

Introduction to Timed Automata

Lab session – UPPAAL

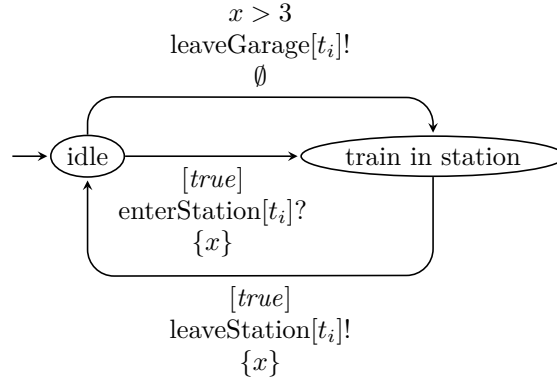
Gaëtan Staquet

8th September 2026

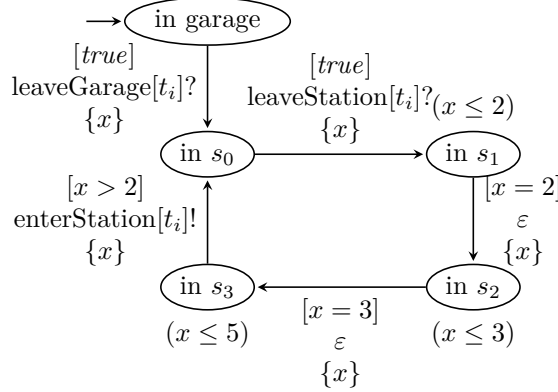
This lab session is split into three parts.

1. Part I introduces UPPAAL's interface and constructs the roller coaster example of the slides. How to write properties in UPPAAL's language is also explained.
2. Part II focuses on modeling a system that manages sharing ressources between processes, while ensuring that two processes cannot get the same ressource at the same time.
3. Finally, Part III already provides the model and requests to write (interesting) properties for the system.

The first part can be skipped if you would rather discover the tool by yourself.



(a) The station.



(b) A train t_i .

Figure 1: Timed automata for the station and the trains, as seen in the course.

Part I — Aboard the UPPAAL roller coaster

The goal of this part is to discover UPPAAL’s interface, how to model a system, and how to write and verify properties. To do so, we will build an UPPAAL model for the roller coaster example seen in the course, and repeated in Figure 1. If you would rather play yourself with the final model and its properties, a file `rollerCoaster.xml` is provided alongside this PDF.

We first construct the model inside UPPAAL, and explore it through the symbolic and concrete simulators. Finally, we write some properties of the system, and verify them.

We will mainly use the “Selection”, “Add Location”, and “Add Edge” tools here.

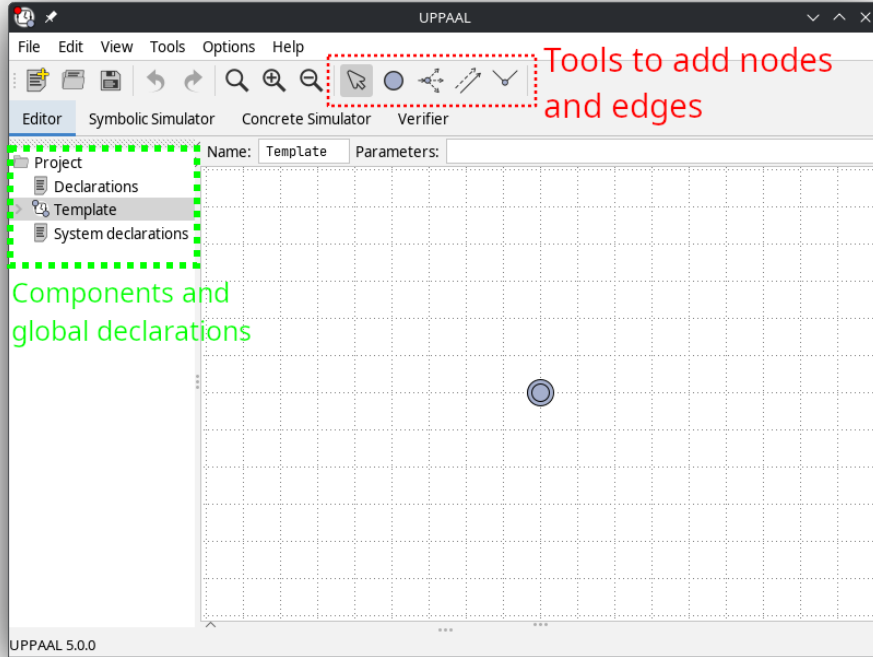


Figure 2: UPPAAL’s new project.

I.1. Creating a model in UPPAAL

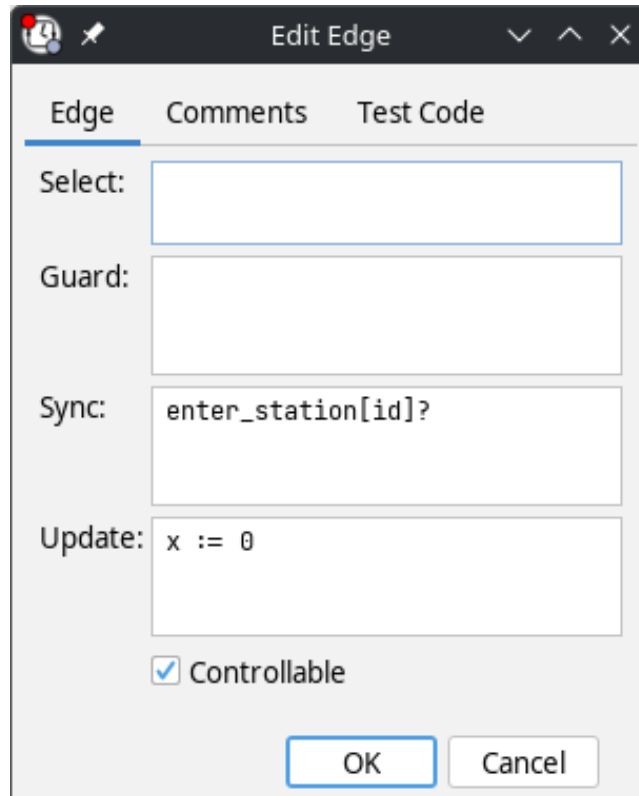
When starting a new project in UPPAAL, the program is in the “Editor” mode, and we get a timed automaton composed of a single component, called **Template**, as shown in the following figure. The initial state of each component is indicated by a doubly circled state. See Figure 2

The tools above the editor allow adding new states (called **locations** in UPPAAL) and edges, while the panel on the left lists the current components and the global declarations of the system.

