`.

Task L1: ConnectionAlarm

Note that the this module is *not* shown on the interaction scheme in Figure 2. This module reads the connection status UP, and raises the ALARM flag when the number of connection errors reaches a certain limit, signalling that the connection hardware is probably faulty and should be replaced. Its declaration in LUSTRE is

```
node ConnectionAlarm(up:bool) returns (alarm:bool);
```

We now explain its behaviour using the diagram below.



The module sets ALARM to true when one of the following happens:

- The number of "short drops" is ≥ 7. A short drop is a loss of communication (UP is false) lasting ≥ 2 steps.
- A "long drop" happens. A long drop is an absence of UP lasting ≥ 10 steps.

Note that a long drop is counted as a short drop as well. In the diagram above at step 6 we observe a short drop because UP is absent for 2 consecutive steps (at moments 5 and 6). For the same reason we observe a short drop at time steps 9, 11, 13, 15, 17. In step 17, we observe a long drop, since UP is absent for 10 steps (since moment 8). Observing a long drop, the module raises ALARM. Note that once the alarm is raised, it should stay high from now on forever (see steps 18, 19).

Task. Implement the module. Verify it using KIND2 against the property "once the alarm is set, it stays set forever". Your report should contain:

- the code of that module;
- the diagrams[3] illustrating the short and the long drops, and the alarms caused (a) by too-many short drops, and (b) by a long drop;
- how would you write that property in LTL? Add your LTL formula into the report;
- the outcome of the verification step[4].

[3]To be able to see `short_drop` and `long_drop` boolean flags (or whatever you name them), create a copy of module `ConnectionAlarm` which has those flags in the output.

[4]For instance, provide a screenshot.

Task L2

Implement the component

```
node BrakeCableActuator(brk:real; up:bool) returns (clinch:bool);
```

The component is inspired by the description in the intro, but slightly more involved. It should monitor the connection using `ConnectionAlarm` that you've implemented in the previous task. When the cyclist presses the brakes hard, CLINCH should be engaged. It should also be engaged when the cyclist brakes softly and there is a connection alarm. When there is no connection alarm, and the cyclist brakes softly, CLINCH should not be engaged. Note that this implies that CLINCH might be inactive for short durations when UP is false but the alarm has not been raised yet.

Task. Implement the module. Verify the following properties using KIND2:

- whenever the cyclists brakes hard, the clinch should happen;
- whenever the clinch happens, BRK is not zero.

(Feel free to add more properties.) Add to your report the code of the module and the outcome of the verification steps.

Task L3

Implement the component

```
node LossyChannel(noise,brk:real; up:bool) returns (brk_rcvd:real);
```

It is used to model the nondeterministic behaviour of the communication channel when the connection drops. When UP is false, the value BRK sent by the brake sensor can be damaged during the transmission, thus BRK_RCVD can be anything. We model this using a new input signal NOISE. The component relays BRK to the output when UP is true, otherwise it outputs anything (NOISE).

Task L4

Implement the component

```
node MotorController(brk_rcvd:real; up:bool; human,speed,battery:real)
returns (asst_lvl:real);
```

The controller should monitor the connection status using the module `ConnectionAlarm`. Note that when UP is false, the controller cannot trust the value of BRK_RCVD. The behaviour of the controller was described in the introduction, but here we will use a slightly different logic, which depends not only on the connection status but also on the connection alarm. First, if the connection alarm is set, the controller should not engage the motor neither in braking nor in assisting. Then, when the alarm is low, the controller should satisfy the following properties.

- If the connection is down, the motor cannot be used for braking (but can be used for assistance).

- There is no assistance on speeds above 7 m/s.
- There is no assistance when the battery is 0 (but regenerative braking can occur).
- When regenerative braking is engaged, ASST_LVL equals −BRK_RCVD.
- When assistance is engaged, ASST_LVL equals HUMAN.

Task. Implement a controller satisfying these properties. If you find that certain cases are not described, make a reasonable design choice. Include into report the code and explanation for your design choices (if any).

