Verification of a SysML Railway Specification with a Translation to UPPAAL

Wijtse Rekker

Supervised by Djurre van der Wal

LaTeX Template "The Legrand Orange Book" by Mathias Legrand [21]

UNIVERSITY OF TWENTE

February 2021

Contents

List of Tables

List of Figures

1. Introduction

When technological systems in for example the field of computer science or electrical engineering grow large it can become difficult to ensure that the behaviour of the system is correct. Especially if the system is not yet created and all that is available is documentation and design of the system. You can write many tests and try out every test case you can think of, but it is difficult to know for sure that a certain requirement of the system always holds. This is where model checking can help out.

In model checking you take an existing system and recreate its behaviour in a formal model. These models can have different forms. In this research we focus on automata. These are directed graphs where the vertices represent different states of the system and the edges represent transitions between these states which can be seen as actions. Due to the formal nature of the models, they can be simulated. Additionally, model checking tools provide property checking languages in which it is possible to write specific requirements that describe behaviour of the system. The model checking tool can then take that requirement and formally prove that the requirement is satisfied or give a counter example which proves that the requirement is not satisfied.

Examples of model checking tools are mCRL2 [12] and UPPAAL [19]. Both tools have their own way of specifying their models, but both modelling languages describe automata. While mCRL2 specifies models using process algebra in a textual form, UPPAAL has a graphical model editor in which you can create vertices and connect them via edges. To give an example application, mCRL2 and UPPAAL were used to help verify a pacemaker developed by Vitatron. This was done in the research of Wiggelinkhuizen et al. [32]. A pacemaker is a device that helps people who have a heart rate disorder called arrhythmia. It is surgically connected to the heart so it can send stimuli via pulses to help maintain an adequate heart rate. Because a pacemaker has to adapt to all possible heart rates and different cases of arrhythmia, the firmware consists of a complex combination of collaborating processes. Due to the fact that the device influences a vital organ of a persons body it is crucial that its behaviour is correct. Therefore, it is incredibly useful if the behaviour of the pacemaker can be formally proven correct.

Another example of systems where the correct functioning of those systems is crucial for the safety of people are systems in railway engineering. In these include systems such as railway junctions and level crossings. The Dutch railway company ProRail and Germany's DB Netze are working together with the University of Twente and the Eindhoven University of Technology on the verification of an upcoming standard for railway signalling equipment interfaces [23]. This project is called FormaSig [11]. The standard is developed by the EU-level organisation EULYNX [9] that consists of the collaboration between 13 Infrastructure Managers. They created a dialect of SysML which we call EULYNX in the rest of the paper. This semi formal modelling language is used to specify the standard. To help with verifying these models the contributors of FormaSig proposed to create a translation [3] to the formal model checking tool mCRL2.

While the translation from EULYNX to mCRL2 is still currently in development, the developers found that the simulation trace explorer of mCRL2 is difficult to work with when applied to the models of EULYNX. It is mainly hard to trace back model elements to their corresponding EULYNX components when looking at the state history of a counter example to a requirement.

UPPAAL, on the other hand, provides a graphical simulation trace explorer, which gives a clear overview of the state of the model at every point in the history of the trace. Therefore, we found that an additional translation to UPPAAL could be useful. Additionally, the accessible property checking language of UPPAAL is beneficial for the project, so possibly railway engineers can specify and verify their own requirements in the future. Next to that we can potentially use the time system in UPPAAL for implementing the timed elements of EULYNX. These elements are not yet specifically implemented in the translation to mCRL2. Finally, an additional translation can also help verify the correctness of the other translation. However, this is not included in the scope of this project.

In this paper we create and evaluate a translation to facilitate the use of UPPAAL in the verification of models part of the EULYNX standard. First, we introduce EULYNX, UPPAAL, and mCRL2 to provide some background information. Here we also make a comparison between UPPAAL and mCRL2 to see what UPPAAL can provide that mCRL2 does not offer. Next in this research we determine which interpretation of SysML and EULYNX we should adhere to. Here we look at what fits best with respect to the goals for the translation, the interpretation used in the mCRL2 translation, and UPPAAL as target modelling language. Next, we create the translation, while taking inspiration from existing work on similar translations to UPPAAL. After that, we evaluate the translation in terms of correctness, usability, and performance. The correctness of the translation is tested by validating the behaviour of small test EULYNX models using UPPAAL requirements. These small models each cover a small feature of the EULYNX framework, and together they cover EULYNX as a whole. We test the usability of the output models by applying it to an existing EULYNX specification. Here we aim to check a variety of different requirements to determine what is possible to validate using the output models of the translation in UPPAAL. We do not intend to fully prove that the EULYNX model under test is correct. Lastly, we evaluate the performance of the output models in UPPAAL, and propose some ways in which the translation can be improved.

We found that an interpretation of SysML using cycles to synchronise the whole model is the best option for our translation, as it enables the centralisation of the orchestration of the processes. This allows us to manage the processes that model the state machines with additional manager processes, which keeps the main processes more clear and readable as they do not contain a lot of synchronisation logic. The decision to use direct memory sharing for the communication between state machines followed from this, because the publication of port values to other state machines is part of the cycle. This can be implemented in UPPAAL by copying values from source variables to target variables, hence the direct memory sharing.

Next we created our own translation, using the existing input DSL of the mCRL2 translation as a base for the translation. We could also reuse some preprocessing steps that they created to eliminate for example overlapping internal block diagram definitions. We took inspiration from existing translations of UML and SysML state machines to UPPAAL, but we could not completely use existing work as the features of EULYNX differ from those of SysML and the interpretations of SysML state machines themselves are not the same everywhere.

In the validation of the translation we found it to be correct with the exception of a few elements that have been implemented in a simplified way that enables the translation to still be useful. While this is not a formal proof we are still confident in the correctness of the translation, but a formal proof is a useful addition that could be looked into in future work.

In terms of performance we found that it is not yet possible to check complete models that are part of the upcoming standards developed by EULYNX, because of performance issues. However, we are able to check some requirements on isolated components of these models. The expressiveness that the UPPAAL requirement language gives in combination with the output models of the translation is found to be sufficient for the verification of basic requirements on the models of EULYNX. To improve the performance and therefore increase the usefulness of the translation we propose three enhancements that can be implemented in future work. These improvements focus on how the state space of the output model can be reduced such that UPPAAL is better able to verify the requirements of the model.

All in all the translation proved to be useful from our usability study, and it was found to be correct with confidence. However, there is still room for improvement in the performance of the output model. Additionally, a formal correctness proof can help with validating the translation with stronger confidence.

2. Research questions

In order to gain additional insights in the models of EULYNX next to those that mCRL2 presents, we will also use UPPAAL to verify these models. To be able to do that reliably and frequently, we need to create an automatic translation tool which translates models from the EULYNX standard to the modelling language used by UPPAAL. From this goal we derive the following main research question:

How can we create a jEULYNX to UPPAAL translation which can be used on models of railway signalling equipment interfaces?

To help evaluate the outcome of the main research question we try to answer the following sub questions:

a) Which interpretations of EULYNX fit the goals of the translation to UPPAAL best, with regard to the usability, model complexity, and time constraints?

Next, we create the translation, after which we try to answer the following questions.

b) Is the translation to UPPAAL correct?
c) How well can the output models be used to verify model requirements?
d) How well can the input models of the translation scale in size and complexity such that UPPAAL is still able to check them?

2.1 Clarification

With these extra questions we guide the research further by understanding how we can create a suitable translation for UPPAAL, and evaluating the quality of the translation.

a) Which interpretations of EULYNX fit the goals of the translation to UPPAAL best, with regard to the usability, model complexity, and time constraints?

Here we look at different interpretations of EULYNX and the functionalities of UPPAAL. Translations of other UML state machine dialects to UPPAAL will also be evaluated for this purpose. We compare the benefits and possible drawbacks of the different interpretations, along with how it can be implemented in UPPAAL. Because the model simplicity is an important factor, we also look at how the different interpretations could effect the complexity of the output model. This is important because it makes the models and therefore the simulation traces easier to read. In short, the criteria for a good interpretation are that the intended model behaviour is achieved, and that the output models are easy to read. These points are discussed in the first part of Chapter 5.

b) Is the translation to UPPAAL correct?

The correctness of the translation is important, because the purpose of the translation is to test the behaviour of models. If the behaviour of a model is incorrectly translated, the translation loses its credibility. A formal proof would give a more complete indication of correctness, but this was too large of a task to also do in this project. Therefore we try to achieve some level of certainty by testing the different components in many scenarios. These test cases are developed after evaluating existing SysML specifications. Each scenario covers different edge cases of a specific EULYNX element in a model. This model is then verified with a number of properties that check for the correct behaviour or in some cases exclude incorrect behaviour. The application and results of this process can be found in in Chapter 6.

c) How well can the UPPAAL requirement language be used in combination with the output models to verify model requirements?

With question c) we want to evaluate the usability of the output models by composing a list of different types of model requirements and checking if we can verify them using UPPAAL. With this this we want to test to what extent it satisfies the needs of railway engineers. This is important to know because it indicates if it can be used in practise. As case study we use components of the Point model [28], because it provides a realistic scenario for us to test on. We determine which properties and types of properties we want to check, and if and how they can be specified in UPPAAL. This process is discussed in Chapter 7.

d) How well can the input models of the translation scale in size and complexity such that UPPAAL is still able to check them?

In answering question d) we want to find out how fast the state space of the output model expands beyond UPPAAL's exploring capabilities. This can be done by increasingly taking more components of a large example model like the Point model [28]. To get a more fine grained result we can create simple example models that can easily be expanded to create a larger state space. Producer consumer models could be used for this purpose. As a measure for how difficult the model is to check, we take the execution time for checking a certain property. This question is partly discussed in Chapter 7 and Section 10.1.

3. Related work

There exist some papers already out there that propose a translation from certain types of UML state machines to the model checking tool UPPAAL. Huang et al. propose an MDE-based translation algorithm that generates an UPPAAL model from a MARTE model [15]. MARTE models are SysML models with a few extensions like time and probability. In the paper they apply their translation to a Railway-Control System as a case study. In their approach they first createded a meta-model for MARTE state machine diagrams and created a meta-model for UPPAAL models. These meta-models can be seen as the input and output types of the translation. This general approach could also be useful in the creation of our translation. An important difference between MARTE models and EULYNX models is that the state machines in MARTE communicate via global variables and use events only in the form of named triggers, while in EULYNX they communicate via port connections and events can also be triggered from expressions evaluating to true.

Muniz et al. created the tool TANGRAM (Tool for Analysis of Diagrams) which can translate UML diagrams to UPPAAL models [26]. The main design of the tool revolves around the CORBA Component Model. This model adds extra functionality to UML that takes care of managing components during their execution time. The tool is demonstrated on a small case study used in train control systems. While they also do not use boolean expression based events, their centralised event system in the UPPAAL model can be partially applied in our translation. Instead of boolean expression based events, they have events that are defined solely with a label. These events can be seen as external events triggered by the environment of the model, and are handled as such.

In the research of Knapp and Merz [18], they create a tool called HUGO which mainly generates code from UML state machines. As an extra functionality it can also export to UPPAAL. In this export feature it flattens the whole model into a single process. This was done because their focus is on maximising performance rather than model readability.

There are also some papers which cover other facets of UML. For example in the research of Cui et al. [7] they propose a translation from UML timing diagrams to UPPAAL, and apply it to a small coffee machine example.

The paper 'Verifying Liveness in Supervised Systems Using UPPAAL and mCRL2' [24] by Markovski and Reniers uses the tool Supremica to translate their source model to both UPPAAL and mCRL2. The source models are not close to UML. The model that they use models a supervisory controller. With the generated models they compare the two tools, and give a translation of requirements written in the specification language of UPPAAL to the one of mCRL2. They find that UPPAAL fits better as target for their translation than mCRL2. The reason for this is the way UPPAAL deals with state information opposed to mCRL2. In UPPAAL you can easily access state variables in both the modelling language and the property checking language. In mCRL2 you have to put the state information on events in order to be able to work with it.

UPPAAL has been applied to many case studies. We list a few of the case studies mentioned in the source paper of UPPAAL by Larsen et al as examples. [19]. Many of these case studies take a look at some form of protocols, since these can contain time critical elements.

The Audio/Video Protocol developed by Bang and Olufsen has been studied using UPPAAL. This protocol is used to allow multiple audio/video devices to communicate with each other over a single bus by sending messages. In this case [14] they found an error in the protocol, and proposed a solution to fix this problem.

A Collision Avoidance Protocol implemented on an Ethernet like framework was also studied using UPPAAL [16]. The protocol aims to prevent collisions of messages while minimising the delay of the messages. Using UPPAAL they were able to prove that the protocol completely prevents collisions and that the minimum message delay has been reached.

Another case study is the Gear-Box Controller made by the company Mecel AB for modern cars [22]. It waits for signals given by the surrounding system that request a gear change. The controller then makes the gear change. UPPAAL was used to prove 46 properties that were derived from the design requirements of the system. All properties were satisfied.

As relevant previous work part of this project we consider the two published papers regarding the verification of models from the EULYNX standard in mCRL2. The first paper by Luttik [23] discusses a proof of concept of the translation to mCRL2 in the form of a case study on the Point [28] model of the EULYNX standard. The model is translated manually, and the usability is evaluated by deriving nine safety requirements and verifying them in with mCRL2.

Continuing this research, Bouwman et al. propose an automatic translation to mCRL2 in a following paper [3]. They find that mCRL2 provides the right functionality to model the behaviour of SysML state machines in an elegant way. Which means that they could model the behaviour of SysML state machines without a lot of extra boiler plate model code.

4. Background

In this chapter we introduce and provide a basic understanding of the relevant tools and specifications of this research. First we discuss the features and semantics of the modelling language used in the EULYNX standard. Then, we describe the tools UPPAAL and mCRL2. After that we evaluate the differences between the two tools, and establish the points where they complement each other. Lastly, we look at how we could translate

4.1 EULYNX

EULYNX [9] is a modelling language made for designing and specifying systems in railway engineering. Development of this standard started in 2014, initialised by railway infrastructure managers of 6 European countries. It is a visual standard to create graphical models. These models contain for example use-case diagrams, sequence diagrams, object relationship diagrams, and state machine diagrams. Because EULYNX is used to design and specify systems in a graphical way, it is not possible to formally verify them with existing tools. Therefore we want to translate these models to models that can be read and verified by formal model checking tools.

The parts of EULYNX that are relevant to this project are the state machines, and how they can be combined to form complex multi process systems. The state machines are built with basic components like vertices and transitions between those vertices. These are extended with features such as guards and effects. To help describe the modelling features of EULYNX we take a look at an example. This same example will also be used in the description of the other modelling tools. The example models 4 Vikings that are on one side of a bridge. The goal is to all get on the other side of the bridge, but they can only cross the bridge while holding a torch. There is only one torch available and it can be held by a maximum of two vikings. Lastly, every viking has its own maximum speed at

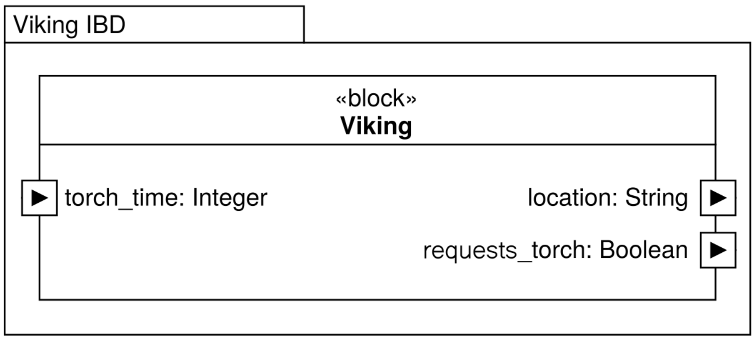


Figure 4.1: EULYNX Viking IBD

which they can cross the bridge.

We first take a look at the general definition of the Viking diagram. It is shown in Figure 4.1. The Viking diagram has three variables of which the `torch_time` is an input variable, and the other two are output variables. This represents the data flow between the different state machines. The `torch_time` is the amount of time the viking is allowed to hold the torch for at the moment. The location variable lets other block diagrams know on which side of the bridge the viking is at the moment, and the `request_torch` variable makes sure other processes know if the viking wants the torch or not.

Now for the internal logic of the viking, we look at the state machine specification. This is shown in Figure 4.2. The viking starts in the `NEAR_SIDE` state. The first line of the contents of this state represents the code that is executed on entry of the state. This sets its output variables to their initial values. On the next two lines the internal behaviour of the state is described. When the `torch_time` equals zero, it makes a non-deterministic choice between setting the `request_torch` variable to true or false. The transition between the `NEAR_SIDE` state and the `MOVING_TO_FAR_SIDE` state has a guard that checks if the torch time is more than zero. When this is the case it can take that transition. The next transition contains a timeout event. It first needs to wait for the amount of time it takes to cross the bridge, as specified with `after(torch_time)`. The same but the other way around happens to get from the `FAR_SIDE` state to the `NEAR_SIDE` state.