I.1.1. Constructing the station

First, rename **Template** into **Station**. Then, add a new state (either by using the tool above the editor view, or by middle-clicking on the grid). We can now create an edge from the initial state to the new one. By double-clicking on this edge (while in the selection tool), a new window appears.

We want this edge to wait for an **enter_station** event from some train t_i , have no conditions over clocks, and reset the clock x (see Figure 1). So, in the **Sync** field, write **enter_station[id]?** (each train will have its own specific action, indexed by the



The image shows a software window titled "Edit Edge". It has a dark header bar with a small icon on the left and window control buttons (minimize, maximize, close) on the right. Below the header, there are three tabs: "Edge", "Comments", and "Test Code". The "Edge" tab is currently selected and highlighted with a blue underline. Inside the "Edge" tab, there are four text input fields arranged vertically. The first field is labeled "Select:" and is empty. The second field is labeled "Guard:" and is empty. The third field is labeled "Sync:" and contains the text "enter_station[id]?". The fourth field is labeled "Update:" and contains the text "x := 0". Below these fields, there is a checkbox labeled "Controllable" which is checked. At the bottom right of the window, there are two buttons: "OK" and "Cancel".

Edge	Comments	Test Code
Select:		
Guard:		
Sync:		enter_station[id]?
Update:		x := 0

☒ Controllable

OK Cancel

Figure 3: Window for editing an edge.

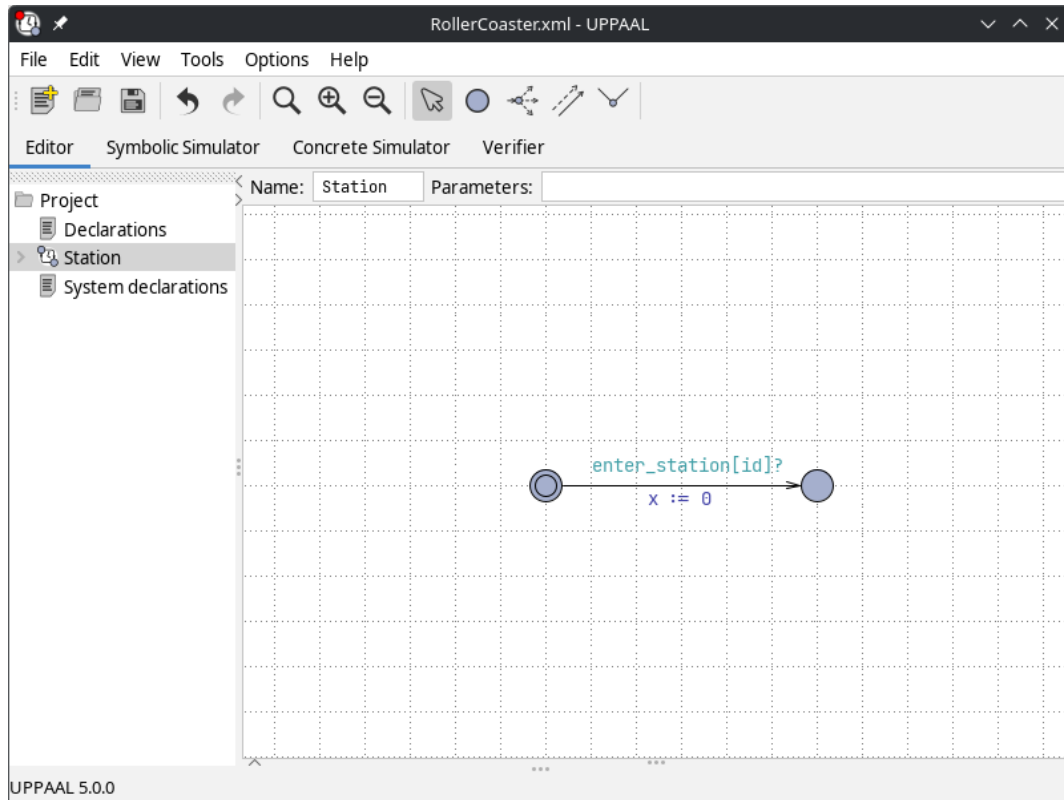


Figure 4: After editing the first edge.

variable `id`, the identifier of the train), and, in the **Update** field, `x := 0` (or `x = 0`, both notations set x to zero), as in Figure 3. After closing this window, you should obtain something similar to Figure 4.

Checking the syntax of the system. Before adding more components and edges, let us verify whether what we have done so far is valid. To that end, either press `F7` or, in the menu at the top of the window, select “Tools”, then “Check Syntax”. A pop-up should appear with an error message. You can close it, as the most interesting details are given at the bottom of the main window, as seen in Figure 5.

The errors.

- The action `enter_station`, the train identifier `id`, and the clock x are unknown.
- There is a problem in the declaration of the whole system (as indicated by the `/system` position).

Indeed, we have not declared anything, and we renamed the component without touching the system declarations. Let us fix all of that!