Task L5

We provide an implementation of the component

```
node System(up:bool; noise,brk,human,speed,battery:real)
returns (clinch:bool; asst_lvl:real);
```

This component represents the whole system, with inputs coming from the cyclist (HUMAN, BRK) and the environment (NOISE, UP, SPEED, BATTERY).
Task. Add and verify the following properties:

- if the connection alarm is true and the braking happens, brake-cable clinch must happen (i.e., CLINCH must be true);
- when the connection alarm is true, the regenerative braking cannot happen;
- regenerative braking occurs only if the braking is requested by the human;
- when the speed is above 7 m/s, the assistance is negative or zero;
- the assistance level is always equal to one of the following values: 0, -BRK, HUMAN.
- if the braking and assisting are both requested at the same time and the connection is up and there is no alarm, then the motor should engage the regenerative braking.

Your report should contain the code with these properties added and the outcome of the verification step.

4 Strix: synthesis of discrete-time systems

A specification is useless if it cannot be realized by any concrete implementation. There are obvious reasons why it might be unrealizable: it might require the computation of a nonrecursive function, or it might be logically inconsistent. A more subtle danger is specifying the behavior of part of the universe outside the implementor's control. This source of unrealizability is most likely to infect specifications of concurrent systems or reactive systems. It is this source of unrealizability that concerns us.

excerpt from a paper by Martin Abadi, Leslie Lamport, and Pierre Wolper.

In this part, we focus on checking realizability of specifications for the bike controller and synthesizing a model of the controller if the specification is realizable. You will use the tool Strix [5], [6] which is a tool for reactive LTL synthesis combining a direct translation of LTL formulas into deterministic parity automata (DPA) and an efficient, multi-threaded explicit state solver for parity games. In brief, Strix (1) decomposes the given formula into simpler formulas, (2) translates these on-the-fly into DPAs based on the queries of the parity game solver, (3) composes the DPAs into a parity game, and at the same time already solves the intermediate games using strategy iteration, and (4) finally translates the wining strategy, if it exists, into a Mealy machine or an AIGER circuit with optional minimization.

Your task is to express the interaction between the cyclist, the braking system and the motor in Linear Temporal Logic and use the tool Strix to verify that the specification is realizable, and if it is the case, then synthesize an operational model of the controller, i.e. a Mealy Machine.

4.1 High level description of the Controller

The motor-controller maintains a continuous stream of interaction between itself, the motor and the braking system to ensure that:

- When the cyclist is pedalling (i.e. the brakes are not activated), the motor provides *assistance* to the cyclist.
- Whenever the cyclist brakes, the braking system actives the brake-regeneration mode *quickly enough* (real-time), i.e. the motor uses the existing momentum to recharge its batteries thereby decreasing the speed of the bicycle.
- The motor assistance and the brake regeneration are separated in time, i.e. the motor doesn't assist *too-quickly* after it stops recharging.

4.2 Tasks

You are asked to use Strix to synthesize a model for the controller from a LTL specification that describes the interaction between the motor and the braking system. In your report, you will provide systematically the formulas that you have written and give a description of the model produced by Strix when the specifications are realizable. For each task and subtask, you must represent the LTL specifications that you provide as assumptions and guarantees to Strix as a JSON file. Additionally, every mealy-machine generated by Strix must be represented in a text file in the Hanoi-Omega-Automata format (HOA-format) [1]. The proper format for those types of files are provided as examples on UV. When the specifications are unrealizable, you must explain how the output of the tool can be used to understand why they are not realizable.

We will ask you to write a specification step by step following the sequence of tasks below.

Input and Output Propositions

Input Propositions You must use the proposition **brake** to model the activation of the brakes by the cyclist, i.e. **brake** is set to True whenever the cyclist applies the brakes and False if not.

Output Propositions You must use the propositions assist, recharge and idle to model the three states of the motor.

Assumptions-Guarantees You will see below that some of the properties are assumptions on the input that we receive from the environment and some of the properties are guarantees that the controller needs to enforce.

Task A: Modelling the properties of the Motor and Braking System

Task A1: Consistency Properties

1. **Motor:** The controller must *guarantee* that motor is in one and only one state at all time instances, e.g., there cannot be a run where, for example, both recharge and idle are true at the same position in the run.

TO-DO: Your task here is to express the above property in Task A1: Consistency Properties as an LTL specification. You can now use STRIX to generate a preliminary model (mealy-machine) of the controller. Note that STRIX should return REALIZABLE.