The block diagram definition of the torch state machine is shown in Figure 4.3. It has as properties the `torch_location`, `torch_time`, and `requestCount`. The property `torch_location` keeps track of on which side of the bridge the torch is. `torch_time` is zero if no viking is holding the torch or the maximum speed of the vikings that are currently holding the torch. Lastly, `requestCount` keeps track of how many vikings currently want the torch. The internal helper functions the Torch uses in its state machine are listed below the properties. The definition for these can be found in Appendix A. The input and output variables are listed below the list of internal operations. For each viking the torch IBD has as input variable the location of the viking and if they request the torch or not. As output variables it has for each viking the amount of time for which they are allowed to hold the torch.

The composition of all the IBDs are shown in Figure 4.4. This shows the dependencies between the different processes, and the general flow of data.

The full state machine of the torch process is shown in Figure 4.5. The transition between

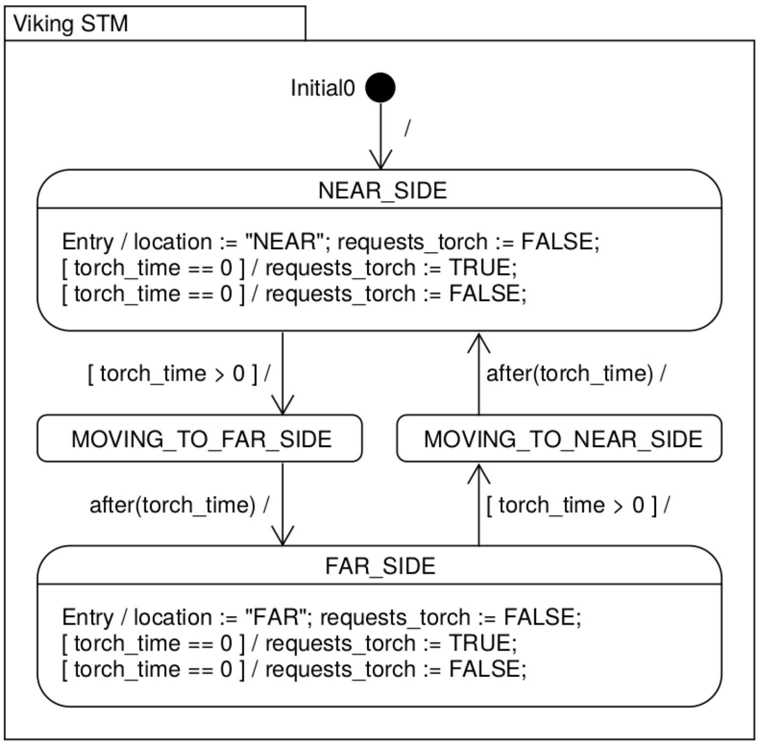

Viking STM
Initial0
/
NEAR_SIDE
Entry / location := "NEAR"; requests_torch := FALSE;
[torch_time == 0] / requests_torch := TRUE;
[torch_time == 0] / requests_torch := FALSE;
[torch_time > 0] /
after(torch_time) /
MOVING_TO_FAR_SIDE
MOVING_TO_NEAR_SIDE
after(torch_time) /
[torch_time > 0] /
FAR_SIDE
Entry / location := "FAR"; requests_torch := FALSE;
[torch_time == 0] / requests_torch := TRUE;
[torch_time == 0] / requests_torch := FALSE;

Figure 4.2: Viking state machine in EULYNX

`CHOOSING` and `WAITING` has both a guard, and an effect. The guard checks for how many vikings on the same side as the torch currently request the torch. The needed amount if the torch is on the near side is two, and one for the other side. It is limited to these amounts, because traveling with only one person from the near side to the far side does not help with solving the problem of getting all the vikings to the other side. The same holds for traveling with two vikings from the far side to the near side. When the guard evaluates to true, it can take the transition, and the effect will set the torch time output variables for each viking that is currently allowed to have the torch. After that it waits until all the vikings have moved. The diamond like state represents a choice vertex. When all the vikings are on the far side, it transitions to the `SUCCESS` state, and otherwise it goes back to the `CHOOSING` state and the whole process is repeated.

The programming language used in the guards, effects, and internal operations is the ASAL language [2]. ASAL stands for Atego Structured Action Language. It contains only basic statement types:

- Assignment
- Return
- Operation calls
- If statements
- Multi statements

The expression types contain two interesting types. The PulsedIn and PulsedOut types are equivalent to the boolean expression, but indicate a different physical signal type. In EULYNX they mean that once a value is published as a pulse, it can only be consumed/read once.

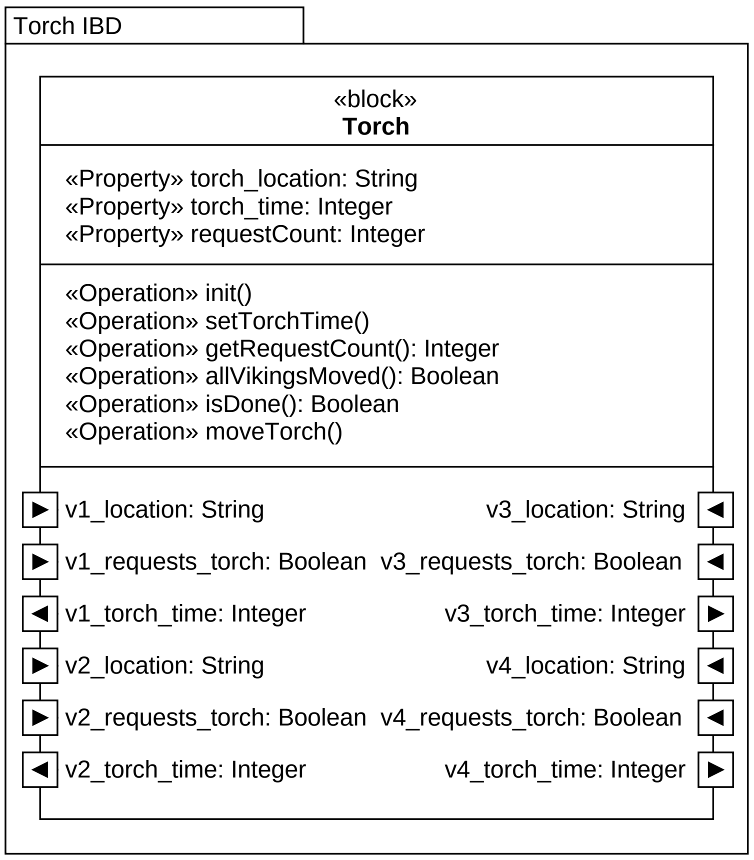

Torch IBD
«block»
Torch
«Property» torch_location: String
«Property» torch_time: Integer
«Property» requestCount: Integer
«Operation» init()
«Operation» setTorchTime()
«Operation» getRequestCount(): Integer
«Operation» allVikingsMoved(): Boolean
«Operation» isDone(): Boolean
«Operation» moveTorch()
v1_location: String
v1_requests_torch: Boolean
v1_torch_time: Integer
v2_location: String
v2_requests_torch: Boolean
v2_torch_time: Integer
v3_location: String
v3_requests_torch: Boolean
v3_torch_time: Integer
v4_location: String
v4_requests_torch: Boolean
v4_torch_time: Integer

Figure 4.3: EULYNX Torch IBD

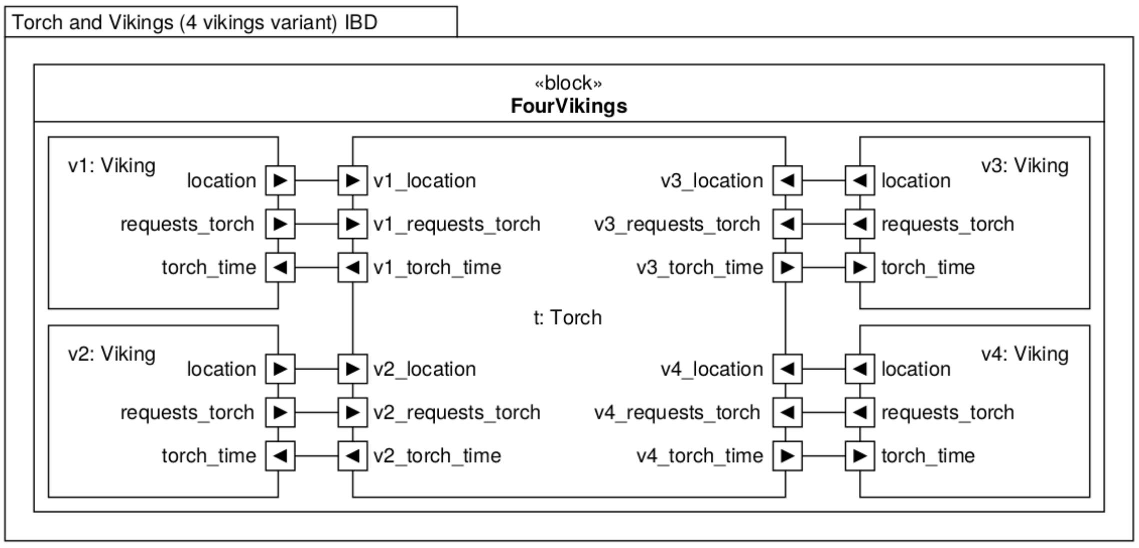


Figure 4.4: Vikings and Torch IBD composition in EULYNX

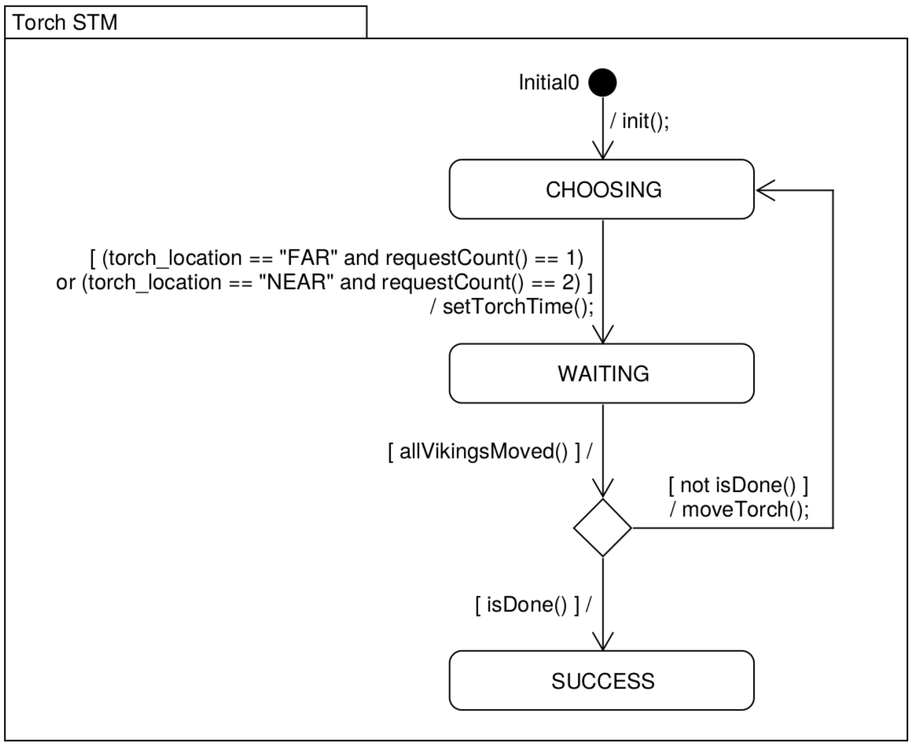


Figure 4.5: Vikings and Torch IBD composition in EULYNX

4.2 UPPAAL

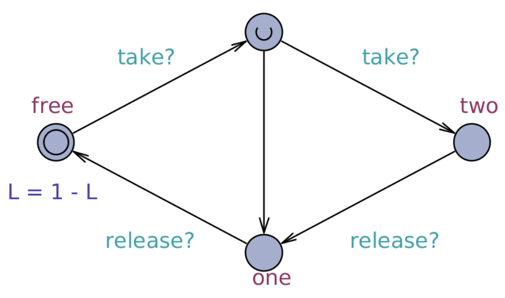
UPPAAL [19] is a model checker which allows the verification of complex systems. It supports non-deterministic timed state machines which can consist of multiple separate processes. These can communicate through variables and channels. In order to check properties of its models UPPAAL has its own specification language. The two main design criteria of the model checker are efficiency and ease of use. To achieve efficiency, UPPAAL implements several optimisation techniques. The detailed debug functionalities show their commitment to usability. All these points are discussed in further detail in the following sections.

4.2.1 Input language

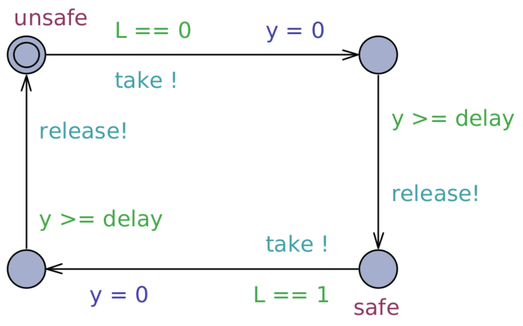
When creating a model manually, the graphical user interface of UPPAAL gives better at-a-glance information than the textual model representation. The models are shown as directed graphs, and you can edit the graphs directly.

To demonstrate this we take a look at the same example used in Section 4.1. The code for this example was already present in the UPPAAL installation folder, and it is used in this section. In Figure 4.6 the two process templates of the system are shown. It models vikings that can only cross a bridge to the safe side while holding the torch. The torch can only be held by two vikings at the same time, and there are four instances of the viking process, each with its own delay (how long it takes to cross the bridge).

The processes communicate with each other through the channels `take` and `release`.



(a) Torch



(b) Viking

Figure 4.6: UPPAAL model view of the bridge example

These channels and the variable `L` are declared in the global scope. A light blue `take!` synchronisation label triggers an action with the light blue `take?` label. These edges can only be taken together. The green edge labels are guards, so that edge can only be taken if the guard evaluates to true. The dark blue labels are assign statements that are executed when the edge is taken.

The location of the torch is tracked with the global variable `L`, so that in the viking model a guard can ascertain if a viking is in the same place as the torch, and only take the torch if that is the case. The value of `L` is changed once the torch is released by the last viking holding the torch. So if a viking wants to travel to the other side and back again, they have to take the torch, release it, and then take it again. This is also visible in the way the process of the viking loops.

To model the different speeds at which the vikings cross the bridge a clock is used. The variable `y` is a clock which is declared (not visible in Figure 4.6) in the scope of a viking. Once a viking takes the torch, the clock is set to zero. The viking can then only release the torch after the clock reaches a value greater or equal than the delay property of the viking.

4.2.2 Input files

UPPAAL supports three file formats [8] as input:

- XML
- XTA
- TA

Models created using the graphical user interface of UPPAAL are stored in the XML format. This format supports all the features of UPPAAL 4.0, including the graphical positioning of model elements. The XTA file format is a more readable format if you want to model in a textual way. This format does not include the graphical positioning of model elements, but they can be added with a UGI [10] file. The TA format is the older version of the XTA format. It does not support some newer features like process templates.

As an example we show the bridge example in XTA file format [31] in Listing 4.1.

Listing 4.1: Bridge example in ‘XTA’ file format.

```
chan take, release;
int [0,1] L;

process Viking(const int delay) {
  clock y;
  state S0, safe, S1, unsafe;
  init unsafe;
  trans
    S0 -> safe {
      guard y >= delay;
      sync release!;
    },
    safe -> S1 {
      guard L == 1;
      sync take!;
      assign y = 0;
    },
    S1 -> unsafe {
      guard y >= delay;
      sync release!;
    },
    unsafe -> S0 {
      guard L == 0;
      sync take!;
      assign y = 0;
    };
}

process Torch() {
  state one, S0, free, two;
  urgent S0;
  init free;
  trans
    free -> S0 {
      sync take?;
    },
    S0 -> one { },
    S0 -> two {
      sync take?;
    },
    one -> free {
      sync release?;
      assign L = 1 - L;
    },
    two -> one {
      sync release?;
    };
}

const int fastest = 5;
const int fast = 10;
const int slow = 20;
const int slowest = 25;

Viking1 = Viking(fastest);
Viking2 = Viking(fast);
Viking3 = Viking(slow);
Viking4 = Viking(slowest);

system Viking1, Viking2, Viking3, Viking4, Torch;
```

The XML file format has the same structure as the XTA format. All the model elements are declared in scopes denoted by curly brackets. This means that a variable declared in a certain scope can be accessed by elements that are descendants of this scope, while elements in parent scopes cannot access these variables. For example the channels `take` and `release`, the location variable `L`, the delay constants, and the model composition are declared in the global scope. Inside the process declarations it is possible to declare process independent variables like in this case the clock variable `y`. Next to that the states are listed, the init state is defined, and the transitions are specified. The transition labels are declared within the scope of the transitions.