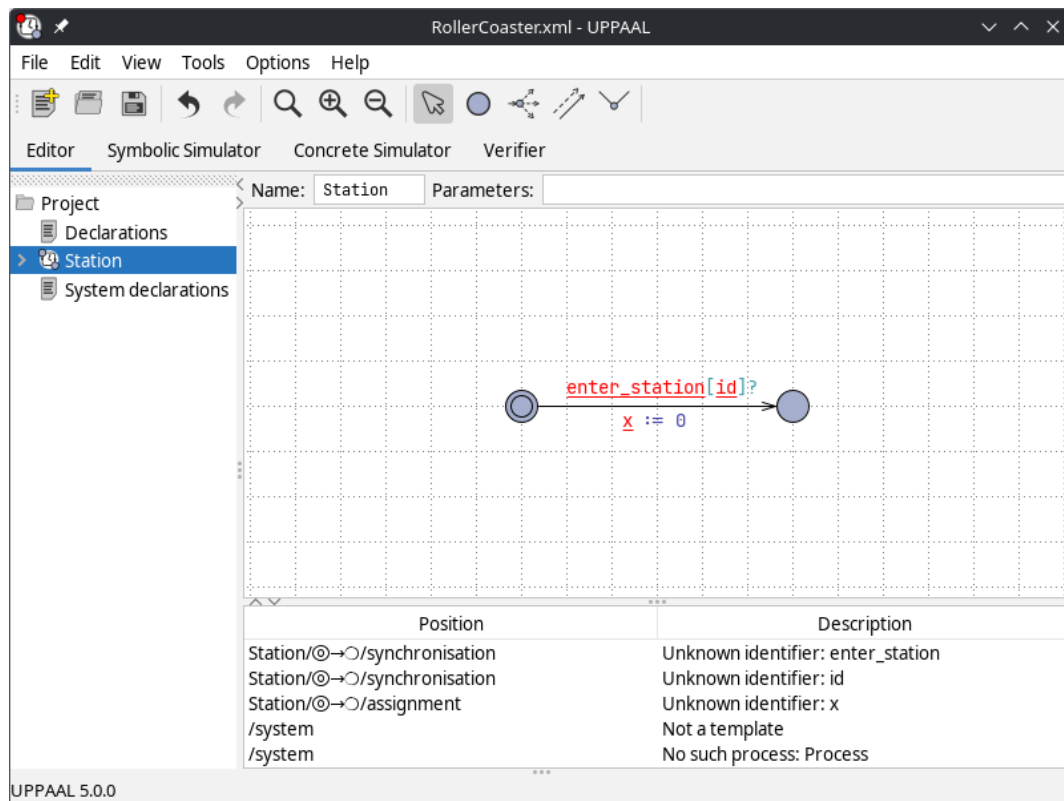


Figure 5: Errors after checking the syntax.

Declaring global variables. On the left, click on **Declarations**. A text file opens. This is where we will declare variables that can be accessed by every component of the system. In our case, we need to define the following elements:

- the number of trains we desire;
- the type of the identifier for a train; and
- the action `enter_station`, one for each train.

This is done by adding the following lines in the text file (see Figure 6):

```
const int N_TRAINS = 3;
typedef int [0, N_TRAINS - 1] train_id_t;
chan enter_station[N_TRAINS];
```

Observe that UPPAAL's syntax is close to C's. The first line declares a constant, and fixes the number of trains to 3. The second line declares the type that will be used to identify each train, *i.e.*, an integer between 0 and the number of trains minus 1. The last line declares a **channel** `enter_station` for each train. (UPPAAL's **channels** coincide with the synchronized actions used in Figure 1.)

Declaring variables local to a component. Expand the `Station` to reveal the **Declarations** text file specific to that component. Open it, and add the line

```
clock x;
```

to declare a clock named *x*. All the variables declared in this file will only be visible from `Station`.

Declaring variables local to an edge. We now fix the fact that the variable `id` is unknown on the edge we created. This variable is meant to designate which train is entering the station. Thus, `id` must be of the type `train_id_t` defined earlier. Moreover, we have to declare `id` local to the edge, as the value of `id` may vary each time the edge is taken.

Edit the edge again (by double-clicking on it) and write `i : train_id_t` in the **Select** field, as in Figure 7. This will create a local variable `id` of type `train_id_t`, as desired.

If you now check the syntax of the model, there should remain two errors, related to `/system`.

Defining the system. Now, open the **System declarations** file. This file is meant to declare how to combine the components to form the whole system. Replace the file content by `system Station;` to use the single component we have defined so far.

If you now check the syntax of the model, you should no longer obtain error messages.

Finishing the station. Your task is to now finish defining the `Station` component.