Task A2: Safety Properties

1. **Braking System:** To specify the time-delay between changing of states of the motor, the controller must *guarantee* that the motor cannot jump instantaneously between states assist and recharge, i.e the motor has to pass through the state idle anytime it wants to transition between providing assistance to the cyclist (assist) and using the momentum of the bike to recharge the batteries while the user activates braking (recharge) and vice-versa.

TO-DO:

1. Your task is to express the property in Task A2: Safety Properties as an LTL specification. You may now combine the properties in Task A1: Consistency Properties and Task A2: Safety Properties and use STRIX to generate a preliminary model (mealy-machine) of the controller. Is your model any different than the one obtained at the end of Task A1: Consistency Properties?

2. What if you add a **temporary guarantee** that the motor must visit the states assist and recharge infinitely often? Does your model look different now? Include both the models that you have generated in your report and also mention any comments that you may have about the models that you have generated in your report.

Task B: Modelling the interaction between Motor and Braking System

Task B1: Braking and Assist Functionality

1. You must now guarantee that whenever the cyclist brakes (**brake** is set to True), then the motor must, in the future (eventually), visit the state recharge.

2. You must now guarantee that whenever the cyclist stops braking (**brake** is set to False), then the motor must, in the future (eventually), visit the state assist.

TO-DO:

1. Your task is to express the property in Task B1: Braking and Assist Functionality as an LTL specification. You may now combine the properties in Task A: Modelling the properties of the Motor and Braking System and Task B1: Braking and Assist Functionality and use STRIX to attempt to synthesize a preliminary model (mealy-machine) of the controller. Does STRIX return REALIZABLE? If yes, is your model any different than the one obtained at the end of Task A: Modelling the properties of the Motor and Braking System?

2. Are you satisfied with the behaviour of your model? If not, comment on why and give an example of a trace that you are not satisfied with. What happens if you add a **temporary guarantee** that whenever the cyclist brakes (**brake** becomes True), the motor must stop assisting in the next time-step (assist should not be true in the next position)? Is the model still REALIZABLE (Hint: STRIX should return UNREALIZABLE and a counter-example in the form of a strategy for the environment to falsify the specification)? If not, you can try to **(temporarily)** remove the guarantee made in Task A2: Safety Properties and see if STRIX gives you a REALIZABLE model now. Include the two models and the counterexample generated in your report. It is important that you remember to include again the guarantees from Task A2: Safety Properties in your forthcoming tasks.

Task B2: Continuity in Brake-Regeneration and Assist

1. You must guarantee that once the braking starts and the motor is in the state recharge, the motor must continue to remain in state recharge until the cyclist stops braking, i.e., until **brake** stops being true.

2. Similarly, you must guarantee that when the braking is not activated and the motor is in the state assist, the motor must continue to remain in state assist until the cyclist starts braking, i.e., until **brake** becomes true.

TO-DO:

1. Your task is to express the property in Task B2: Continuity in Brake-Regeneration and Assist as an LTL specification. You may now combine the properties in Task A: Modelling the properties of the Motor and Braking System, Task B1: Braking and Assist Functionality and Task B2: Continuity in Brake-Regeneration and Assist and use STRIX to attempt to synthesize a preliminary model (mealy-machine) of the controller. Does STRIX return REALIZABLE (Hint: STRIX should return UNREALIZABLE and a counter-example)?

2. What if you add a **temporary assumption** that the cyclist brakes and stops braking infinitely-often? Does STRIX return REALIZABLE now?

3. The analysis above indicates that you must use the weak-until or release operator to express the correct LTL specifications. Can you explain why using the strategy of the environment that shows that the above specification was unrealizable ?

Task B3: Modelling delays in cyclist

1. Computers are quite fast these days and they are definitely faster than human reaction speed. So, one can safely assume that one timeunit (clockspeed) of the controller is much faster than human reaction speed. In our case, make the *assumption* that the cyclist cannot switch between braking and not-braking within 3 time-steps. **TO-DO**: Your task is to express the assumption as an LTL specification and combine the specifications in previous tasks to generate a model using STRIX. Does STRIX return REALIZABLE?