When editing an XTA file in the graphical user interface of UPPAAL it automatically creates a '.ugi' file. An example file is shown in Listing 4.2. It uses the same names as in the XTA file and provides x and y coordinates for each element.

Listing 4.2: Example 'UGI' file for the bridge XTA file.

```

process Viking graphinfo {
  location S0 (195,76);
  locationName S0(-10,-30);
  location safe (-17,76);
  locationName safe(-10,-30);
  location S1 (-17,229);
  locationName S1(-10,-30);
  location unsafe (195,229);
  locationName unsafe(-10,-30);
  guard S0 safe 1 (-30,-42);
  sync S0 safe 1 (-30,-27);
  guard safe S1 1 (9,-31);
  sync safe S1 1 (9,-16);
  assign safe S1 1 (9,-1);
  guard S1 unsafe 1 (-30,0);
  sync S1 unsafe 1 (-30,15);
  guard unsafe S0 1 (9,-31);
  sync unsafe S0 1 (9,-16);
  assign unsafe S0 1 (9,-1);
}

process Torch graphinfo {
  location one (76,195);
  locationName one(-8,9);
  location S0 (76,51);
  locationName S0(-10,-30);
  location free (-51,127);
  locationName free(-10,-30);
  location two (212,127);
  locationName two(-10,-30);
  sync free S0 1 (-29,-30);
  sync S0 two 1 (-8,-30);
  sync one free 1 (-46,0);
  assign one free 1 (-42,19);
  sync two one 1 (-17,0);
}
```

Because requirement specification is also not supported in the XTA file format, UPPAAL saves the queries to a '.q' file. An example of this is shown in Listing 4.3.

Listing 4.3: Example query file for the bridge XTA file.

```
//This file was generated from (Academic) UPPAAL 4.1.24 (rev. 29
    A3ECA4E5FB0808), November 2019

/*
The system does not contain deadlocks.
*/
A[] not deadlock

/*
There exists a path which results in all vikings being on the safe side
*/
E<> Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe
```

4.2.3 Requirement language

To be able to formally express properties of models created in UPPAAL, it provides a custom property checking language. The language is similar to the commonly used CTL language, but it does not support nested path formulas, which limits the expressiveness. This does make the language easier to understand. The path formulas it does support are:

- Possibly: `E<> p`

  Evaluates to true if there exists a path where in some state `p` holds.

  CTL formula: `EF p`

- Invariantly: `A[] p`

  Evaluates to true if `p` holds for all reachable states.

  CTL formula: `AG p`

- Potentially always: `E[] p`

  Evaluates to true if there exists a path which results in an infinite loop of in which p always holds.

  CTL formula: `EG p`

- Eventually: `A<> p`

  Evaluates to true if all possible paths eventually reach a state in which `p` holds.

  CTL formula: `AF p`

- Leads to: `p -> q`

  Evaluates to true if for every state in which `p` holds all paths starting from that state eventually reach a state in which `q` holds.

  CTL formula: `AG p => AF q`

Next to that, UPPAAL also offers some syntax for querying statistical properties of the model. This includes constructs that give the probability of a certain state expression evaluating to true, compare probabilities to a specified bound, the comparison of two probabilities, and value estimation.

Both the path formulas and the statistical formulas make use of state expressions.With these it is possible to query anything in the state. They are basic formulas, also called predicates, that cannot have side effects. This could be the value of a clock, a global variable, or even properties defined in the scopes of locations. Common operators like `+`, `==`, and `&&` can be used here. In addition to that, UPPAAL provides some helper functions like `forall`, `exists`, `sum`, and `deadlock`. Especially deadlock is a very strong expression which can be used only in the requirement language. It specifies a state in which all processes do not have any transitions they can take, and therefore are stuck in the current location. The other helper functions can also be used in the programming language of UPPAAL, so they can appear in the guard and update labels.

As an example we can specify a requirement for the viking example. If we wanted to know if there exists a path where all the vikings end up on the safe side of the bridge we would write the following:

‘`E<> Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe`’

The `E<>` bit is the ‘Potentially’ operator, and it evaluates to true if the formula after it holds in a state in at least one of the possible paths. With `Viking1.safe` we specify that ‘Viking1’ is in the safe state.

4.2.4 Diagnostics

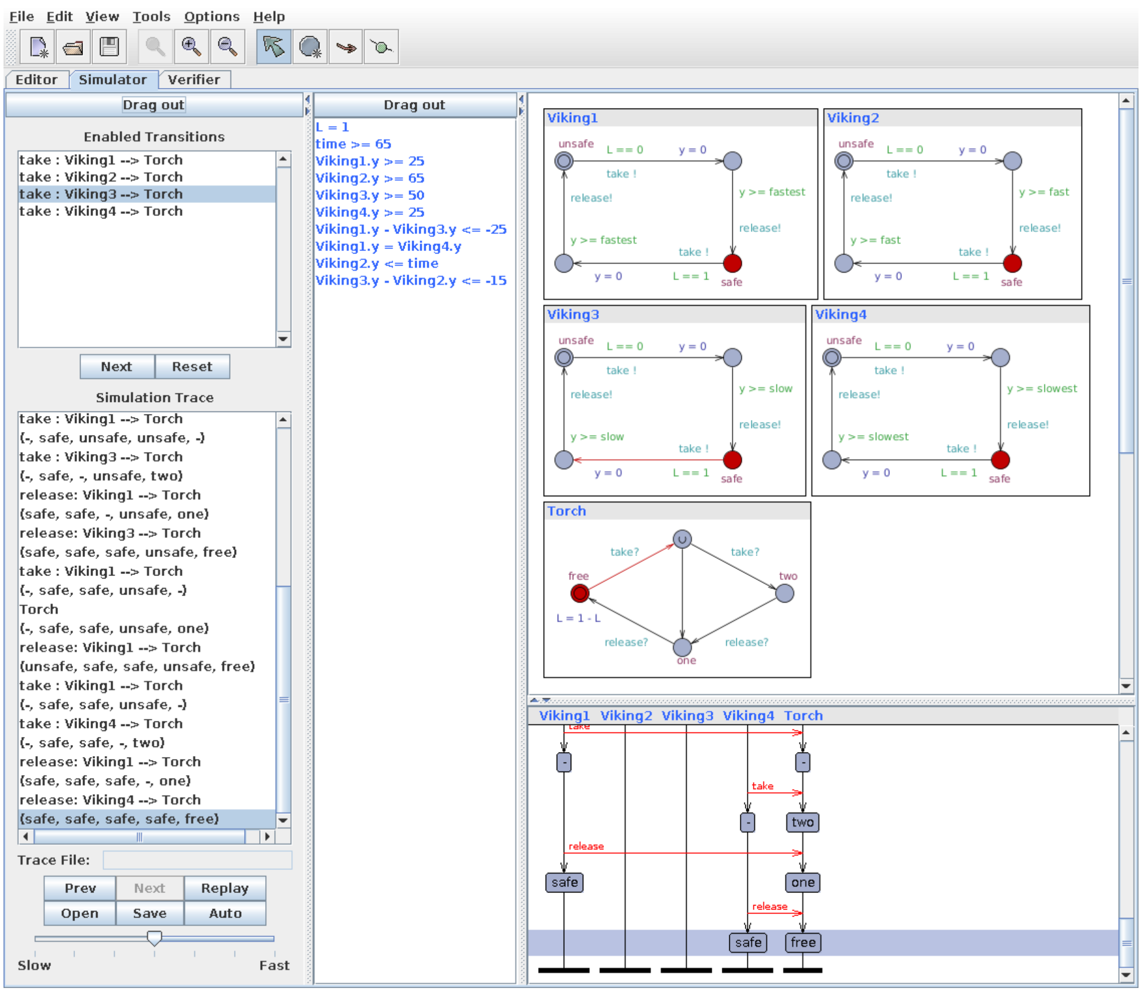
To demonstrate the diagnostics of UPPAAL, we tell it to check the requirement described above. As additional option we specify that UPPAAL should look for the shortest trace, satisfying this requirement. The resulting trace is shown partly in Figure 4.7 as it is too big to fit entirely in one screen.

In the top right window we see the model instances of the system and in which location each process is at the moment. To the left of that all the local and global variables are shown. And in the top left corner we see which transitions can be taken at this point, and they can even be selected and taken. Below that window in the bottom left corner, we see the simulation trace. A graphical representation of the trace is shown in the bottom right window.

4.2.5 Verification algorithm

In its core the verification algorithm of UPPAAL is a reachability algorithm, which looks for states in which the given constraints evaluate to true. At the same time, the algorithm tracks how such states can be reached. In their implementation they make use of symbolic states. This pairs states with the given requirements which hold in that state. UPPAAL gains much performance by preprocessing the model using the following techniques.

Most of the computation time is spent doing checks on the symbolic states and comparing them with each other. Because of this, the overall performance can be greatly improved by optimising these operations. Data structures which support such operations in an efficient way are Difference Bounded Matrices. They were implemented with some alterations [20], which resulted in a big performance boost.



Enabled Transitions

```
take : Viking1 --> Torch
take : Viking2 --> Torch
take : Viking3 --> Torch
take : Viking4 --> Torch
```

Simulation Trace

```
take : Viking1 --> Torch
(-, safe, unsafe, unsafe, -)
take : Viking3 --> Torch
(-, safe, -, unsafe, two)
release: Viking1 --> Torch
(safe, safe, -, unsafe, one)
release: Viking3 --> Torch
(safe, safe, safe, unsafe, free)
take : Viking1 --> Torch
(-, safe, safe, unsafe, -)
Torch
(-, safe, safe, unsafe, one)
release: Viking1 --> Torch
(unsafe, safe, safe, unsafe, free)
take : Viking1 --> Torch
(-, safe, safe, unsafe, -)
take : Viking4 --> Torch
(-, safe, safe, -, two)
release: Viking1 --> Torch
(safe, safe, safe, -, one)
release: Viking4 --> Torch
(safe, safe, safe, safe, free)
```

```
L = 1
time >= 65
Viking1.y >= 25
Viking2.y >= 65
Viking3.y >= 50
Viking4.y >= 25
Viking1.y - Viking3.y <= -25
Viking1.y = Viking4.y
Viking2.y <= time
Viking3.y - Viking2.y <= -15
```

Figure 4.7: Trace view of the vikings example.

Next to that, UPPAAL does some static analysis on the symbolic states to check if some are redundant and therefore can be removed. This reduction of the symbolic state space will also make the algorithm more efficient.

On a final note, UPPAAL caches the generated symbolic state space so it can be reused to check other properties, if more than one requirement has been given to evaluate.

4.3 mCRL2

To determine the benefit of an additional translation to UPPAAL next to the one of mCRL2 we also take a closer look at the tool mCRL2 [13] and see where the two tools complement each other. This section therefore contains a short description [12] of the tool. It is a newer version of the tool μCRL, which was designed to be minimal. With mCRL2 it is possible to analyse abstract models of concurrent processes that can communicate with each other. It utilizes process algebra to describe its input models. These models can be tested with properties specified in the language of the modal mu-calculus.

4.3.1 Input language

The input models of mCRL2 are specified in a textual way. The definition of the system is split up into separate sections of the file, where each section starts with a specific keyword. In its simplest form the model definition contains three sections:

- The definition of all the actions possible by the system. This section is prefixed by the keyword `act`.
- The definition of the processes of the system. This section is prefixed by the keyword `proc`.
- The definition of the initial state or the initialization of the system. This section is prefixed by the keyword `init`.

The actions section contains a simple list of all the action labels of the system. In the process section, the processes are described using process algebra. A simple simple example would be `X = a . b . X`. This process would accept an `a` action, then a `b` action, and then it would repeat itself again recursively.

An extra feature mCRL2 supports is that the action labels and process declarations can take parameters, for which constraints can be written to express more complicated systems.

To further get an understanding of the modelling language, we take a look at how the viking example, discussed in Section 4.1, can be modelled in mCRL2. This model is used as an example in the online tutorial for the tool [30]. The model is shown in listing Listing 4.4.

Listing 4.4: Vikings example in mCRL2.

```
% Specification for the rope bridge problem
% Written by Bas Ploeger, June 2008.

sort Position = struct start | finish;

```

```
6  act forward_viking,
7      forward_torch,
8      forward_referee,
9      forward: Int # Int;
10
11     back_viking,
12     back_torch,
13     back_referee,
14     back: Int;
15
16     report: Int;
17
18 % Models the torch which can move to the other side of the bridge
19 proc Torch(pos:Position) =
20        (pos == start) ->
21          sum s,s’:Int . forward_torch(s,s’) . Torch(finish)
22        <>
23          sum s:Int . back_torch(s) . Torch(start);
24
25
26 % Models an viking who can move to the other side of the bridge with
27 % its designated speed
28 proc Viking(speed:Int, pos:Position) =
29   (pos == start) ->
30     ( sum s:Int .
31         (s > speed) -> forward_viking(speed,s) . Viking(speed,finish)
32                     <> forward_viking(s,speed) . Viking(speed,finish)
33     )
34   <>
35     back_viking(speed) . Viking(speed,start);
36
37
38 % Models the referee who counts the number of minutes passed and the
39 % number of vikings that have reached the far side of the bridge
40 proc Referee(minutes:Int, num_finished:Int) =
41   sum s,s’:Int . forward_referee(s,s’)
42                . Referee(minutes + max(s,s’), num_finished + 2)
43   +
44   (num_finished < 4) ->
45     sum s:Int . back_referee(s)
46               . Referee(minutes + s, num_finished - 1)
47   <>
48     report(minutes) . Referee(minutes, num_finished);
49
50
51 init allow( { forward, back, report },
52   comm(  { forward_viking | forward_viking |
53            forward_torch | forward_referee -> forward,
54            back_viking | back_torch | back_referee -> back },
55     Viking(1,start) || Viking(2,start)  ||
56     Viking(5,start) || Viking(10,start) ||
57     Torch(start)    || Referee(0,0)
58 ));
```

On line 4 the datatype `Position` is defined. It can be seen as some kind of enumeration, which can have the value `start` or `finish`. These resemble the two positions the vikings and the torch can have, relative to the bridge. Next, the possible actions are defined. Each entity has a forward and backward action with the same parameter types. Both parameters contain the time an viking who is holding the torch needs in order to cross the bridge. The first one is always the one that takes the longest time. This is achieved with a constraint written in the `Viking` process. The backward motions however only have one argument, as it does not benefit the vikings to travel back with two people.

The definition of the `Torch` process on line 19 is very straightforward. It can do a forward motion or a backward motion depending on its current location.

The Viking process is a little more complicated. It takes two arguments, its speed (minutes it takes to cross the bridge), and its current position. Depending on its position it can do a forward motion. On line 31 it checks if its speed is greater than s, which is a free variable that represents the speed of the other viking crossing the bridge together with this viking.

Lastly the Referee process is defined to keep track of the time that has passed, and to see if all the vikings have crossed the bridge. It takes as arguments the number of minutes that have passed, and the total number of vikings on the right side of the bridge. It can record a forward and backward motion of the torch, and if all the vikings are on the right side of the bridge it will take the report action with as parameter the number of minutes that have passed. On line 46 it is visible how the referee keeps track of the time and the number of vikings on the right side by recursively calling itself and incrementing its parameters.

On line 51 it starts the definition of the system composition by stating that only the actions `forward`, `back`, and `report` are allowed. The `forward` and `back` actions are defined in the first part of the `comm` block. It specifies that for example the forward action consists of two `forward_viking` actions, a `forward_torch` action, and a `forward_referee` action occurring simultaneously. In the second part of the `comm` it specifies that the system consists of four viking processes, a torch process, and a referee process running in parallel. Within the definition the initial parameters of the individual processes are also given.

4.3.2 Requirement language

As requirement language, mCRL2 supports the modal mu-calculus [4] with their own syntax [25]. It is split up into State formulas, Regular formulas, and Action formulas.

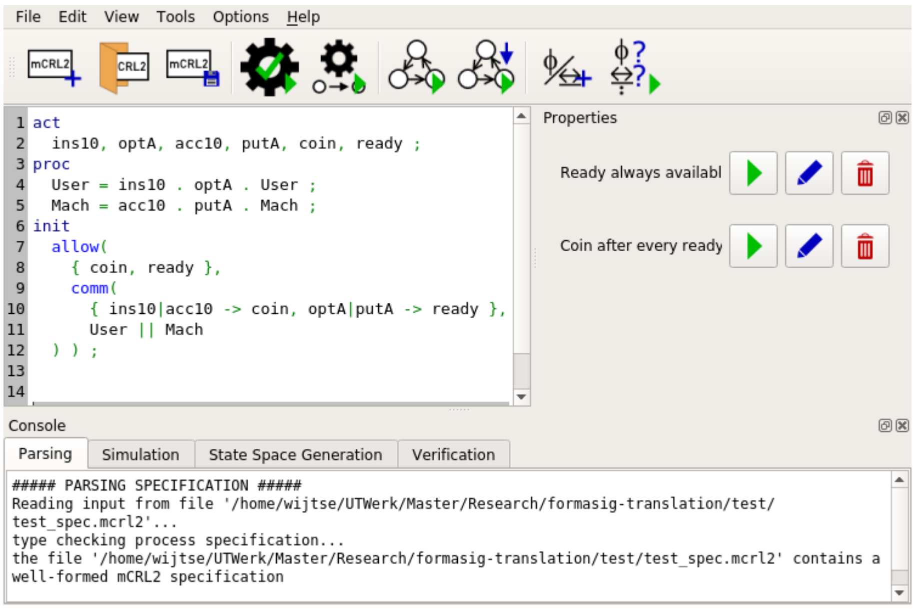
With the action formulas you can specify action labels used in a process of the model, or data expressions. Data expressions can access state variables and may contain common arithmetic operators. These formulas are extended with useful functions like `forall` and `exists` to help with specifying a state condition.