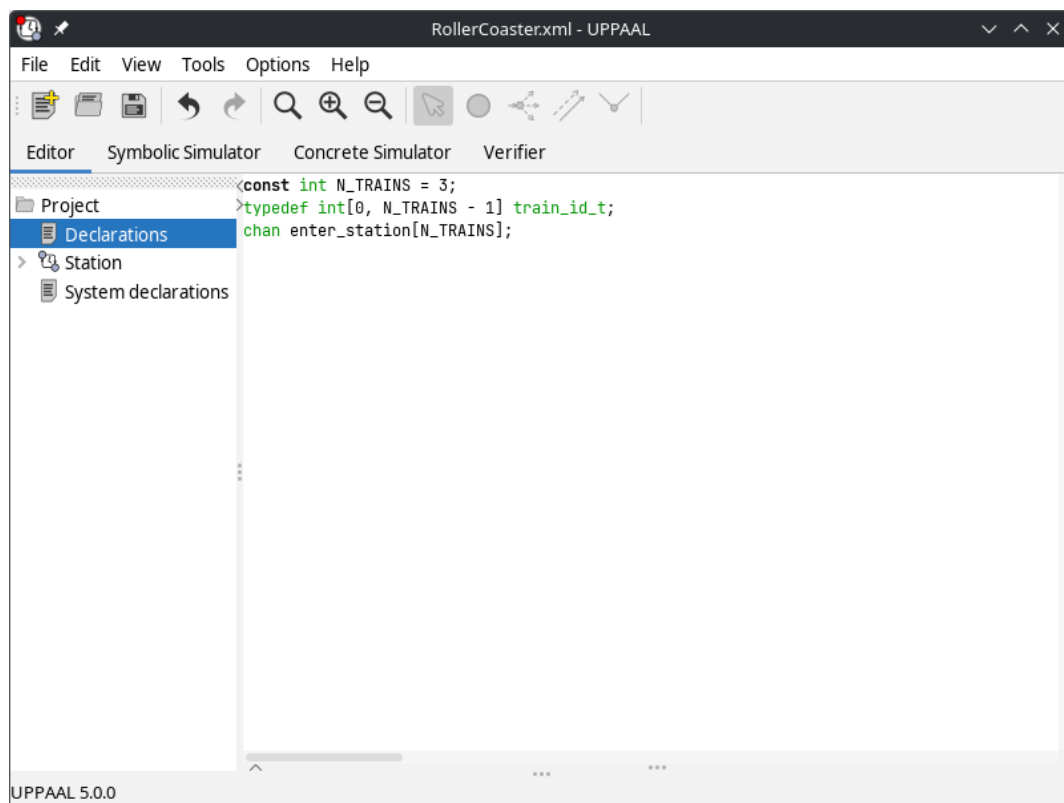
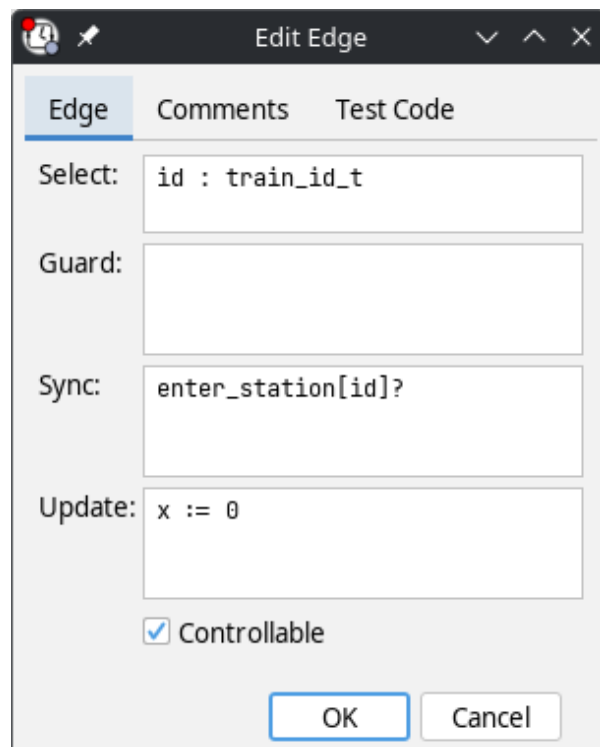


Figure 6: The global declarations.



The image shows a software dialog box titled "Edit Edge". It has three tabs: "Edge" (selected), "Comments", and "Test Code". The "Edge" tab contains four text input fields: "Select:" with the text "id : train_id_t", "Guard:" which is empty, "Sync:" with the text "enter_station[id]?", and "Update:" with the text "x := 0". Below these fields is a checkbox labeled "Controllable" which is checked. At the bottom right are "OK" and "Cancel" buttons.

Field	Value
Select:	id : train_id_t
Guard:	
Sync:	enter_station[id]?
Update:	x := 0

☒ Controllable

OK Cancel

Figure 7: The properties of an edge with the **Select** field.

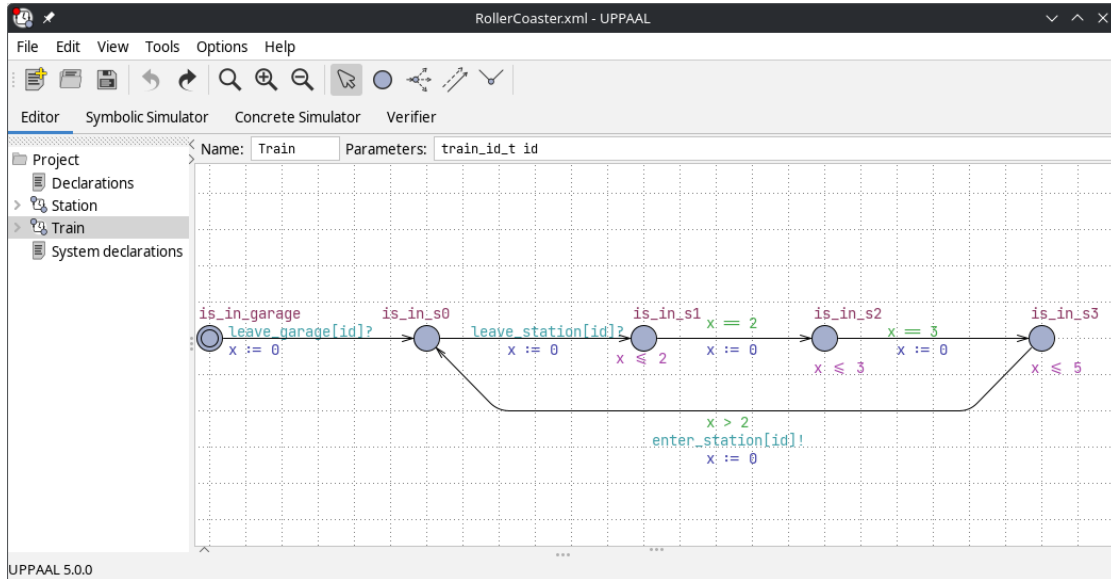


Figure 8: The **Train** component.

Exercise I.1. Define the two missing edges for the station. You may have to add more global variables in order to get a syntactically correct automaton.