2. How much faster than humans do computers have to be? Find out the minimum number of time steps required for the model of the controller to be realizable. **TO-DO**: Your task is to replace the above assumption with an a suitable assumption with the minimum number of time steps and combine the specifications in previous tasks to generate a model using STRIX. For which value does STRIX return REALIZABLE?

Include both the models obtained in your report.

Task B4: Time-Constraints

1. You now examine the reaction speed of your controller. Can you guarantee that whenever the cyclist brakes, the motor must reach the state recharge within 4 time-steps? How about 3 time-steps? Additionally, find the minimum number of time-steps that the controller can guarantee, i.e. the model should be REALIZABLE.

2. Similarly, examine the guarantees that you can make when the cyclist stops braking. How fast can the motor must reach the state assist? Can it do so within 4 time-steps? How about 3 time-steps? Once again, find the minimum number of time-steps that the controller can guarantee, i.e. the model should be REALIZABLE.

TO-DO:

1. For each of these cases in Task B4: Time-Constraints, express the guarantee as an LTL specification and combining it with the LTL specifications in all the tasks in Task A: Modelling the properties of the Motor and Braking System and Task B: Modelling the interaction between Motor and Braking System to generate a model using STRIX. Include each of the models in your report. In the case that the model becomes UNREALIZABLE in any of these conditions, include the counter-example in your report.

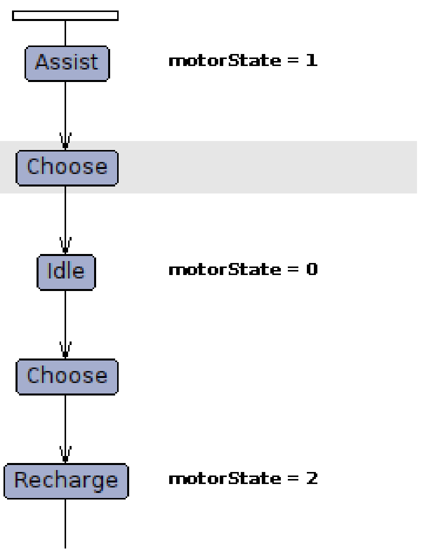
5 UppAal TiGa: verification and synthesis of timed automata

In this part, we will model the motor, brake and the battery of the bike. Then we will generate a safe and efficient strategy for the motor. For task U1 to U6, the file describing your system should be named 'uppaal_project.xml' and the associated query file should be named 'uppaal_project.q'. In task U6, the strategy should be saved in a text file 'uppaal_strategy.txt'. You are asked to create another system in task U7 to U9, which will be saved as 'uppaal_project2.xml' and the query file for it should be named 'uppaal_project2.q'. You need to submit these 5 files along with your report.

In your report, you need to briefly describe the automata you have created. You should mention the global or local variables and clocks that you have defined and what do they represent.

Task U1: Modelling the motor

You are given a UppAal TiGa project file called 'motor.xml'. This contains an automaton for the motor. The motor has three different ASST_LVLs, namely $= 0$, > 0 and < 0. We have defined an integer variable named motorState that takes values 0, 1 and 2 to represent these three ASST_LVLs.



The automaton has four states: Assist, Idle and Recharge to represent these three values of motorState and a state called Choose. Initially the automaton is in Assist state. To change the value of motorState, the transitions from the states Assist, Idle and Recharge to the state Choose are taken first. From the state Choose, there are three transitions back to the states Assist, Idle and Recharge available which changes motorState.

For example, notice the trace given in the right. The initial state is Assist and the initial value of motorState is 1. The transition to the state Choose is taken first, then when the transition to the state Idle is taken, value of motorState becomes 0.

Create a project file named 'uppaal_project.xml'. Import the automaton from 'motor.xml' in it.

Add a clock variable timer1, which resets every time the transition from Assist to Choose is taken. What does this clock variable describe?

Task U2: Modelling the brake

Either the brake is released, or the cyclist applies the brake. We can add a Boolean variable brake that is true when the brake is pressed and false when the brake is released.

Similar to the automaton for motor, we can create an automaton to model the brake that has three states: Released, Pressed and Choose that changes the value brake. Initially the automaton is in Released state.

Note that the brake is controlled by the cyclist, so the edges in this automaton should be uncontrollable.