The regular formulas wrap around the action formulas to combine Action formulas or quantify them. It contains 4 operators:

- `formula + formula` specifies a choice between the two formulas.
- `formula . formula` specifies a concatenation of the two formulas.
- `formula *` repeats the formula zero or more times.
- `formula +` repeats the formula one or more times.

The regular formulas are used inside the State formulas. The State formulas we are mainly interested in are the ones related to the modal mu-calculus:

- `mu StateVar . StateForm` maps to the mu-calculus formula: $\mu X.\phi$



```
act
  ins10, optA, acc10, putA, coin, ready ;
proc
  User = ins10 . optA . User ;
  Mach = acc10 . putA . Mach ;
init
  allow(
    { coin, ready },
    comm(
      { ins10|acc10 -> coin, optA|putA -> ready },
      User || Mach
  ) ) ;
```

```
##### PARSING SPECIFICATION #####
Reading input from file '/home/wijtse/UTWerk/Master/Research/formasig-translation/test/
test_spec.mcrl2'...
type checking process specification...
the file '/home/wijtse/UTWerk/Master/Research/formasig-translation/test/test_spec.mcrl2' contains a
well-formed mCRL2 specification
```

Figure 4.8: Model view of the vending machine example in mCRL2

- `nu StateVar . StateForm` maps to the mu-calculus formula: $\nu X.\phi$
- `[ RegForm ] StateForm` maps to the mu-calculus formula: $[\![a]\!]\phi$
- `< RegForm > StateForm` maps to the mu-calculus formula: $\langle a\rangle\phi$

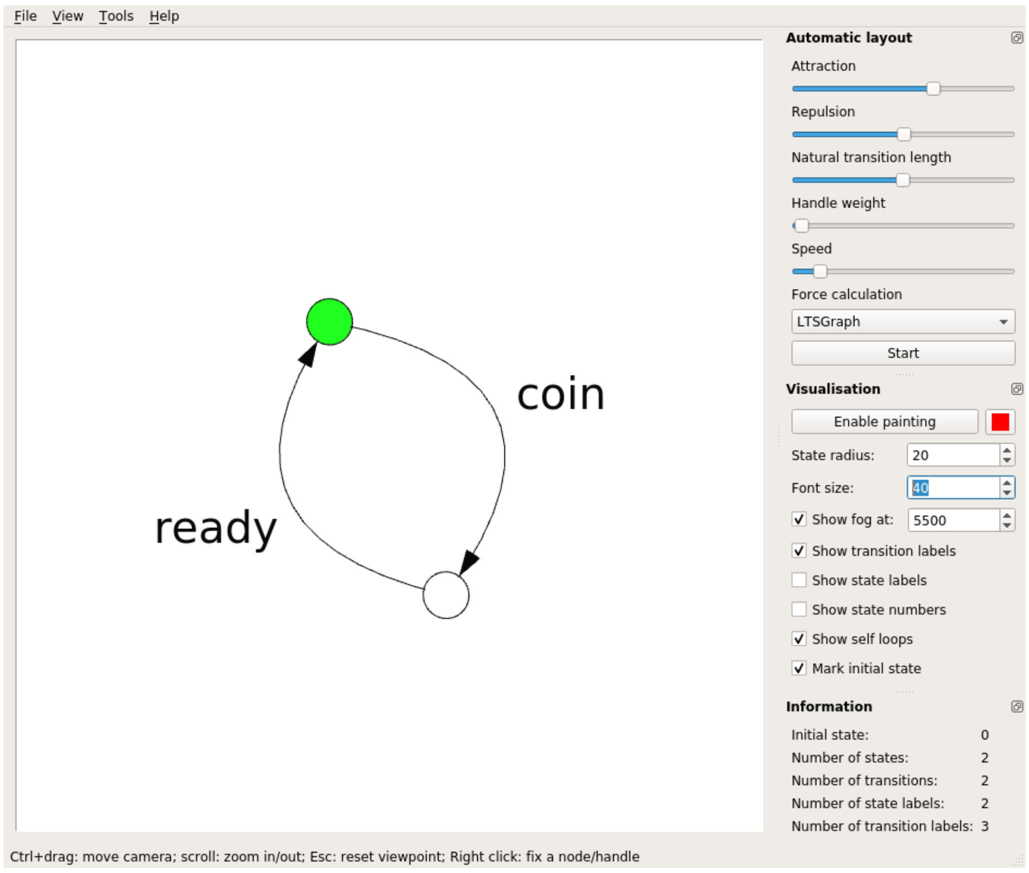
StateVar contains a declaration of a state variable, StateForm is recursively a State formula again, and RegForm is a Regular formula containing Action formulas.

4.3.3 Diagnostics

As a simple example we take the vending machine [1] from the mCRL2 website. The model description and the mCRL2 modeling interface are shown in Figure 4.8.

Here we can see the separate model sections `act`, `proc`, and `init` as described before. The User process can only do the action ins10 (which means insert 10 currency inside the vending machine), and after that an action optA (which means select the apple in the vending machine). Next, the user can repeat those actions in the same order. The vending machine's actions are 'accept 10 currency', and then `put/give an apple`. These actions can also be repeated infinitely in that order. In the innit section, the `ins10|acc10 -> coin, optA|putA -> ready` part means that the actions `ins10` and `acc10` synchronize and emit a `coin` action. The same for the `optA` and `putA` actions respectively. The `User || Mach` part specifies that the two processes `User` and `Mach` run in parallel.

In Figure 4.9 an example diagnostic trace is shown for the property `[ true* . ready . !coin ] false`, which means that after every `ready` action a `coin` action takes place.



Force calculation: LTSGraph

State radius: 20
Font size: 40
✓ Show fog at: 5500
✓ Show transition labels
☐ Show state labels
☐ Show state numbers
✓ Show self loops
✓ Mark initial state

Initial state:	0
Number of states:	2
Number of transitions:	2
Number of state labels:	2
Number of transition labels:	3

Ctrl+drag: move camera; scroll: zoom in/out; Esc: reset viewpoint; Right click: fix a node/handle

Figure 4.9: Diagnostic trace view of a requirement of the vending machine example.

Aspect	UPPAAL	mCRL2
GUI	• Graphical representation is possible. • Easy at a glance information.	• Only graphical snapshots can be generated of the model. • The model cannot be edited in a graphical way.
Textual	• The structure of the model uses scopes. • Contains a C like language for update actions, guards, and invariants on locations and edges. • Language describes the structure of state machines.	• Data is scoped, but functions/processes are not. • Language uses process algebra to specify processes.

Table 4.1: Moddeling in UPPAAL compared to mCRL2

4.4 UPPAAL compared to mCRL2

To further support the choice of using UPPAAL to verify EULYNX models in addition to mCRL2 we need to take a look at the advantages and disadvantages of UPPAAL compared to mCRL2. In the paper ‘Verifying liveness in supervised systems using UPPAAL and mCRL2’, Markovski and Reniers [24] compare the two tools by applying both to a case study. Their findings are used as well as my findings in the comparison below. The aspects of the two tools can each be separated into three, the modelling language, the requirement language, and the diagnostics.

4.4.1 Modelling language

In UPPAAL you can create models in two ways, with a textual representation of the model, or through a graphical user interface. Here the models are created in an object oriented way. mCRL2 only supports textual representations of models. These models are described in a more formal way, focusing on the specifying the behaviour of a system and its processes.

4.4.2 Requirement language

UPPAAL provides its own requirement language. The formulas that can be created with this language is a subset of CTL. mCRL2 contains a syntax with which it is possible to create modal mu calculus formulas that can be used for checking the model.

Aspect	UPPAAL	mCRL2
Path expressions	• Is a subset of CTL. • Nested path expressions are not possible. • Limited expressiveness. • Easier to understand.	• Supports the modal mu-calculus. • Superior expressiveness.
State expressions	• State based expressions. • Full access to all the variables of the state.	• Action based expressions. • Not possible to access state variables directly.

Table 4.2: Requirement specification in UPPAAL compared to mCRL2

Aspect	UPPAAL	mCRL2
Traces	• Can give a trace as a counter example or a witness. • Can walk through the trace. • Shows the whole state at every step.	• Gives traces only as counter examples, no witnesses. • Can walk through the trace. • State of individual components are difficult to untangle. • Provides multiple tools to make traces easier to read.

Table 4.3: Diagnostics in UPPAAL compared to mCRL2

4.4.3 Diagnostics

Both tools have the ability to check the specified requirements in a graphical user interface, and give counterexamples when a property is not satisfied. The information UPPAAL gives, however, is a little more accessible.

5. Translation

In this chapter we discuss the structure and design of the EULYNX to UPPAAL translator. First we go into the EULYNX interpretations that are suitable for this research and core design decisions of the translator, followed by the general structure of the generated model. After that we zoom in on the different steps the translator takes and how the different components in EULYNX are translated. This is followed by a general overview of how the fundamental elements of EULYNX models are translated.

5.1 Possible EULYNX interpretations

Because the semantics of UML, SysML, and EULYNX are not exactly defined and some behaviour is ambiguous, there are multiple interpretations of the modelling language. Next to the goal of creating a translation that produces as similar behaviour as possible to the mCRL2 translation, we also look at which interpretations fit best with UPPAAL as target language. There are two main points where we have to decide which path we should take, because these lay the foundation of the semantics of parallel state machines. Other minor interpretation choices follow from these decisions. The first is how multiple state machines should interleave, and the second is how alterations in the memory are shared between the different state machines.

5.1.1 State machine interleavings

How the different parts of the model run together lays the foundation for how the rest of the model needs to be implemented. UPPAAL gives the different processes the possibility to do a step (take a transition) at any time, given that there are no time constraints or channel synchronisations. All these steps of the different processes are done sequentially. It is possible that one process takes many steps, while the other stays in its current state, again given there are no time constraints or channel synchronisations. This is also a possible interpretation of EULYNX.

The other interpretation we consider for this subject is where the model runs on cycles [29], and that in each cycle all the state machines do one step. This prevents that in some traces one state machine is being blocked whiled the other is constantly activated. It also has the added benefit of a reduced state space, because there are less state machine interleavings possible. While the individual steps still occur sequentially, each state machine has done one step before the cycle repeats. This makes it also possible to implement in UPPAAL. To orchestrate the processes to take one step each cycle, a separate process is required that manages the cycles of the model and triggers the different processes to take a step.

5.1.2 Memory sharing

State machines in EULYNX are connected to each other via ports and flows. These can also be seen as properties where a change in value is reflected on the connected ports. The other state machine can then read the new value of this property. How these variable alterations are handled can be done in several different ways. The current translation to mCRL2 uses queues for this. Each alteration is put in a queue and handled separately by the receiving state machines. UPPAAL does not offer queues as a specific functionality, however they can be implemented using arrays and a bit of additional code for the different operations.

A different interpretation is the variant with direct memory sharing. In this variant the state machines operate on shared variables, so the change is visible to the other state machine immediately. Implementing this in UPPAAL is fairly straightforward as UPPAAL provides a global scope in which shared variables can be declared. Another variation of this interpretation is where there are two variables declared for every port connection, the input and the output side. With this it is possible to publish all changes at the same time, by creating an atomic operation that writes the values of all the source sides of the flows to the variables corresponding to the target sides of the flows.

5.1.3 Decisions

When looking at the first option for possible state machine interleavings, the main drawback we see is the bigger state space compared to the more strictly organised cycle variation. However, a benefit of this interpretation is that the model requires less orchestration, as the different components can act on their own. We also see that the first option regarding memory sharing in the model could increase the state space of the model, as every change in the memory needs to pass through a queue which adds more states in between. It also requires extra code in UPPAAL to manage the queues as they are not available as a standard feature. Direct memory sharing with a publication phase requires little additional code, and it does not increase the state space by adding extra steps. This interpretation on memory sharing also works well with the cycle approach to model orchestration. The publication of the port values to the other state machines can become a step in the execution cycle of the model. Therefore we choose to combine the cycle interpretation for the state machine interleavings, with the direct memory sharing interpretation. These fit the UPPAAL modelling language the best, along with our goals for the translation.

5.2 General design

We define the order of the model execution steps with a clock cycle. This approach is similar to the one described in [29]. Here all the state machines and their sub-regions are synchronised by following the steps below.

- State machines finish one step
- Publish port values
- Update event queue
- Pop first event from queue

First, the state machines have to do one step, and finish their run-to-completion until they are in the next state. With 'run-to-completion' we mean the steps a state machine has to take to transition from one state to the next state. These steps need to be completed before the next cycle can begin, hence the term 'run-to-completion'. This means that all the exit, update, and entry behaviour that should be triggered by taking that transition has been executed. Here it is important that all state machines take a transition or internal behaviour step if their is one enabled. It is not allowed that one state machine does not do anything while it can do something.

Then when all the state machines are stable, the port values are published. This is done by taking the values of the output side of the port and assigning those values to the input sides of the ports.

Next, the event queues of all the state machines are updated based on the newly updated variables.

After that, the event managers will start taking from their queue. When a queue is empty the cycle starts again with the first step. Otherwise, the first item of the queue is popped and is triggered so the state machine and its children can react to it. If there is no state machine which can handle this event, this event is discarded and another one is taken from the queue until either the queue is empty or one of the state machines could take care of an event. When all the event managers are done with updating and popping from the queue, the cycle goes back to the first step. Note, that some state machines already started with executing the behaviour triggered by an event. These will finish their run-to-completion and will not take an additional step this cycle.

To realise the synchronisation of state machines in this way we chose to create a separate process for each state machine to manage the event queue of that state machine. This removes additional complexity from the main state machine's process, which would have made it more difficult to read. Next to that, in order to force processes to do a step, the orchestrator process is introduced. This process will keep track of where in the cycle the model is, and signal the state machines to take a step if they are able to.

5.2.1 Existing solutions

Before we designed our own translator, a few papers on similar translators have been taken into consideration. Though inspiration was taken from these papers, we could not reuse those translations, as their interpretations of the state machine semantics tend to differ and the EULYNX dialect adds more features then are available in SysML state machines.

In the paper by Muniz et al [26]. they use additional processes to manage events in the model. These processes are called middelware automata. They created an automata for putting events in the queue, and one for taking events from the queue and dispatching them. Their event system uses priorities, so the event queue is priority based. This does not match with our interpretation of EULYNX events, but we do use the idea of the event queue being managed by separate processes. Additionally we combine the queue update process and event dispatching process into one process. The complete explanation on how we implemented them is in Section 5.3.2.

A single event queue manager process is also used by Knapp et al. in their paper on translating UML state machines to UPPAAL [18]. Their event queue is also not priority based, but in their implementation they do not use model execution cycles. The event manager process can dispatch events and add events to the queue at any time. Another translation strategy they use is flattening the whole model to a single UPPAAL process. The active state configuration of a state machine with multiple sub-regions is encoded in a single UPPAAL state. From there are all the transitions calculated which are possible for that specific state configuration. While this results in an output model which is barely readable, it does prevent the possible extra overhead of synchronising between multiple UPPAAL processes. Because of our goal that we want to keep our output model as simple and readable as possible we choose to not flatten the model in this way, but instead create a separate UPPAAL process for each state machine and sub-region. This is further explained in Section 5.3.1. Another interesting note they make is how they clean the unused part of the event queue array. This prevents similar model states being classified as different, because UPPAAL takes the whole array into account when comparing different states. If you do not reset the unused part of the array, there could be data in the rest of the array that is different in some traces while it is not being used anymore.

5.3 Structure

In this section we will discuss the structure of the generated model. In order to automatically translate a EULYNX system to an equivalent model in UPPAAL, we chose to create a framework around the state machines. This moves certain model management duties to dedicated processes, which keeps the state machine processes more readable. These extra components were briefly discussed in the previous section, and are explained in further detail here.