Hint: the field **Guard** when editing an edge can be used to define constraints over the clock x .

In order to obtain a readable model, you can add nails (allowing you to visually define edges in multiple steps) by left-clicking on an empty part of the grid, when creating a new edge.

I.1.2. Constructing a train

Add a new component **Train** (in the **Edit** menu on top of the window). Each train must be identified by its identifier. Luckily, UPPAAL can automatically generate the identifiers. All we need to do is indicate that **Train** expects a parameter and its type. So, in the field **Parameters** above the grid, write `train_id_t id`.

Exercise I.2. Completely construct the **Train** component by performing the following tasks.

- Add four nodes in **Train** and define the edges needed to reproduce Figure 8.
- In the **System declarations**, replace the line we previously wrote by `system Station, Train;` to add the newly defined **Train**.
- Check the syntax of the system, and fix the errors, if any arises.

Hint: double-clicking on a node opens an editing window to set the name and the invariant of the node.

Make sure to write the correct names for the nodes! They will be used in the properties.

Figures 9 to 11 give the final version of each component, and of the global and system declarations. Check that you have something similar to these figures.

I.2. Symbolic and concrete simulators

Now that we have completely defined the components, we can manually explore the system, in order to have an intuition on the possible runs, and the constraints over the clock that appear alongside it.

Symbolic simulator. The first simulator we explore is the symbolic one, that abstracts the exact timing of the action. Instead, it works by creating **zones** and checking which edge can be followed according to the current zone.

Open the **Symbolic Simulator** (above the grid). You should see your system on the right, and the enabled transitions on the left. If you open the panel that is between them (by dragging the vertical bar), you can observe the description of the zone.

By double-clicking an enabled transition on the left (or clicking on it, then pressing the **Next** button), the simulator will move one step, updating the enabled transitions and the zone.

Exercise I.3. Manually build a symbolic run that ends in a configuration where **Train(0)** is in the state **is_in_s0** and **Train(1)** is in the state **is_in_s2**.

Concrete simulator. While the symbolic simulator abstracts the exact timing of the events, the concrete simulator takes into account the precise timing of each event. That is, this simulator creates actual run of the timed automaton. The interface permits to select precisely at which point in time an action must be performed.

Exercise I.4. Manually build a concrete run that ends in a configuration where **Train(0)** is in the state **is_in_s0** and **Train(1)** is in the state **is_in_s2**.

I.3. Properties in UPPAAL

One tab is left unexplored so far: the **Verifier** one, allowing us to write properties the system must satisfy. The whole syntax for the (symbolic) queries of UPPAAL is described in the official documentation¹. We here focus on a subset that will be used throughout the rest of the lab session.

Queries in UPPAAL are stated over **paths** (or executions) of the system. For instance, one can express that, along every path, two trains can never be in the same section; or