Define the variable brake and create the automaton for the brake.

Task U3: Modelling the battery

The battery level can either be high or low. We can add a Boolean variable batteryLevel to represent the battery level and create an automaton for the battery that has three states: High, Low and Choose. Initially the automaton is in High state.

With added guards, we can make sure that the battery can discharge (i.e. battery level can go from high to low) only when the motor is assisting. Similarly, the battery can charge (i.e. battery level can go from low to high) only when the motor is recharging. Battery levels should not change when the motor is idle.

Note that discharging of battery may depend on external factors, so the edges should be uncontrollable.

Define the variable batteryLevel and create the automaton for the battery.

Task U4: Modelling the scheduler

We want to make sure batteryLevel, brake and motorState update exactly once every 1 time unit. We need to create an automaton with four states. The initial state is BatteryUpdate, from which it synchronizes with the automaton for battery to update the battery and goes to the state BrakeUpdate. From this state, it synchronizes with the automaton for the brake to update the state of the brake and goes to the state MotorUpdate. From there, it synchronizes with the automaton for the motor to update the state of the motor and goes to the state Wait.

The automaton should have the following properties:

- The scheduler automaton should spend 0 time unit at BatteryUpdate, BrakeUpdate and MotorUpdate.
- At state Wait it should stay for exactly 1 time unit and go back to the state BatteryUpdate.

To make sure the updates always happen in that specific order, we also need to make the states Chose in previous three automata committed, so that the transitions from Chose always happen before any other transitions of automata running in parallel.

The composition of four automata (motor, brake, battery and scheduler) should generate traces like in figure 3. Note the synchronizations used in the system are drawn in red.

Task U5: Creating an observer automaton

We want to measure the duration for which the brake is pressed. We can do it by creating an automaton with 2 states and a clock variable timer2 such that if $\text{brake} = 1$, timer2 denotes the time since the cyclist started pressing the brake. If $\text{brake} = 0$, we do not care

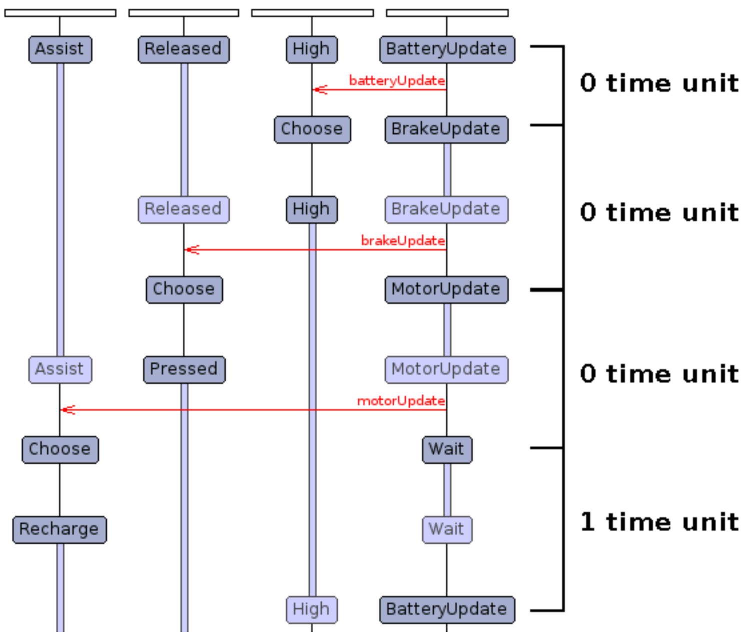


Figure 3: Example of a trace showing how the scheduler interacts

about the value of timer2. This way we can formulate the condition 'the brake is pressed for at least x time units' as 'brake == 1 and timer2 $\geq$ x'.

Create such an automaton that implements it.

Task U6: Finding a strategy

In our model, we are allowing all possible transitions from the state choose in the motor. We need to create a strategy to restrict some actions (This will be done by adding guards in the next task). We want to make sure while in the state Wait in the automaton for scheduler, these following conditions holds:

Condition 1: The motor should not assist when

(a) the brake is pressed, or

(b) the battery is low.

Condition 2: The motor should not engage recharge when

(a) the brake is not pressed, or

(b) the battery is high, or

(c) the time since the motor has stopped assisting is less or equal to 2 time units.