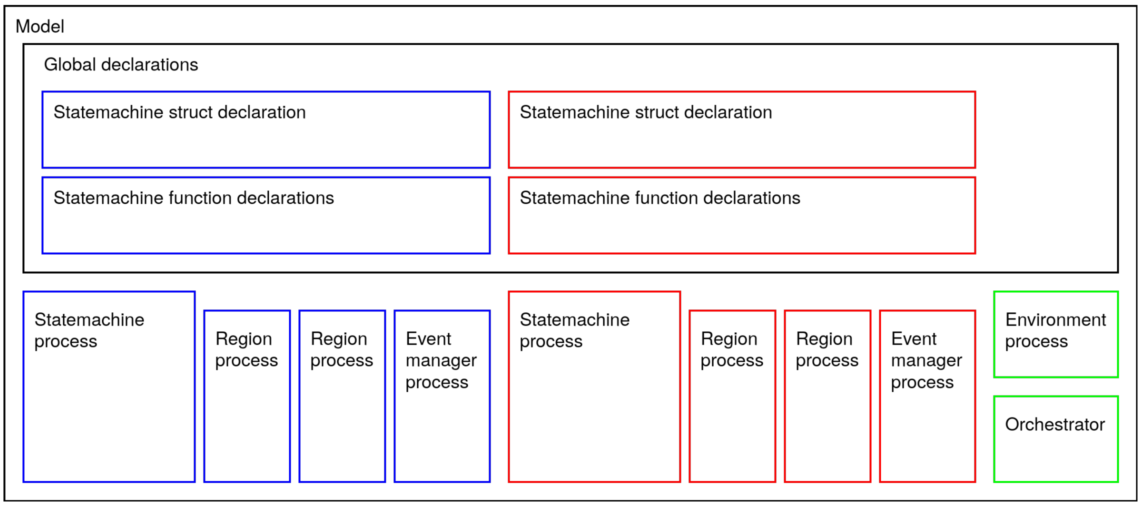


Figure 5.1: Model architecture

5.3.1 State machines as processes

The general model architecture on a global level is shown in Figure 5.1. Each state machine is represented as its own process in UPPAAL. A EULYNX state machine can have many properties in the form of ports and variables. These properties cannot be stored as local variables of a process, because those cannot be accessed by other processes. This is necessary for example for the data transmission between ports of different state machines. Therefore those are stored in the global scope of the model. This does come with the downside of the model being less readable, but we mitigate this effect by organising the variables in a separate struct per state machine. This also fixes the possible naming duplicates between two state machines. All the operations of each state machine are translated to UPPAAL functions which are declared in the global scope. The reason for this is that we represent composite states and sub-regions of a state machine as separate processes in UPPAAL, and these process also need access to those functions, so they could not be declared in a process.

Composite states and regions

Composite states in EULYNX add the possibility for parallel behaviour. To support this in the generated model we chose to create a new process for each region. It is also possible that a region itself contains a composite state with sub-regions. To distinguish the processes from each other, the name of the process is prefixed with the name of the parent process. Each state machine process, its region processes, and event manager process will only access the variables in the state machines struct.

All processes created from a region or sub-region start in a waiting location. When the parent process of one of those processes enters the corresponding composite state, they activate their sub-processes by way of a channel synchronisation. A simple example of a state machine with a single sub-region is shown in Figure 5.2. The state machine is called "STM", and it has a composite state called "C1". This composite state has itself an exit transition to S6 which if taken should stop the sub-region from running. Next to that the region has two transitions which terminate the region and cause the parent state machine

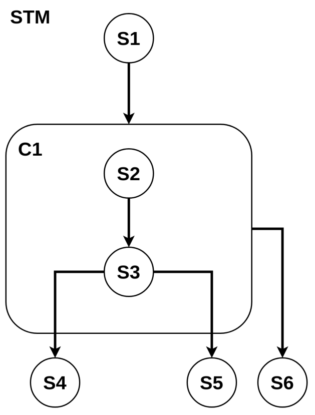


Figure 5.2: Simple composite state example

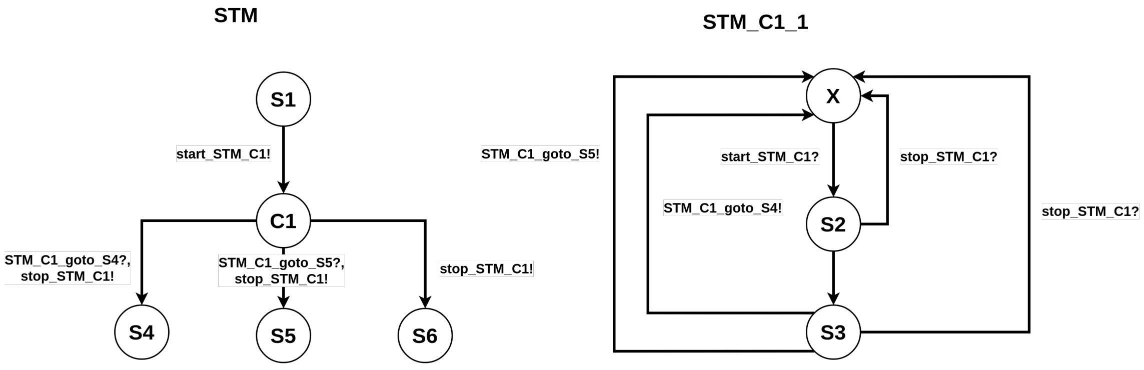


Figure 5.3: Simple composite state example in UPPAAL

to go to either state "S4" or state "S5".

The simplified translation of the example to UPPAAL is shown in Fig. 5.3. This consists of two processes, one called "STM" and one called "STM_C1_1". The latter represents the single sub-region of STM. For this example the states of STM are translated one to one. Some edges have two channel synchronisations written on them. These can be seen as two sequential transitions with a committed state between them, as it is not possible in UPPAAL to have two channel synchronisations on a single transition.

The child process starts in the waiting state marked "X". Here it stays until it receives on the `start_STM_C1` broadcast channel. The states `S2` and `S3` are translated normally, but both get an extra transition to the waiting location `X` with the a receive synchronisation on the broadcast channel `stop_STM_C1`. This allows the parent process to stop the child process at any time. For instance if the parent process takes the transition to `S6`. Lastly, the two transitions between `S3->S4`, and `S3->S5` cross the border of the region. Therefore they are translated in the child process as transitions back to the waiting state with a sending action on a non-broadcast channel. This signals the parent process to go to the state the child process would have gone to outside its region.

5.3.2 Event managers

As EULYNX state machines make use of event queues to handle changes in their data, we need to keep track of these queues in some way. This is done with a separate event manager process for each state machine. Because the sub-regions work with the same event queue, we only create one per state machine as is shown in Figure 5.1. The thing we need to keep in mind when dealing with sub-regions is that the regions at a lower depth in terms of nesting depth have priority when handling events. So if an activated state in a sub-region is able to handle the event, and the activated state in their parent is also able to handle the event, the event will be handled by the state in the sub-region.

An example of how a generated event manager process could look like in UPPAAL is shown in Figure 5.4. Here we consider a state machine `stm`. It has a max nesting depth of 1, so it has sub-regions, but those do not have sub-sub-regions. It has two events:

0) `y == 2`
1) `x == 2`

An event manager has several helper variables. One is the `queue_index` variable. This holds a pointer to where the next queue element can be stored. So if the queue is empty, it will read `0`, and if the queue contains 2 elements it will read `2`. To keep track of when the expression of an event becomes true, the event managers have an array called `event_state` in which the evaluation of the expression in the previous cycle is stored.

The event manager starts in the `NEUTRAL` state. It waits for the orchestrator process to send on the `event_check` channel. Then it checks for every event expression if the current value is different than the one in the previous clock cycle. If an expression has become true, it will add it to the queue. We chose to create transitions for each event individually instead of putting all checks in a function on one transition. Otherwise the order in which events are being put in the queue would have been deterministic. Once the `event_state` array has been fully updated, it can either go back to the initial state if the queue is empty, or to the `TRIGGERING` state if the queue is not empty.

How events are handled is as follows. If the active state of a state machine process is waiting for an event it is listening on one of the `<stm_name>_handle_event_<depth>` channels. The depth part of the channel name signifies how deeply nested the state machine is at that point. The most deeply nested states in the state machine have priority over those of their parents when it comes to handling events, so therefore the event manager first signals on the broadcast channel with the highest depth number. Transitions that receive on a `handle_event` channel have a guard that checks if the event ID at the start of the queue is the one they are waiting for. If this is not the case, that process will do nothing. If the current event is the one the process is waiting for, it will take that transition and set their `event_handled` variable on true. From that variable the event manager can determine if the event has been handled by a state machine. If it is handled it will pop it from the queue and return to the `NEUTRAL` state, otherwise it will try again on a lesser depth in the state machine. If the event has been triggered on all the levels of the state machine and could not be handled by a single one, that event is popped from the queue, and the same process starts with the next event in the queue. Unless the queue is empty. In that case the event manager goes back to the `NEUTRAL` state.

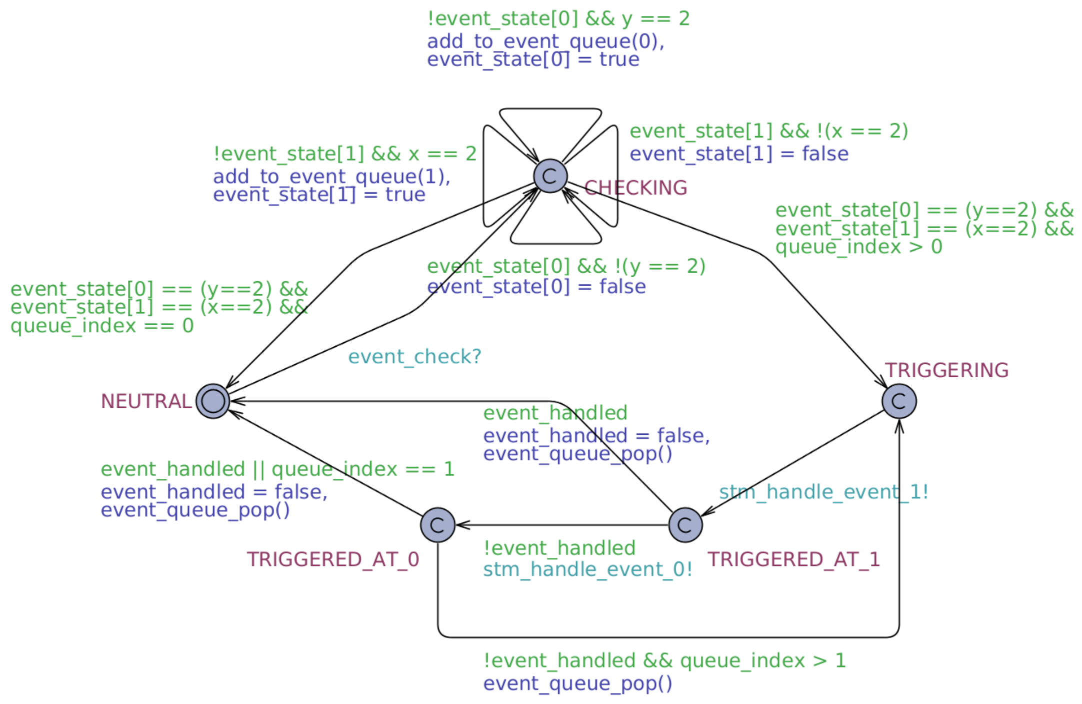


Figure 5.4: Event manager example

In a previous implementation of event managers, we implemented the priority of the different state machine depth levels with UPPAAL's channel priority system. Each state machine depth event channel had a priority in the order of most deep first. With this UPPAAL handled the order of execution for us. However, during testing we found out that UPPAAL limits the property checking language when the model contains channel priorities. In this case the operators `A<>`, `E[]`, and `->` cannot be used. The `deadlock` predicate is also excluded. Therefore we opted for the aforementioned approach without channel priorities.

The overall coordination of the different processes is displayed in Figure 5.5. Here we consider a model with two blocks, but we focus on only block A. A state with a red colouration is considered a committed state. All the processes start in a neutral state. It is possible that a state machine process can take a transition and start handling all the behaviour related to that. The orchestrator process waits until all the other processes are in a stable state. This includes state machine processes, event managers, and the environment process. When this is the case the orchestrator process can send the event check signal. When this transition fires, all the port values are published so all the state machines have each others updated values. The event check signal will trigger the event managers to start updating the event queue, and later on pop elements from the queue if there are any. This can make a state machine begin a step, which could be for example taking a transition and executing all the entry and exit behaviour connected to it. Next, the orchestrator process will wait for all the event handlers to be in a neutral state. Then, it will send a signal on the broadcast channel `do_step`. This will trigger all the state machines to do a step if they are able to and have not already done so in this cycle. And then the cycle repeats.

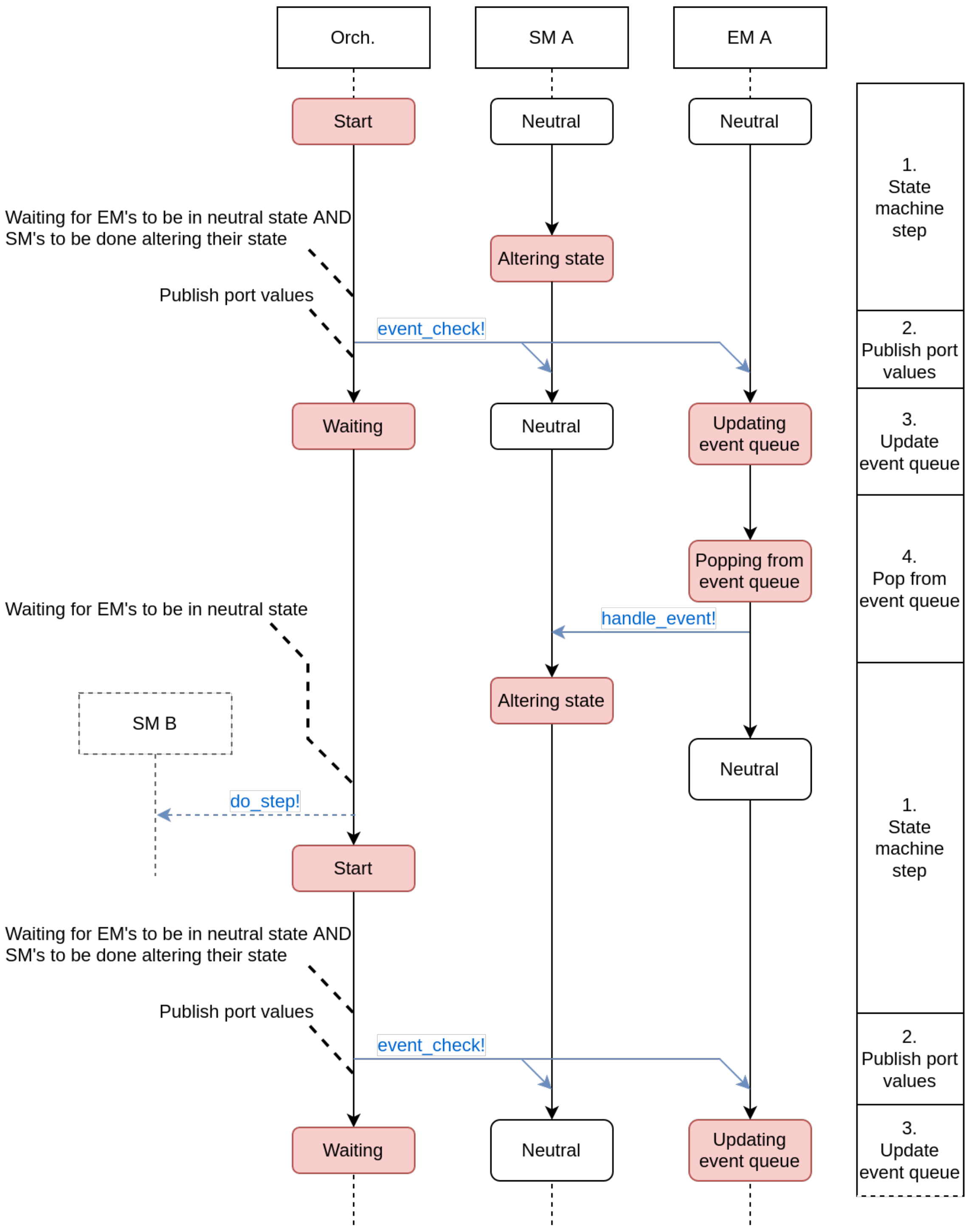


Figure 5.5: State machine orchestration

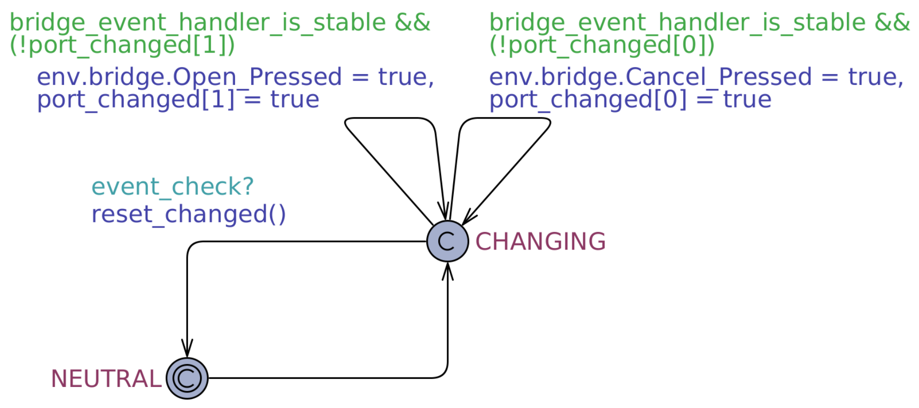

bridge_event_handler_is_stable &&
(!port_changed[1])
env.bridge.Open_Pressed = true,
port_changed[1] = true
bridge_event_handler_is_stable &&
(!port_changed[0])
env.bridge.Cancel_Pressed = true,
port_changed[0] = true
event_check?
reset_changed()
C
CHANGING
NEUTRAL

Figure 5.6: Environment process

5.3.3 Environment

To simulate the environment, we created an extra process. An example of one is shown in Figure 5.6. The functionality of the environment process is simple. It only changes the input end of environment ports in a non-deterministic way. Creating a location with a self-loop for every possible change to the environment leads to unnecessarily many possible executions. To decrease the complexity and unnecessary behaviour we limited the environment process by letting it change each port at most once every cycle. This is what the `port_changed` boolean array is for. This array is reset to all false at the end every cycle on the transition back to the neutral state.