¹https://docs.uppaal.org/language-reference/query-syntax/symbolic_queries/

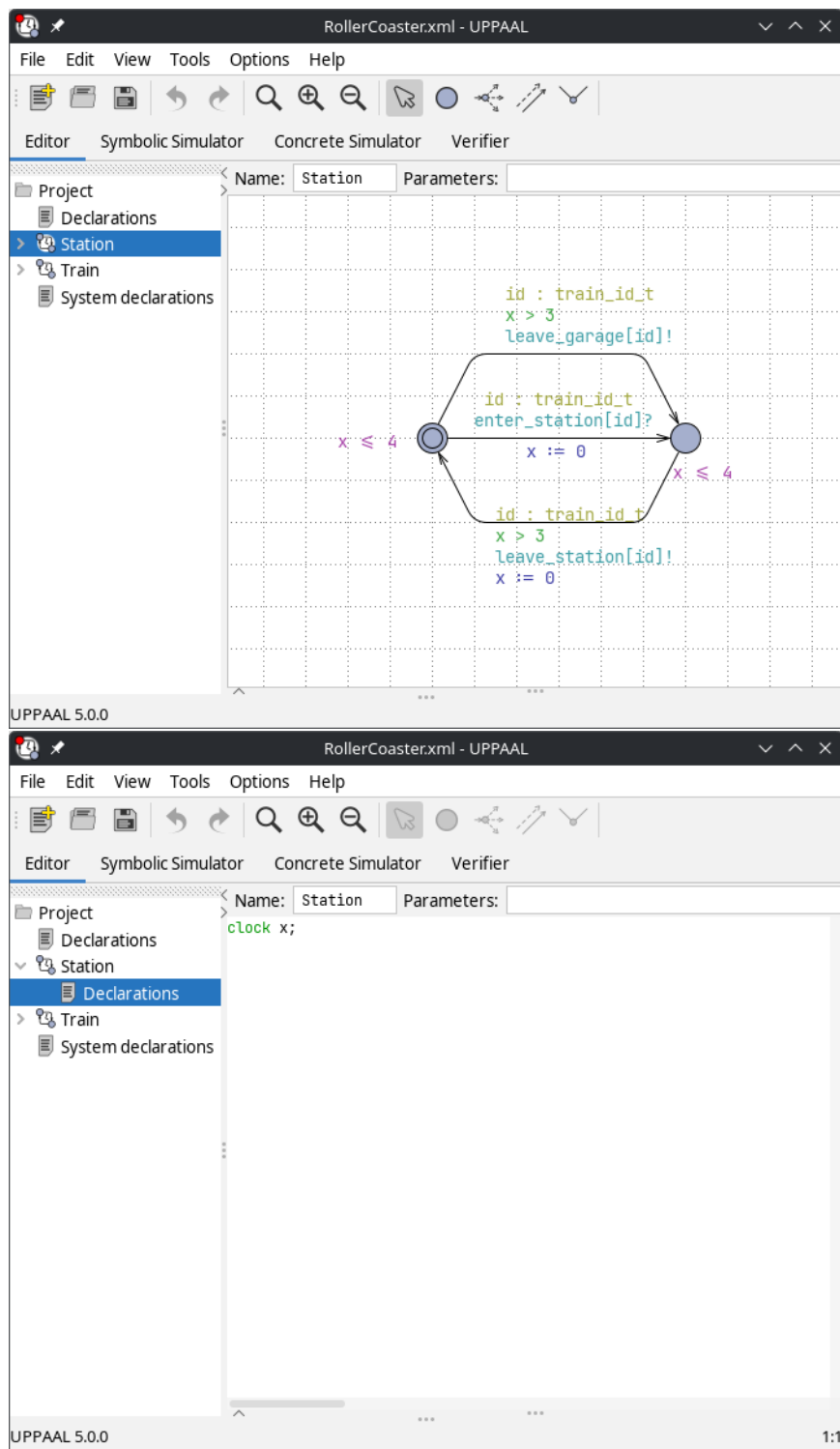


Figure 9: Complete **Station**.

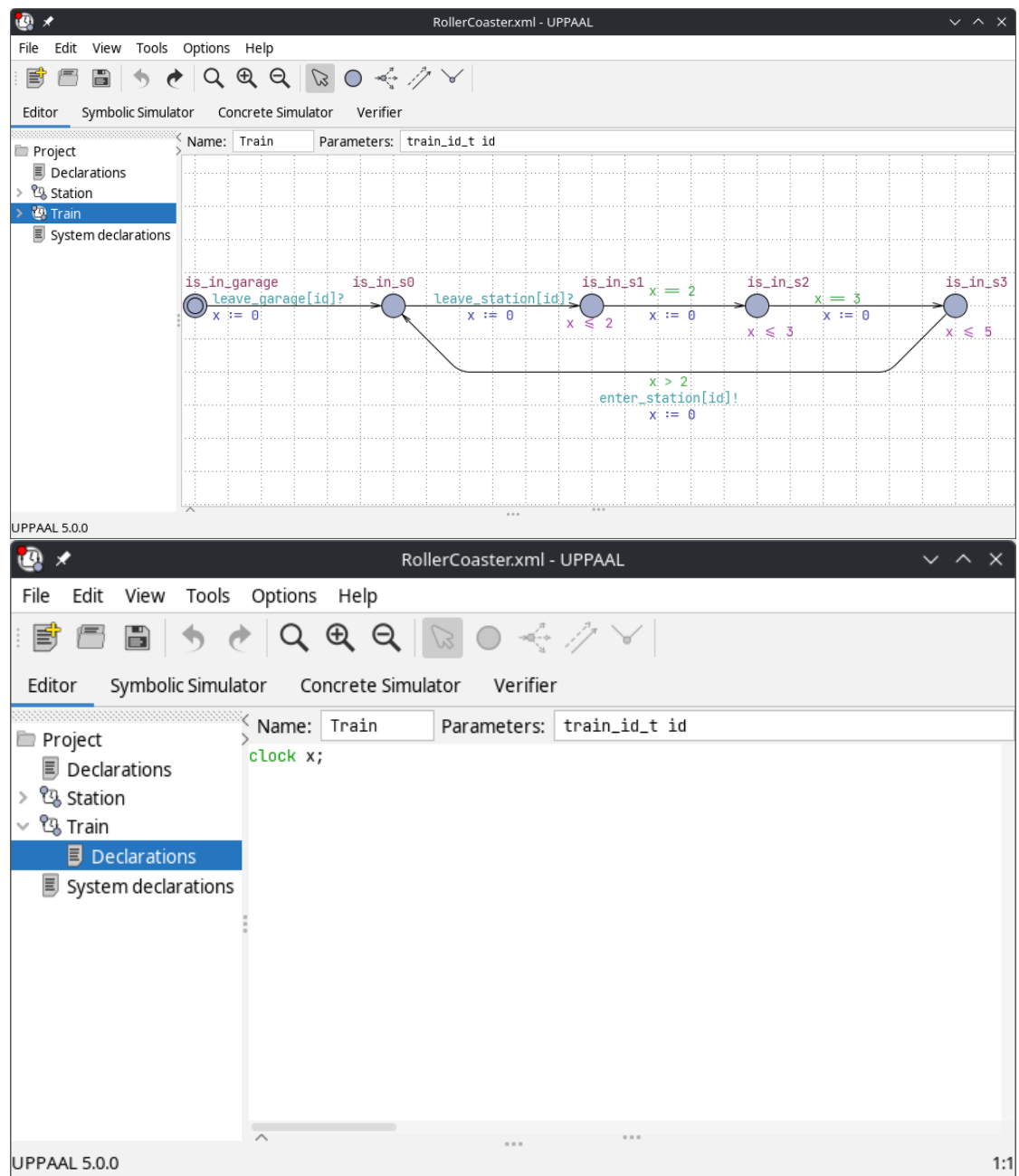


Figure 10: Complete **Train**.