These conditions are also satisfied when the motor always stays idle. But this is not a practical model for the motor. So we need to add some more conditions to make the motor efficient.

Condition 3: The motor should not be idle in the following scenarios:

(a) When the brake is not pressed and the battery is high.

(b) When the battery is low and the brake is pressed for at least 5 time units.

Write a query that checks if there is indeed a strategy for the controller that makes sure that these three conditions are always satisfied. You need to use the clocks defined before.

In your report write the query formula.

Use 'verifytga' to find such a strategy and save it in a file 'uppaal_strategy.txt'. You can do this by running 'verifytga' with argument '-w 0 -c -s' and then saving the output in a text file.

Task U7: Creating a controller

Create a copy of your 'uppaal_project.xml' file. Name it as 'uppaal_project2.xml'. In this new file, in the automaton for motor, add appropriate guards in the edges from the state Chose such that following conditions always hold:

- Conditions 1 to 3 in task U6
- **Condition 4:** There is no deadlock

You can consult the strategy generated in the previous task to find appropriate guards.

For each condition, verify that they are indeed satisfied. Write the query you used to check this in your report.

Task U8: Verification of properties

Verify the following property : "Always, if the brake has been pressed for more than 1 time unit, the motor engage recharge". Report the query you used. If the property is false, report a counterexample. (You can generate a counterexample trace by running 'verifytga' with argument '-t 0'.)

Task U9: Modelling the rim-brake

As you may have figured out while answering previous task, due to condition 2(b) and 2(c), the motor does not always engage recharge, even when the brake is pressed for more than 1 time unit. In this case, we want the rim brakes to be activated to ensure manual braking.

Create an automaton for the rim brakes that has two states: Released, and Pressed. Initially the automaton is in Released. Using binary synchronization channels between the automaton for motor and the automaton for the rim brakes make sure the following happens:

- Whenever the motor assists or engages recharge, the state of the automaton for the rim brakes should change to Released.
- If the brake is pressed but the motor cannot engage recharge, the state of the automaton for the rim brakes should change to Pressed.

You may need to add more transitions in the automaton for motor.

Now 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". Report the query you used. If the property is false, report a counterexample.

6 Xcos: simulation of dynamical systems

Your goal is to simulate in Xcos a bike trip from Parc Ten Reuken to Parc de la Héronnière. Figure 4 shows a map with its elevation profile.

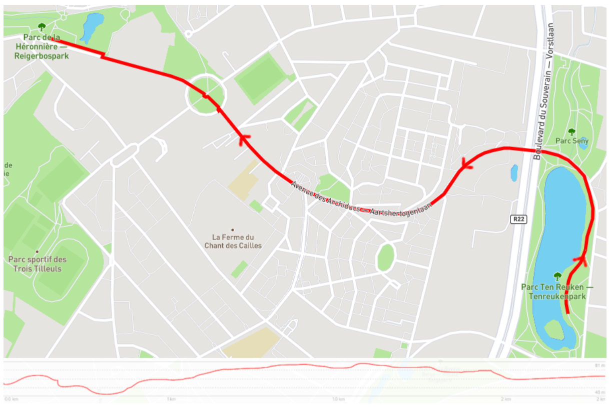

Parc de la Héronnière — Reigerbospark
Parc Sef
La Ferme du Chant des Cailles
Parc sportif des Trois Tilleuls
Parc Ten Reuken — Tenreukenpark
Boulevard du Souverain — Vorstlaan
R22

Figure 4: Track from Parc Ten Reuken to Parc de la Héronnière; total length 1800m.

In the folder on Xcos you will find a project template. The template contains all the necessary components, some already implemented and some are left for you to implement. The main components are: battery, human, electric assistance, and bike dynamics. You need to implement the last two components.

The bike dynamics is described by the following equation (assuming the speed $v > 0$):

$$m\frac{dv}{dt} = F_e + F_h - mg\sin\alpha - C_{rr}mg\cos\alpha - kv^2,$$

where F_e is an electric assistance, F_h is a human effort, $mg\sin(\alpha)$ accounts for the road gradient, and the last two components are the rolling-resistance force and the air-resistance force. We use the following values: $k = 0.25$, $m = 70$, $C_{rr} = 0.003$.