Next, we also saw that very big environment processes could fill the window with possible transitions of the manual simulation screen in UPPAAL with a lot of options. For big models there could easily be a hundred extra possible transitions. To keep this window more clear, we added an extra state with an empty transition, so instead of always seeing one hundred extra options, you at first see only one extra option. When you do want to change the environment, you take that extra transition, and all the extra options become available.

5.3.4 Orchestrator

To keep the behaviour of the state machines in a cycle and force them to do a step each cycle we created an orchestrator process. An example is shown in Figure 5.7. This model contains one state machine called bridge, and that state machine contains a composite state called `OPENING` with a single region. As described in the sequence diagram of Section 5.3.2, the orchestrator process checks when the model execution cycle can go to the next step. First it waits for all the processes to be in a stable state. When that is the case it can send the `event_check` signal and publish all the port values so the event managers can update their queues and send their events to the state machines. When all the event managers are done, the orchestrator process sends a `do_step` signal to make the state machines that did not take a transition this cycle and can, do so.

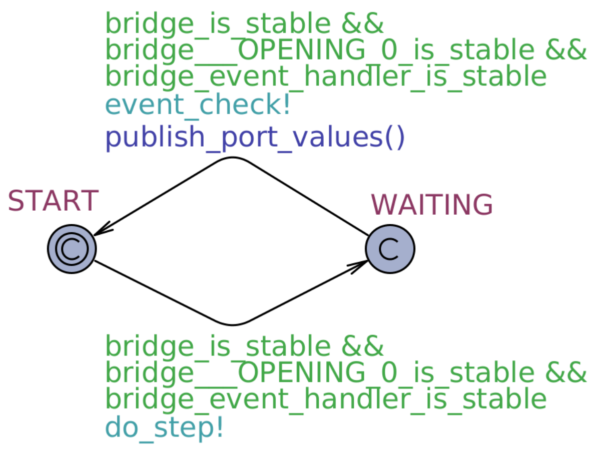

bridge_is_stable &&
bridge___OPENING_0_is_stable &&
bridge_event_handler_is_stable
event_check!
publish_port_values()
START
WAITING
bridge_is_stable &&
bridge___OPENING_0_is_stable &&
bridge_event_handler_is_stable
do_step!

Figure 5.7: Orchestrator process

5.3.5 Data management

As stated before, all the properties of a state machine need to be located in the global scope so other processes can also access those variables. These variables are organised in a struct per state machine. An example is shown in Listing 5.1. The name of the struct uses the state machine name (IBD name) and is prefixed with `str_`.

Listing 5.1: Example state machine struct

```
struct {
  bool Is_Open;
  bool Button_Pressed;
  int Counter;
} str_bridge = { false, false, 0 };
```

The basic programming language used in EULYNX specifications is ASAL. This is in terms of functionality very similar to UPPAAL, so the translation is very straight forward. However, there are a few additional steps required. In terms of data types EULYNX has the same types except for string and pulse types.

Strings can be used in EULYNX, but are not supported in UPPAAL. However, strings in EULYNX can be considered as constants. There are no operations done on strings, so give each different string an ID and treat them as integers in UPPAAL. Here we assume that the source model does not contain any type mixing, so regular integers and string ID integers are kept separate.

The type 'pulse' is a special boolean type that can be used in EULYNX which can only be read once after it has been set to true. After it is read it is set back to false. In our translation we reset the pulse variables to false after one cycle. Because of this, the event managers had to change a little. If a pulse port is set to true, reset back to false at the end of the cycle, and set to true in the next cycle, it should be able to trigger an event in both cycles. However the event managers check the difference between cycles. So from the event managers' perspective it stayed true for two cycles. We solved this by resetting the `event_state` array for all events that contain pulse variables to false after each cycle.

Publishing the port values at the end of the cycle is done with a single generated function.

For every flow in the model it copies the value of the output port variable of the flow to the input port variable. If the port is of type pulse, it also sets the output port variable to `false`, so the input port is automatically set to `false` at the end of the next cycle if the pulse port is not set to `true` again. An example `publish_port_values()` function is shown in Listing 5.2.

Listing 5.2: Example publish port values function

```
void publish_port_values() {
  str_light.Is_Open = str_bridge.Is_Open;
  str_bridge.Button_Pressed = env.bridge.Button_Pressed;
  env.bridge.Button_Pressed = false;
}
```

In the declaration of functions in UPPAAL, the order matters and recursion is not allowed. Specifically, a function needs to be declared before it is used in UPPAAL. The operations in jEULYNX are not ordered in such a way that takes care of this. Therefore we implemented a simple algorithm that orders the functions based on dependency (Topological sort, depth first search [6]).

5.4 Translation steps

As a starting point we use the existing DSL called jEULYNX [3] to specify models of the EULYNX. We also reuse some preprocessing steps that remove things like overlapping blocks. We translate the resulting model to an intermediate model of UPPAAL that we created in Java. This is a common model driven engineering [17] practise where first the meta models (a model that describes the structure of a model) of the source and target models are created, and the translation is built by mapping elements of the source meta-model to the target meta-model. This allowed us to work on the printing of the model file separately from the actual translation, and prevented us from overcomplicating the translation by doing multiple things at the same time. The translation part can be split up in a few major steps. These are shown in the sequence diagram of Figure 5.8.

In the first step we create the global declarations. This is done by looping over all the blocks in the model, creating an UPPAAL struct for each, and putting all the variables of the block in that struct. Additionally, the operations of that block are also translated. For this we created an AST walker, which recursively traverses the ASAL AST and converts it to the UPPAAL equivalent for which we also created a model in Java.

Next we loop again over all the blocks of the model and translate each one to an UPPAAL process (in some cases multiple processes). This is done with a state machine visitor that functions like a recursive state exploration algorithm. It traverses the state machine, starting at the initial state, and builds the UPPAAL process bottom up. The entry locations of the states that have already been visited are saved and reused. When a composite state is encountered, the visitor creates a new visitor for each region of the composite state and lets them explore. All the created processes and other things like created synchronisation channels are returned to the caller. After a state machine is translated, its corresponding event manager is also created.

Lastly, the orchestrator process and the environment process are created.

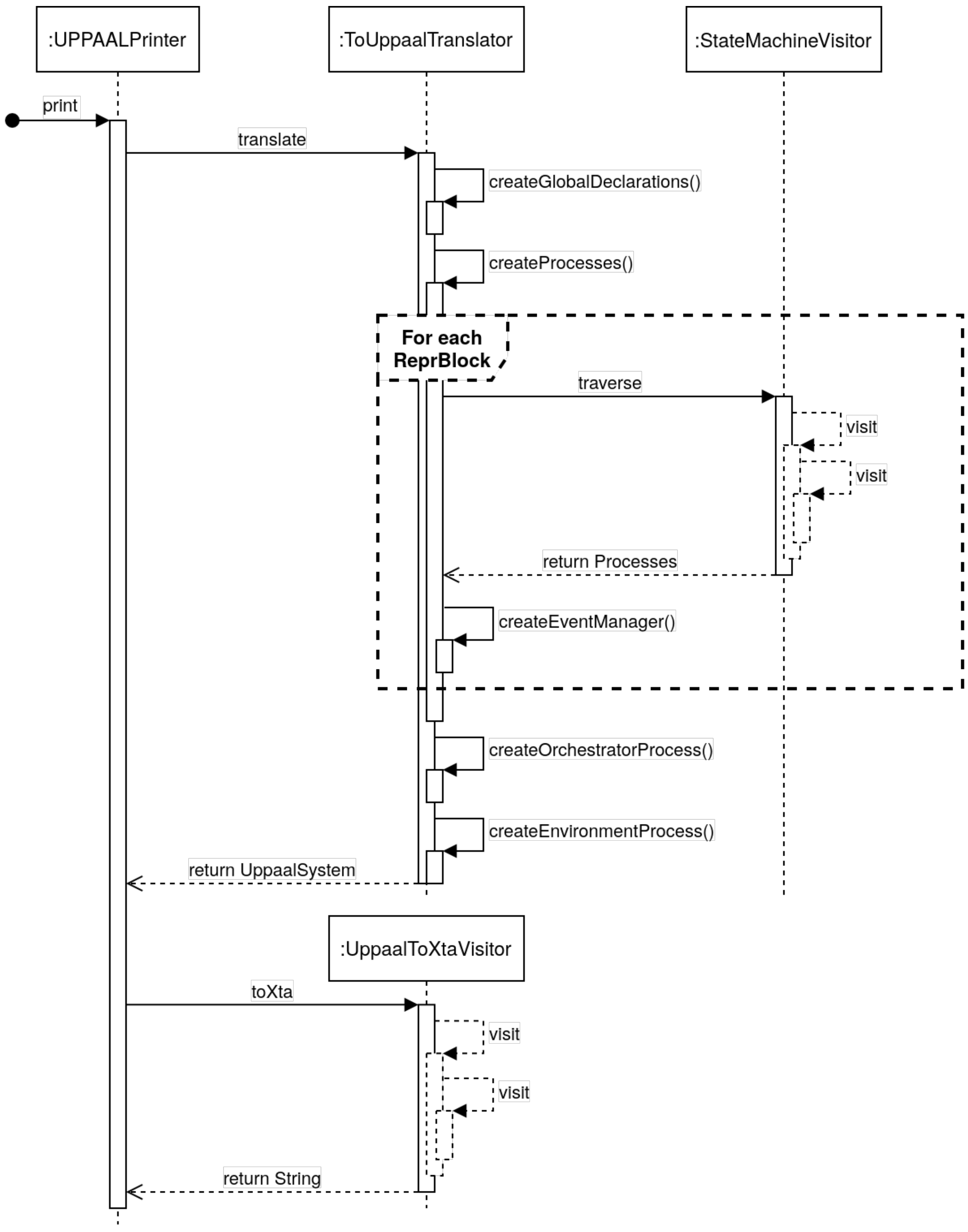


Figure 5.8: Translator sequence diagram

5.5 Translation of basic elements

In this section we describe how the basic components of EULYNX are translated to UPPAAL. Specifically the simple state, composite state, transitions with and without events, and internal behaviour with and without event are discussed. These components are explained in further detail with the help of translation templates. These are figures in which a specific part of the target UPPAAL model is shown. In these figures the same colour coding is used as in the UPPAAL tool. Green is used for the guards of a transition, blue for update behaviour, and teal for channel synchronisations.

5.5.1 Simple state

The translation template of a simple state in UPPAAL is shown in Figure 5.11. This is a translation pattern which is reused for each simple state. The simple state in EULYNX is a state which can have entry, exit, and internal behaviour. Any transition to a simple state first lands in the *Entry point* location. This is a committed location that has a single transition to a waiting location (labelled with a "-"). The entry behaviour of the simple state is put on the transition from the *Entry point* to the waiting state. In the same transition the process's `is_stable` variable is set to true, because the run-to-completion of this state machine is done. Once the model can move on to the next step in the execution cycle, the `event_check` channel is signalled and the process moves from the waiting location to the *Main* location of the state. The waiting location and the `Main` are not committed, so in a later stage time constraints can possibly be implemented. All the outgoing transitions of the EULYNX state have as source location the *Main* location. Also the internal behaviour starts from here.

5.5.2 Transition

A transition of a simple EULYNX state to another state in the same region is translated to a single UPPAAL transition. The translation template for this component is also shown in Figure 5.11. This transition has as source the *Main* location of the EULYNX state and as target the *Entry* location of the successor state. A condition on the EULYNX transition is directly translated to UPPAAL code and put as a guard on the UPPAAL transition. Exit behaviour of the previous state and the transition effect can also directly be translated to UPPAAL code and added to the update behaviour of the transition. Note that the transition effect code is added after the exit behaviour code, in order to conform to SysML specification. Additionally the `is_stable` variable of the process is set to `false`, since the process has started a run-to-completion. Because the transitions shown in Figure 5.11 are not triggered by an event, they are triggered by orchestrator process via the `do_step` channel.

The translation template for a transition with an event is shown in Figure 5.9. The update statements of the transition and the condition are translated in the same way, but a check if the event of the transition is in the first position of the event queue is added to the guard. The event id is a number generated during the translation to identify each different event in a state machine. The other thing that is different from a regular transition is that synchronisation label. On this transition it waits for a `handle_event` synchronisation, which is signalled by an event manager process. Together with the guard it makes sure that this transition is only taken when the correct event occurs.

5.5.3 Internal behaviour

As internal behaviour can be seen as a self loop where the state machine does not exit the current active state. Therefore no exit or entry behaviour is executed, and the transition only carries a condition, transition effect, and possibly an event. The translation template for internal behaviour without an event is shown in Figure 5.11 as the transition from the *Main* state to the waiting state marked with a dash. Note that it does not update the `is_stable` variable as the process is in a stable state in both the source and target location. The template for internal behaviour with an event is shown in Figure 5.10. Like the the transition in the previous section, the only changes are the addition to the guard, and synchronisation on `handle_event` instead of `do_step`.

5.5.4 Composite state

The translation template for a composite state is shown in Figure 5.12. The structure of this state in UPPAAL is very similar to that of a simple state. The first difference is the added `start_regions` synchronisation on the edge from the *Entry* location to the waiting location. This sends a broadcast signal to all the UPPAAL processes that represent the regions of this composite state. Next, it waits just like in the simple state for the orchestrator process to send the `event_check` signal so it can transition to the *Main* location. Internal behaviour is done in the same way as for a simple state.

Where the template starts to differ a lot from the simple state translation template is in the outgoing transitions. Here we need to make sure that the regions are stopped correctly and that the exit behaviour is executed in the correct order. The transition shown at the top right of Figure 5.12 is an example of a transition with an event from the composite state to a different state. The first part takes care of listening for the correct event, and the other regular transition elements including the condition and update behaviour. This UPPAAL transition goes to a committed location where it waits until the processes of the regions are in a stable state, otherwise they cannot be stopped. Once this is true, the process transitions to another waiting location while it signals the regions to stop via the `stop_region` channel. Then it waits until all the regions have successfully terminated and it takes the transition to the *Entry* location of the successor state. This last transition contains the exit behaviour of the composite state. In this way the exit behaviour of the states in the regions is executed before the exit behaviour of the parent state.

The transition shown in the bottom right of the same figure is an example of a transition that is triggered by a EULYNX transition that has as source state a transition inside a composite state region and as target a state outside that region. This should terminate the region process and move the active state of the parent process from the composite state to the target state. Instead of waiting for a `do_step` or `handle_event` synchronisation, the process waits for a `goto_state` synchronisation. In the actual output model this channel has a more detailed name, including which process should go to which state. The UPPAAL transition with this synchronisation is also copied to the waiting location before the *Main* location, because the process should be able to synchronise on this channel while `is_stable` is set to true. In earlier iterations of the translation this extra transition was not added, which resulted in a deadlock when the composite state took an internal behaviour transition and the region transitioned to an external state. This was detected with the validation described in Chapter 6.

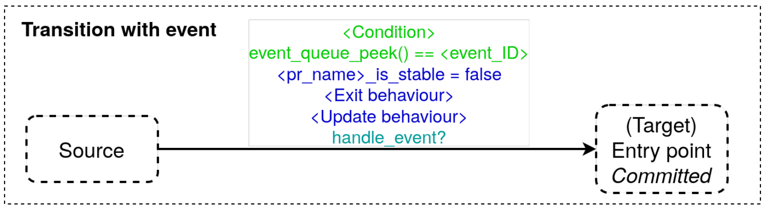


Figure 5.9: Transition with event translation to UPPAAL

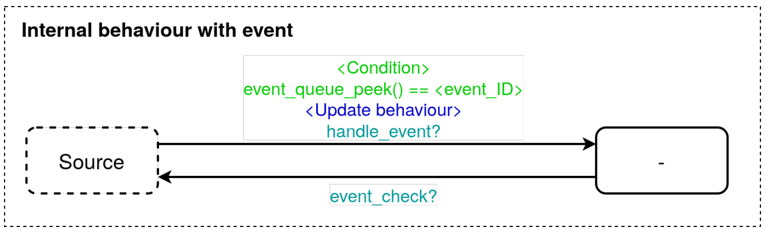

Internal behaviour with event
<Condition>
event_queue_peek() == <event_ID>
<Update behaviour>
handle_event?
Source
-
event_check?