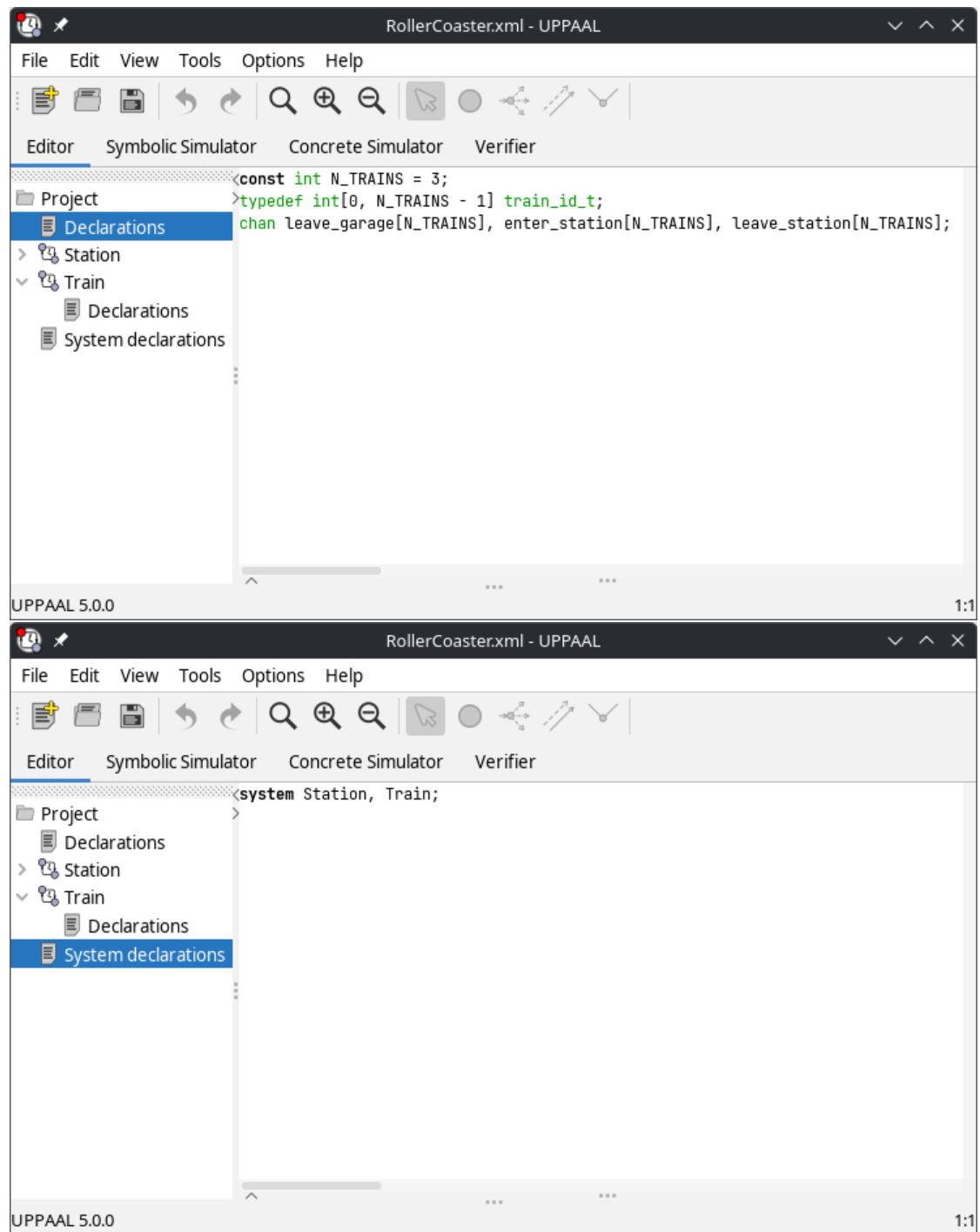


Figure 11: Global and system declarations.

that there exists a path in which all trains have left the garage. That is, some properties are over all paths of the systems, while others will only require the existence of one good path.

Orthogonally to this, queries can express that the property must hold at every position (configuration of the system) along the path, or that there exists at least one good position. Hence, there are four main types of queries that we will use here.

Globally For all paths, on all positions along the path, an expression is satisfied:
`A[] expression.`

Eventually For all paths, there exists a position such that an expression is satisfied:
`A<> expression.`

Potentially always There exists a path such that, for every position, an expression is satisfied: `E[] expression.`

Possibly There exists a path and a position along it such that an expression is satisfied:
`E<> expression.`

Expressions. Inside queries, an **expression** must evaluate to true or false. They can be used to express requirements over the reached state, integers, clock values, and so on. Here, we will only work with the states of the components.

Example I.5. One way to express that the train of index 0 is in the sector s_0 is (using UPPAAL's syntax):

```
Train(0).is_in_s0
```

This can be generalized to any train:

```
exists (i : train_id_t) Train(i).is_in_s0
```

That is, we require the existence of a train's index i such that the component `Train(i)` (i.e., the train of index i) is in s_0 .

Conjunctions can be written using **and** (or `&&`), disjunctions using **or** (or `||`), and negations using **not** (or `!`). A shortcut to write implications is also defined: $a \Rightarrow b$ can be written as `a imply b`. Comparaisons are written as usual in programming languages: `a == b`, `a != b`, `a < b`, and so on. Parentheses can be used to force evaluation in a certain order, or to reduce ambiguity.

Example I.6. The following expression states that, if there exists a train in s_0 , there cannot be any other train in s_0 . That is, there is at most one train in s_0 .

```
exists (i : train_id_t) Train(i).is_in_s0 imply
(forall (j : train_id_t) i != j imply not Train(j).is_in_s0)
```

Full queries. Let us now illustrate a globally, an eventually, and a possibly queries.

Example I.7. In the previous example, we saw how to express that two trains cannot be in s_0 at the same time. Here, we add the information that this must hold at any position alongside any path of the system to get the full query:

```
A[] exists (i : train_id_t) Train(i).is_in_s0
  imply (forall (j : train_id_t) i != j imply not Train(j).is_in_s0)
```

We also saw how to write that there is a train in s_0 . This cannot be true on every position of a path (as we want s_0 to be empty to let another train come in), but, on every path, we must find a good position. So:

```
A<> exists (i : train_id_t) Train(i).is_in_s0
```

Finally, let us express that it must be possible to empty the garage:

```
E<> not (exists (i : train_id_t) Train(i).is_in_garage)
```

This does not hold for every path of the system (as the attraction can work with a single train, i.e., two trains are still in the garage), but there exists an execution of the attraction that satisfies it.

Potentially always queries are not illustrated here, but they follow the same structure.