Task X0. Answer, with explanations, the following questions.

(a) Is the function for $\frac{dv}{dt}$ Lipschitz-continuous? (When answering the question (a), you can drop the multipliers m, mg, $C_{rr}mg$, k in the function.) Here we assume that F_e, F_h, α are inputs. Start your answer with the definition of the Lipschitz-continuity from Rajeev's book. Then state what do you really want to show (continuous or not continuous?), and show it from the definition. Hint: you can rely on the following claim: if a function $f(x, y) = g(x) + h(y)$ can be expressed as a sum of two (or more) functions over individual variables, and one of those functions is not Lipschitz-continuous, then $f(x, y)$ is not Lipschitz-continuous.

(b) Let us assume that $F_e = 0$, $\alpha = 0$, and $F_h = 275exp(-0.5v)$. Using Xcos, draw a plot "derivative vs. speed" (speed on the X axis and derivative on the Y axis); save your work into `x0.zcos`. Find an equilibrium speed of the bike using the diagram. (Do not forget the component $C_{rr}mg\cos\alpha$.) (Hint: it is a very strong rider.) Are these equilibrium speeds stable? Asymptotically stable? Justify the answers using your diagram.

Your report should contain the answers together with the plots, and the file `x0.zcos`.

Task X1. Implement the behavioural model of a rider. The rider pedals on speeds below 10m/s and brakes on higher speeds. See the block "human behaviour" in the template for more details. Draw the diagram "effort vs. speed" with human effort and braking on the Y axis and speed on the X axis. To this end, create the file `x1.zcos` containing only the rider model and the scopes necessary for the diagram. Your report should contain the diagram "effort vs. speed" and the file `x1.zcos`.

Task X2. Implement the dynamics model of the bike, as described by the differential equation above. Simulate your model, observe the diagrams. How much time does the cyclists need to reach the destination? By dividing the total distance (1800m) by the total time, calculate the average speed of the bike without the electrical assistance. Add your answers and the diagrams to the report. Save your solution into `x2.zcos`.

Task X3. (Copy the file `x2.zcos` into `x3.zcos` and work on it.) Implement the basic model of electrical assistance. It repeats the human effort ($F_e = F_h$) when the battery level is above 0.4 (40%), and otherwise provides $F_e = 0.75F_h$. The assistance should be turned off on speeds above 7 m/s. Additionally, the regenerative braking should be activated when the cyclist requests it, and the braking overrides the assistance (has a higher priority). What is the new journey time and average speed? Add your answers and simulation diagrams to the report. Save the solution into `x3.zcos`.

Task X4. (Copy the file `x3.zcos` into `x4.zcos` and work on it.) Implement the advanced model of electrical assistance, which additionally has a hysteresis behaviour around 6...7 m/s. This means: if the assistance is 0 and the speed is below 7 m/s, then we turn on the assistance. Once the speed reaches 7 m/s, turn off the assistance until the speed falls below 6 m/s, then the assistance turns on again, and so on. You might want to use the block `hysteresis` to implement this behaviour. How do diagrams for tasks X3 and X4 differ? Is the travel time/speed are affected? Add the answers and simulation diagrams to your report. Save your solution into `x4.zcos`.

References

[1] Tomáš Babiak, František Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker, and Jan Strejček. The hanoi omega-automata format. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 479–486, Cham, 2015. Springer International Publishing.

[2] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 years. *Softw. Pract. Exp.*, 41(2):133–142, 2011. URL: https://uppaal.org/.

[3] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. URL: https://www-verimag.imag.fr/The-Lustre-Programming-Language-and.

[4] https://scilab.org. Scilab and Xcos. URL: https://scilab.org.

[5] Michael Luttenberger, Philipp J. Meyer, and Salomon Sickert. Practical synthesis of reactive systems from LTL specifications via parity games. *Acta Informatica*, 57(1-2):3–36, 2020. URL: https://strix.model.in.tum.de/, doi:10.1007/s00236-019-00349-3.

[6] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. Strix: Explicit reactive synthesis strikes back! In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 578–586. Springer, 2018. doi:10.1007/978-3-319-96145-3_31.