Figure 5.10: Internal behaviour with event translation to UPPAAL

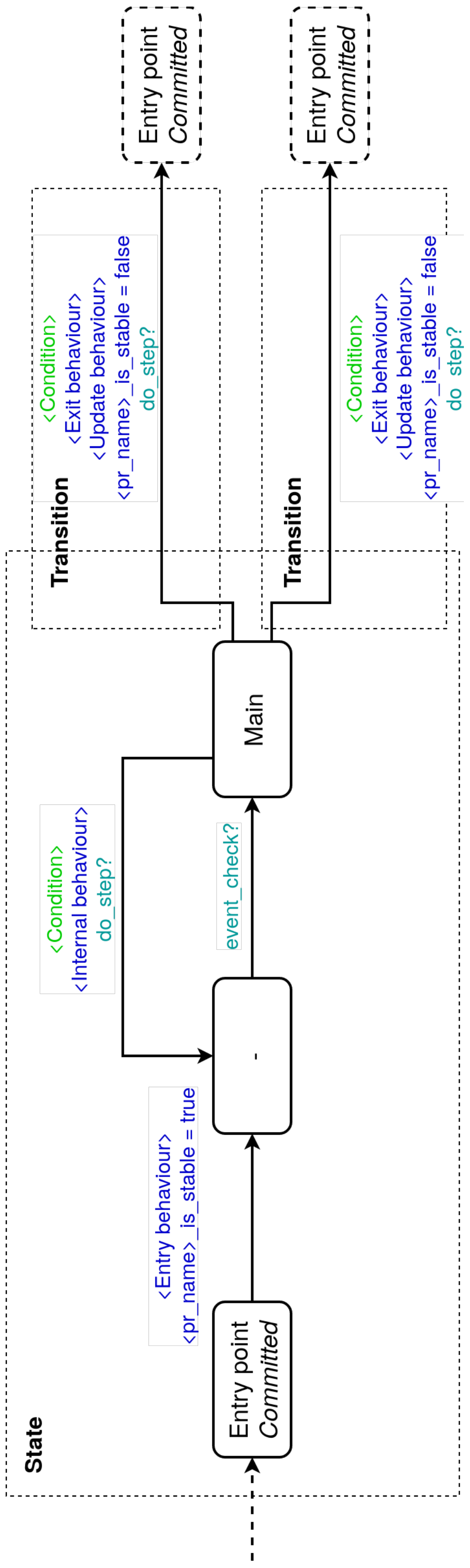


Figure 5.11: State translation to UPPAAL

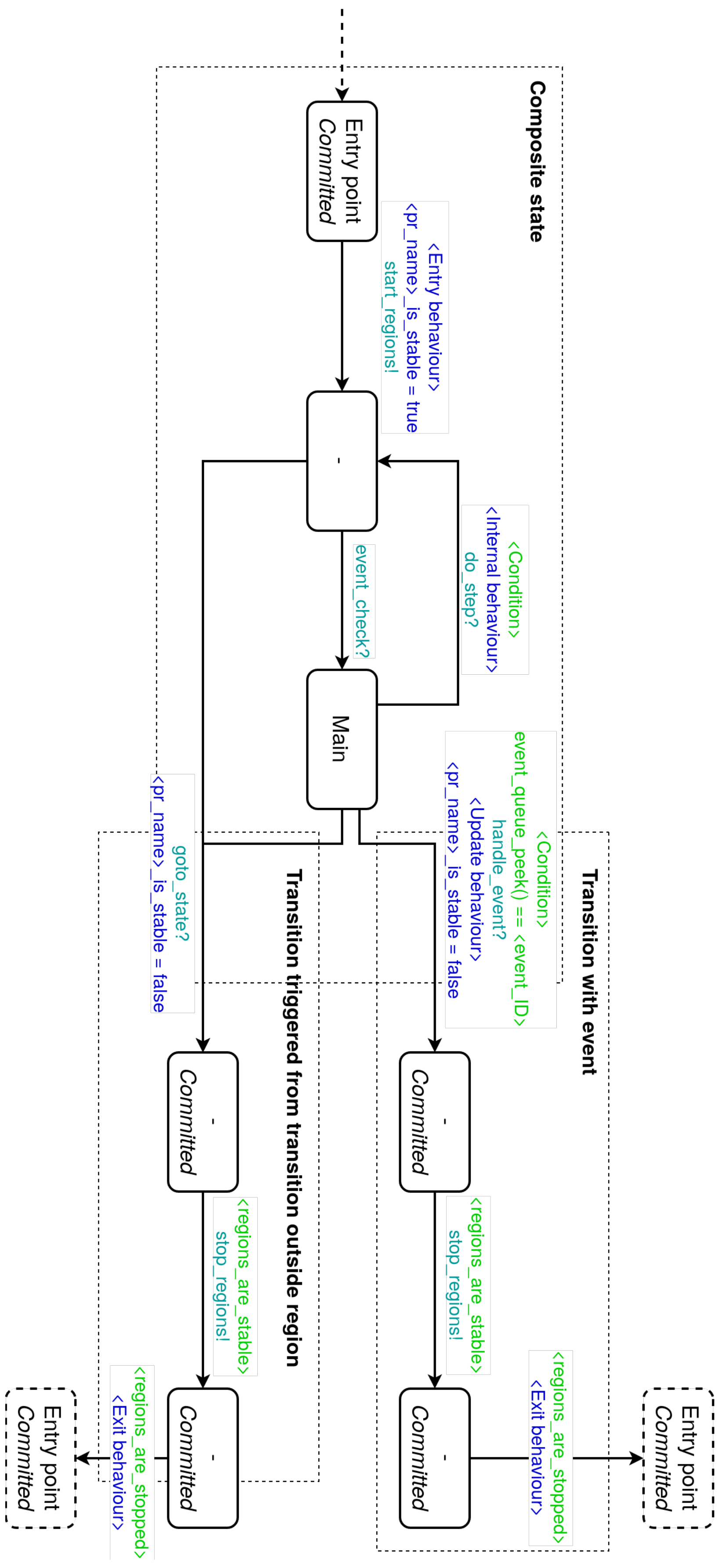


Figure 5.12: Composite state translation to UPPAAL

5.6 Element naming

In this section we describe the naming of different elements in the output model of the translation. We look at elements that will be referenced in the UPPAAL properties.

One of the main elements that will be referenced are the properties and ports of the different EULYNX state machines. As described in Section 5.3.5, all the properties and ports have a corresponding variable declared in the struct of that state machine. These variables can be referenced by writing `str_<stmName>.<portName/propName>`. Directly referencing the output side of ports to the environment is also possible by writing `env.<stmName>.<portName>`.

The state machines and their regions are translated to processes as described in Section 5.3.1. The main process of the state machine is called the exact same as the name it is given in the jEULYNX framework. Sub-processes created from regions of composite states are named `<parentProcName>___<stateName>_<regionNumber>`. The region number is an index starting at zero and incremented by one for each region. Because composite state regions themselves can also contain composite states, the name of the parent process is used instead of the state machine name. For example if we have a state machine called 'stm' which has a composite state called 'A' and the first region of that composite state has a composite state called 'B', the process name of the first region of state 'B' would be called `stm___A_0___B_0`.

The main UPPAAL location for each EULYNX state is named `<name>_<type>`. The type of the state is added for easier debugging. A couple of example location names are `CONNECTING_Composite` and `ERROR_Simple`. In an UPPAAL requirement you can refer to a location of a process with `<procName>.<locationName>`, so that could look like `stm.A_Composite`.

Each state machine can also have functions which in EULYNX are called operations. These are directly translated to UPPAAL code, and can be referenced by writing `<stmName>___<operationName>()` with optional function arguments.

This naming scheme was used to provide predictable variable names and give a better overview of the model in debugging. However it is possible that names can clash in very specific scenarios, that reserved words of UPPAAL are used, or that the names clash with the boilerplate code created added in the translation. To prevent this, a future version of the translator could implement a more elaborate naming scheme, where for example every element of the model gets a generated unique ID, or that every name is checked for reserved words or duplicates and gets a post-fix if it is the case. However, in most cases the current naming scheme does not give problems, especially with the naming conventions of EULYNX.

6. Validation

In this chapter we validate the correctness of the translation. This is done by creating small models that each use a specific functionality of EULYNX. We keep the example models small, but also cover all the use cases of each functionality. These models are then put under test by a number of properties that test for the correct behaviour or exclude incorrect behaviour. The properties are tested using the verify tool in UPPAAL.

6.1 Test suite coverage

We first create a list of features and test scenarios that we want to cover. These are listed in Table 6.1, along with where they are tested. The features are written as requirements, and can get very specific to cover as much edge cases as possible of every feature.

There are some desired features that have not yet been implemented completely. However these are included in the requirement list. In the current version of the translator, initial vertices act as simple states, while they should be treated ad pseudo states. That is why the requirements 13, 14, and 20 are not applicable to the current version, but they should be validated when this feature is added in a future version.

6.2 Test cases

To test if the translation produces an output model with the correct behaviour we create small example models that each focus on a separate feature of EULYNX. For the different models we describe the functionality which is under test, how the model uses this functionality, how we test this with UPPAAL properties, and the results.

Nr.	Requirement	Tested	Section
1.	Transition conditions work correctly in basic scenarios	✓	6.2.1
2.	Transition effects are executed correctly in basic scenarios	✓	6.2.2
3.	Events: A "when" event is correctly dispatched in simple situations	✓	6.2.3
4.	Events: A "when" event is only triggered and handled once	✓	6.2.3
5.	Events: A "when" event interrupts composite states (with-/without multiple regions)	✓	6.2.7
6.	Events: A "when" event does not interrupt a composite state when the same event is handled by a nested state	✓	6.2.4
7.	Events: "after" events are correctly dispatched in simple situations	✓	6.2.5
8.	Events: "after" events do not have to trigger	✓	6.2.5
9.	Events: "after" events can interrupt composite states (with-/without multiple regions)	✓	6.2.5
10.	Entry behaviour is correctly executed in basic scenarios	✓	6.2.6
11.	Exit behaviour is correctly executed in basic scenarios	✓	6.2.6
12.	Junction vertices work in basic scenarios	✓	6.2.8
13.	Junction vertices work when used as successor of an initial vertex in a composite state with one region	n/a	
14.	Junction vertices work when used as successor of initial vertices in a composite state with multiple regions	n/a	
15.	Nested states exit in the correct order when they are interrupted by an event	✓	6.2.7
16.	Nested states exit in the correct order when a transition crosses the border of a region	✓	6.2.7
17.	Do behaviour is executed correctly in basic scenarios	✓	6.2.8
18.	Orthogonal regions act independently	✓	6.2.7
19.	Orthogonal regions act simultaneously when triggered by the same event	✓	6.2.4
20.	Effects after initial vertices are executed correctly	n/a	

Table 6.1: Traceability matrix of feature validation

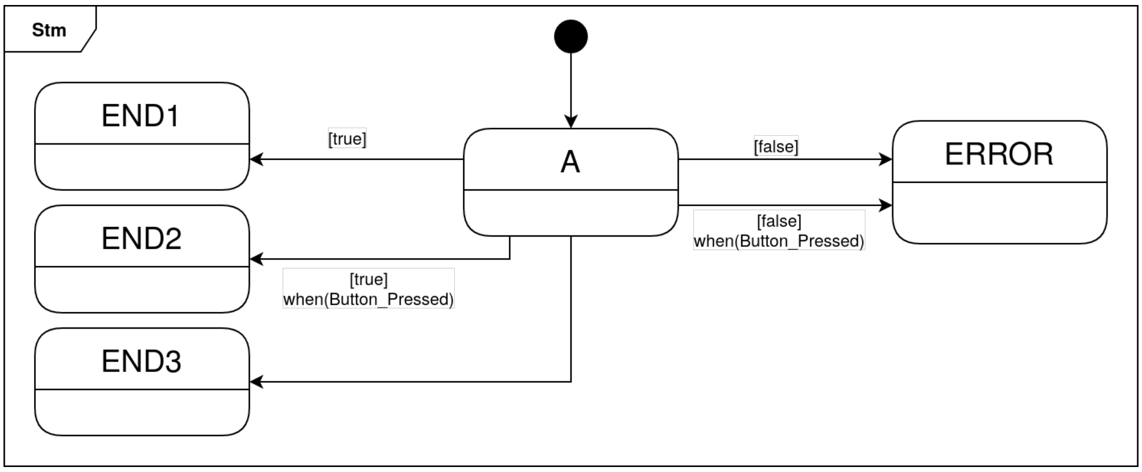


Figure 6.1: Test model for guards

How the different model elements are referenced in UPPAAL properties for the output model of the translation can be found in Section 5.6. This helps with getting a better understanding of the requirement we use to verify the behaviour in the various test cases.

6.2.1 Transition Guards

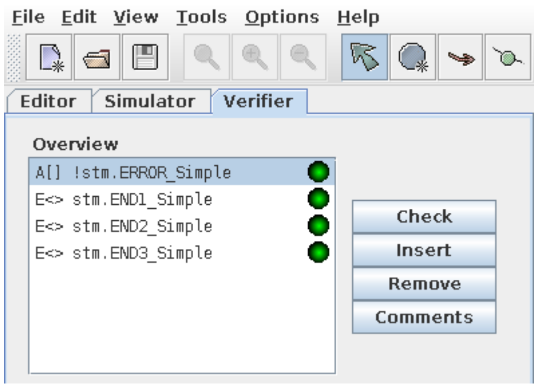
A guard on a transition is a simple boolean expression. It can contain a whole formula consisting of variable references, function calls, and operators. For this purpose we only use the boolean literals `TRUE` and `FALSE`. When the guard on a transition evaluates to `TRUE`, the transition is enabled, and can be taken. If the guard evaluates to `FALSE`, the transition cannot be taken. A transition without a guard is also enabled.

To test this we created a model where some transitions have either a guard with the boolean literal `TRUE` or `FALSE` on them. The model is shown in Figure 6.1. Both transitions with and without events are used. All the transitions that have a guard with the boolean literal `FALSE` have as target state the `ERROR` state. All the other states should be reachable. For completion we also include a transition without a guard. This transition should also be enabled. To generate events we use a pulse port to the environment we called `Button_Pressed`.

The output UPPAAL model of this example can be found in Appendix B. This gives an idea of what the generated models look like. The graphic positioning of the locations, transitions, and labels is done by hand, but the model content is generated using the translation.

The behaviour of the output model is verified with the properties listed below. To test that the guards properly enable and disable transitions, we check for the reachability of all the states.

- `A[] !stm.ERROR_Simple`
- `E<> stm.END1_Simple`
- `E<> stm.END2_Simple`
- `E<> stm.END3_Simple`



```
A[] !stm.ERROR_Simple ●
E<> stm.END1_Simple ●
E<> stm.END2_Simple ●
E<> stm.END3_Simple ●
```

Figure 6.2: Results of guard test model properties

The first property describes that for all possible traces, the location `ERROR_Simple` is not activated in any state. Which should evaluate to true, because all the transitions to that state are always not enabled. The rest of the properties describe that there exists a trace where there is a state in which the specified location is enabled. These should all evaluate to true, because all of those states are reachable.

Testing these properties resulting in all of them passing. As an example, the property checking window containing these queries with their results is shown in Figure 6.2.

6.2.2 Transition Effects

A transition can also have an effect. This is a bit of code that is executed when the transition is taken. This occurs after the exit behaviour of the source vertex and before the entry behaviour of the target vertex. The bit of code can be any possible piece of ASAL code.

For simplicity we only use assign statements in the example model. The model is shown in Figure 6.3. The model starts in state `A` where it has 3 different transitions it can take. They all mutate the property `X` in a different way such that the behaviour is distinguishable in the verifier. From state `END3` the model can take another transition that increments the variable `X` untill it reaches 20. This cap at 20 is added, so the model checker does not encounter an integer overflow error.

The properties we used to verify the behaviour of the model are shown below. First we establish that the property `X` starts at `0`. Next we check that the different states are reachable. And finally we check if the transition effects were properly executed.