“Leads to” queries. There is a fifth type of queries that will be useful in this lab session: **leads to** queries. They are used to express that, if an expression p holds at some point, then q must eventually hold. That is, it is equivalent to $A[] (p \text{ imply } A<> q)$. However, nested quantifiers on paths are not permitted. So, as a way to keep this often used property, UPPAAL has the syntax $p \text{ --> } q$ (note the double dash in the arrow).

Example I.8. To express that, if the train zero reaches s_1 , then the same train will eventually get to s_2 , write:

```
Train(0).is_in_s1 --> Train(0).is_in_s2
```

Unfortunately, it is not allowed to write

```
forall (i : train_id_t) (Train(i).is_in_s1 --> Train(i).is_in_s2)
```

as the $-->$ operator must be the main operator. That is, if we want to check that the property is true for every train explicitly, we have to duplicate the query and change the 0 into whatever index you desire. However, remember that all trains share the same model. So, if it holds for the train zero, it holds for all of them.

Exercise I.9. Add the following queries in the verifier tab (one by one). For each one, explain in a few words what is being expressed, and click on the **Check** button to verify that the model satisfies the property (the bubble at the end of the line will

turn green).

- `A[] not deadlock`
- `A[] exists (i : train_id_t) Train(i).is_in_s0 imply (forall (j : train_id_t) i != j imply not Train(j).is_in_s0)`
- `A[] exists (i : train_id_t) Train(i).is_in_s1 imply (forall (j : train_id_t) i != j imply not Train(j).is_in_s1)`
- `A[] exists (i : train_id_t) Train(i).is_in_s2 imply (forall (j : train_id_t) i != j imply not Train(j).is_in_s2)`
- `A[] exists (i : train_id_t) Train(i).is_in_s3 imply (forall (j : train_id_t) i != j imply not Train(j).is_in_s3)`
- `A<> exists (i : train_id_t) Train(i).is_in_s0`
- `A<> exists (i : train_id_t) Train(i).is_in_s0 and exists (j : train_id_t) Train(j).is_in_s2`
- `E<> not (exists (i : train_id_t) Train(i).is_in_garage)`
- `Train(0).is_in_s1 --> Train(0).is_in_s2`

Note: all properties should hold. If you have an error, check that you wrote the components correctly and/or call me.

Part II — Modeling – Resources and processes

Open the file `processesResources.xml` in UPPAAL. This project already contains two components: `Resource` and `Process`. `Resource` is already modeled, while `Process` only contains three nodes, without any edges. Moreover, all global and local declarations are missing.

II.1. Filling in the gaps

This system should manage resources shared by multiple processes. A resource can be used by at most one process at a time, *i.e.*, two processes cannot have access to the same resource at the same time.

One way to guarantee this is to let a process **attempts** to obtain a resource. If, after two units of time, the access request is not granted, the process goes back to its idle mode and can try again at any time. If the access is granted, the process can perform its work and release the resource after some time. Notice that a resource cannot be `in_use` for more than one unit of time.

Exercise II.1. Add the missing global and local declarations, and the missing edges in `Process`.

Ensure that your model is syntactically correct.

II.2. Does it work?

In the **Verifier** tab, multiple properties are provided.

Exercise II.2. For each property, explain in a few words what is checked. How does it help to prove that the system works as expected?

Exercise II.3. Check each property against your model. Fix the errors, if any occur. **Hint:** you can enable automatic generation of counterexamples in **Options**, then **Diagnostic Trace** and select **Shortest**. If a “for all” property fails, a counterexample will be provided in the symbolic simulator. Note that any “there exists” property that *succeeds* will also provide a trace in the symbolic simulator, to showcase the existence.

Part III — Properties – Amusement park

For this part, open the file `AmusementPark.xml` in UPPAAL. That example uses functions and data structures to manage the state of an amusement park, composed of multiple attractions, and in which multiple users evolve.

III.1. Understanding the system

Your first task is to build an intuition on how the model behaves. You are not requested to read the code of the functions used on the edge, nor how the data structures are implemented. Their names should be sufficient to grasp the general ideas. You can use the simulators as much as you want.

You can ignore the `Park` component, as its role is to make sure that all attractions open at the same time.

For simplicity, we assume that each attraction has a single train, and that it can never break down.

Exercise III.1. Explain the behavior of a user. What do they do in the amusement park? How many users can be in the park?

Exercise III.2. Give an intuition on how an attraction works. How do they manage the queue of users? How many users can board a train? How long does a ride last? How do users unboard the train?

Hint: a state marked with a **C** is called **committed**: when the system is in a committed state, time cannot move forward, and the next transition **must** involve an edge from a committed state.

III.2. Properties

Five properties are already provided in the **Verifier** tab:

- We never reach a deadlock, *i.e.*, the amusement park cannot get stuck.
- An error never occurs inside the queues (for instance, we do not try to remove an element from an empty queue).
- The park always eventually opens. That is, along any execution of the system, the `Park` eventually reaches its state `is_open`.
- The last two properties check that the model is reasonable, by verifying that a user cannot be in two attractions at the same time, and that, if the park is at full capacity, then users have to wait (*i.e.*, queues are not empty).

Exercise III.3. Write the following properties in UPPAAL's language.

- In the **Users are happy** section:
 - When `User(0)` enters a queue, `User(0)` eventually boards a train.
 - When `User(0)` is on board a train, `User(0)` eventually leaves the train and the attraction.
 - On any execution of the system, there exists a moment in time where `User(0)` is on board a train.
- In the **Attractions do not get stuck** section:
 - When `Mars` is waiting for more users to fill the train or for enough time to have elapsed, `Mars` will eventually run its train.
 - When `Mars`'s train is running, `Mars` will eventually enter the state `is_emptying_train`.
 - When `Mars` has reached its state `is_emptying_train`, `Mars` will go to its `is_open` or its `is_waiting` state.
 - There exists an execution in which both `Mars` and `Bermudes` run at the same time.

Exercise III.4. Increase the number of users in the park and run the verifier again. What happens? Why?