- `A[] stm.A_Simple imply str_stm.X == 0`
- `E<> stm.END1_Simple`
- `E<> stm.END2_Simple`
- `E<> stm.END3_Simple`
- `A[] stm.END1_Simple imply str_stm.X == 2`
- `A[] stm.END2_Simple imply str_stm.X == 4`
- `A[] stm.END3_Simple imply str_stm.X >= 6`
- `A[] stm.END3_Simple imply str_stm.X <= 20`

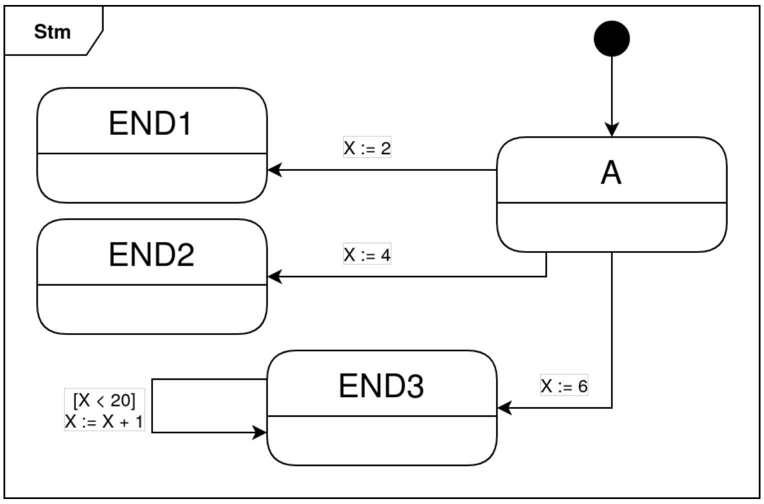


Figure 6.3: Test model for transition effects

- `(stm.END3_Simple && str_stm.X == 6) --> (stm.END3_Simple && str_stm.X == 7)`

6.2.3 Trigger events

Events allow for the possibility to wait for a certain condition, and immediately act upon it if the condition is satisfied. When an event occurs it is put into the queue, and once it is at the front it can be consumed, but only once. After that the condition must evaluate to false, and then true to trigger the event again.

The small model we created to test events is shown in Figure 6.4. We use a port to the environment called `Button_Pressed` and some internal behaviour in state `A` to trigger events. All states except for the `ERROR` state should be reachable. The transition from `A` to `ERROR` waits for `FALSE` to become `TRUE`, which will never happen. The transition between `END2` and `ERROR` waits for the same event as the transition between `A` and `END2`. This should never occur because `X` is only altered in state `A` and once the active state is `END2`, the event has already been consumed.

The properties we created to verify the behaviour of the model in UPPAAL are listed below.

- `A[] !stm.ERROR_Simple`
- `E<> stm.END1_Simple`
- `E<> stm.END2_Simple`
- `(stm.A_Simple && str_stm.Button_Pressed && str_stm.X == 0) --> stm.END1_Simple`
- `(stm.A_Simple && !str_stm.Button_Pressed && str_stm.X == 3) --> stm.END2_Simple`
- `(stm.A_Simple && !str_stm.Button_Pressed && str_stm.X == 3 && stm_event_handler.CHECKING) --> stm.END2_Simple`
- `E<> (stm.A_Simple && !str_stm.Button_Pressed && str_stm.X == 3 && stm_event_handler.CHECKING)`
- `A[] stm.END2_Simple imply str_stm.X == 3`

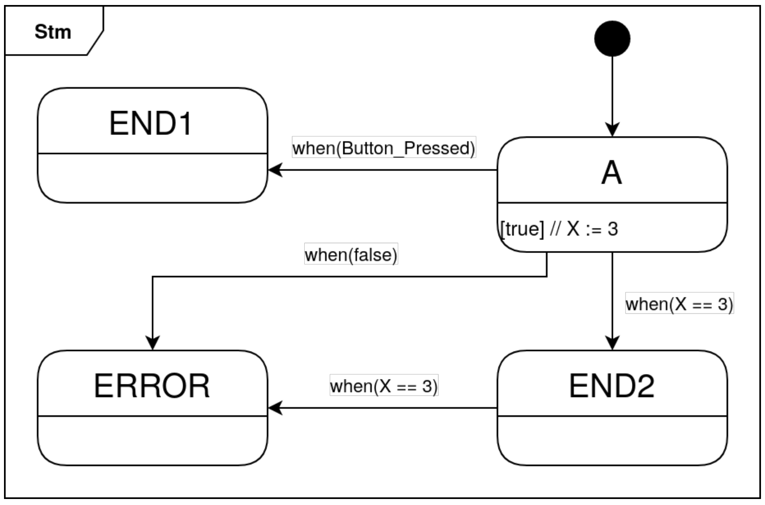


Figure 6.4: Test model for events

The first three properties specify that the correct states can be reached, and the ERROR state cannot be reached. Then we specify for each event scenario that eventually the correct successor state is reached. The first property that specifies behaviour that should result in the transition to the END2 state ended up not being satisfied acording to UPPAAL. After some debugging we found that the environment was able to set the port Button_Pressed to TRUE after the internal behaviour of A was triggered. This can result in the event of Button_Pressed being handled first. This is not incorrect behaviour because both events were triggered in the same cycle. Therefore it is possible that those two events are not always handled in the same order in this case. To work around this case we specified another requirement below that one. In this requirement we add to the left hand side that the event manager is in the CHECKING state. This ensures that the previous cycle has finished, so the environment cannot alter the state anymore before all the current events have been managed.

6.2.4 Hierarchical event dispatching

Another property of events is that they can be handled at different nesting depths of the state machine, and depending on the hierarchical structure of the state machine. The active state configuration of a state machine can be seen as a tree where the root node is the current active state of the state machine. If that state is a composite state, it has children which correspond to the active states in those regions. Those states can also be composite states. The leaves of the tree are simple states. When an event is dispatched it is first presented to the leaf nodes. If those cannot handle the event, the event is passed up the tree until it is either handled, or the root of the tree is reached. An example of a complex scenario is shown in Figure 6.5. The active state configuration contains four composite states C1 to C4 and four simple states S1 to S4. If an event is dispatched, the simple states in the active state configuration first get the opportunity to handle it. In this example state S1 and S2 were not able to handle it (and are coloured red), but S4 and S2 could and did (these are coloured green). The event is not dispatched to C4 or C3, because they have at least one descendent which has already handled the event. The event is passed on to C2, because its only descendent did not handle the event.

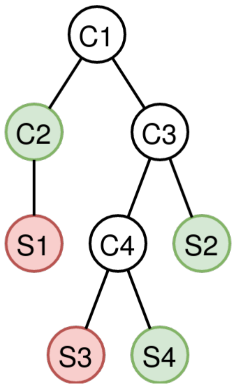


Figure 6.5: Example active state configuration tree

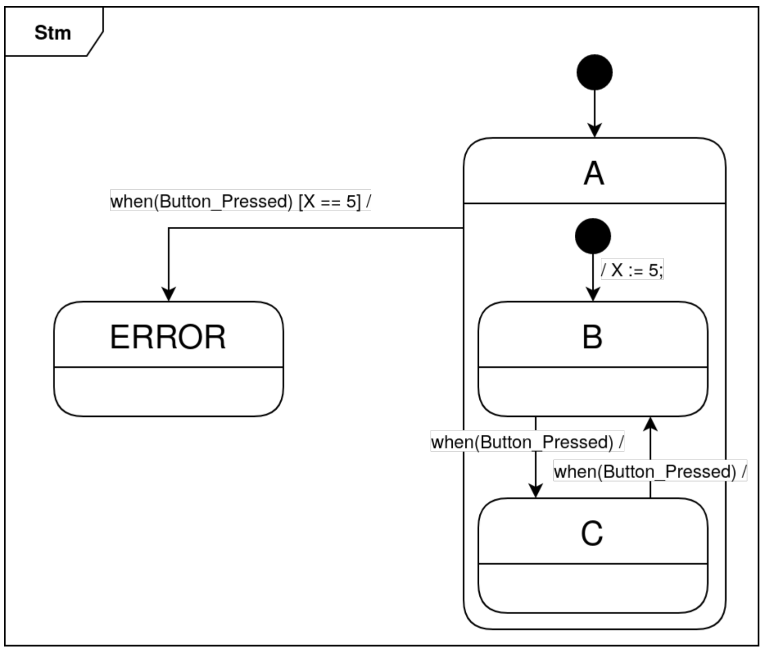


Figure 6.6: Test model for handling events at different levels

In Figure 6.6 we specify a model that tests the case where the composite state can handle event, but its nested states can also handle the event. Therefore the `ERROR` state can never be reached. We added the guard `X == 5` to that transition to make sure that the transition can only be enabled once the active state of the region is either `B` or `C`.

The behaviour of the model is verified using the following requirements:

- `A[] !stm.ERROR_Simple`
- `E<> stm___A_0.B_Simple`
- `E<> stm___A_0.C_Simple`
- `A[] (stm.A_Composite imply !stm___A_0.COMP_WAIT)`
- `A[] (stm___A_0.B_Simple || stm___A_0.C_Simple) imply str_stm.X == 5`
- `(stm___A_0.B_Simple && str_stm.Button_Pressed) --> stm___A_0.C_Simple`
- `(stm___A_0.C_Simple && str_stm.Button_Pressed)`

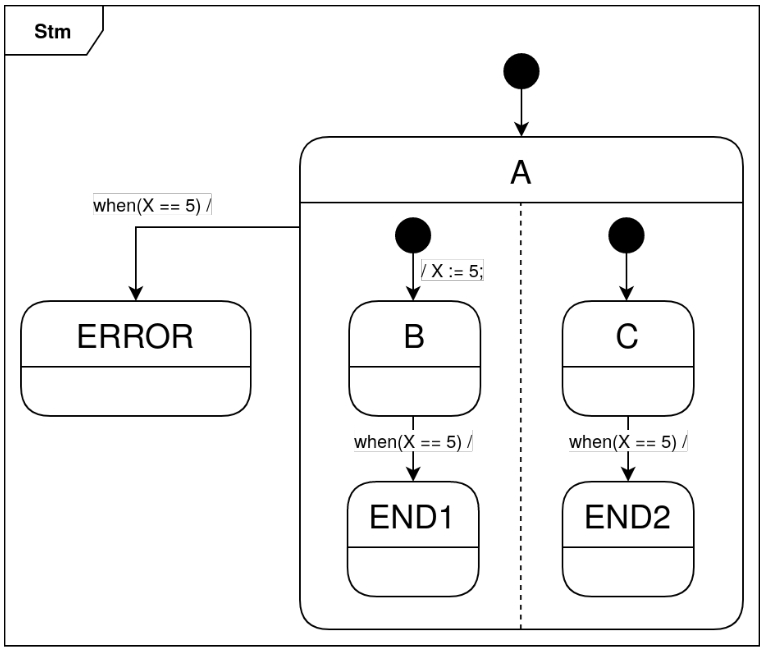


Figure 6.7: Test model for handling events at the same level

```
    --> stm___A_0.B_Simple
```

First we test for the reachability of the states in the composite state region, and test that the `ERROR` state cannot be reached. Next we make sure that the region is properly activated, and that `X` is set to `5` when the region has as active state `B` or `C`. Finally we test that the port `Button_Pressed` being true always results in `B` transitioning to `C` or `C` transitioning to `B`.

To also test that two sibling states in the active state configuration tree are able to both handle the event, we created another example model. This model is shown in Figure 6.7.

This model uses the event of `X` being set to `5`. This event should always be handled by both regions, because they exit their initial state in the same cycle. Therefore, the event cannot be handled by the composite state, because it is already handled by the active states in its regions.

This behaviour is verified with the following UPPAAL requirements:

- `A[] !stm.ERROR_Simple`
- `E<> stm___A_0.C_Simple`
- `E<> stm___A_1.B_Simple`
- `A[] stm___A_1.B_Simple imply str_stm.X == 5`
- `A<> (stm___A_0.END2_Simple && stm___A_1.END1_Simple)`
- `A[] stm___A_1.B_Simple == stm___A_0.C_Simple`
- `A[] (stm___A_0.END2_Simple == stm___A_1.END1_Simple)`

If the model behaves correctly it should always end up in the state configuration `<A, END1, END2>`. This is checked with the fifth property. The rest of the properties check supporting intermediate behaviour.

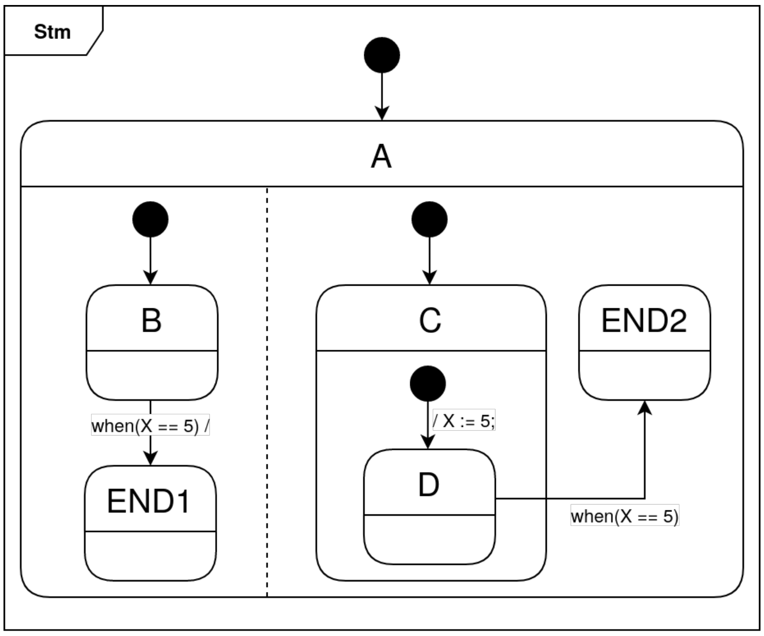


Figure 6.8: Test model for handling events in separate regions at different levels

As a more complex example, we created the model shown in Figure 6.8. This model contains a composite state where one of it regions also contains a composite state. Once the model has reached the state configuration where `A`, `B`, `C`, and `D` are all active the event `X == 5` is dispatched. This event should be handled by both state `B` and state `D`, which results in the state configuration `<A, END1, END2>`.

This behaviour is validated with the following UPPAAL requirements:

- `E<> stm___A_0.B_Simple`
- `E<> stm___A_1.C_Composite`
- `E<> stm___A_1___C_0.D_Simple`
- `E<> stm___A_0.END1_Simple`
- `E<> stm___A_1.END2_Simple`
- `A[] stm___A_1___C_0.D_Simple imply str_stm.X == 5`
- `A<> (stm___A_0.END1_Simple && stm___A_1.END2_Simple)`
- `A[] stm___A_0.B_Simple == stm___A_1.C_Composite`
- `A[] (stm___A_0.END1_Simple == stm___A_1.END2_Simple)`

In checking these requirements, we find that the model cannot reach state `END1`. From looking at the counter examples given by UPPAAL we conclude that the events are only thrown at the deepest level of the tree. When a state on that level can handle the event, the event manager does not continue on other levels. While the current behaviour is incorrect, this implementation is sufficient for this research because this use case does not occur in the models that are used in this research. However, this should be implemented correctly in a later stage to complete the feature support of EULYNX.

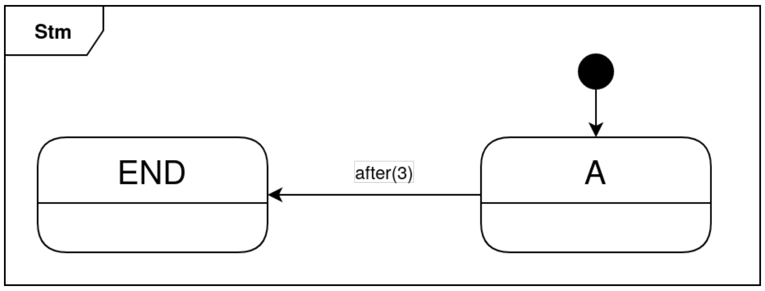


Figure 6.9: Test model for after events in a simple scenario

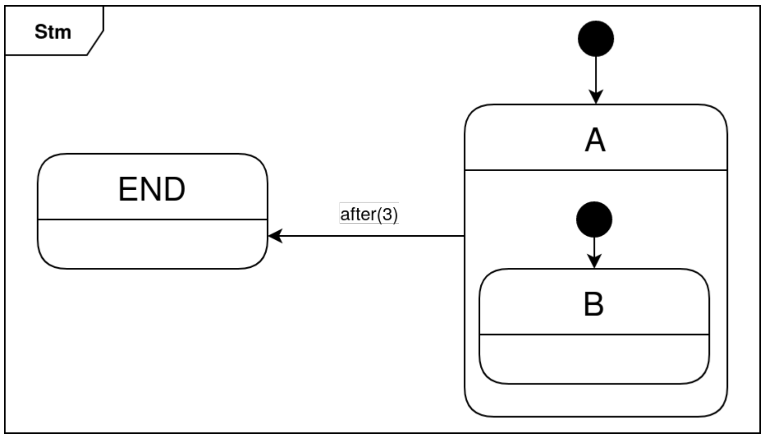


Figure 6.10: Test model for after events with a composite state

6.2.5 After events

After events are events that can be put on transitions and internal behaviour. They trigger after a specified amount of time. In the existing translation to mCRL2, these events are handled as non-deterministic decisions to either stay in the state or take the transition. This simple implementation is sufficient for their research because in the EULYNX models, the specific amount of time does not matter. Therefore, we also aimed for this simple implementation with the possibility of further developing this feature in a later stage of the project.

The first model that gives an example use case of an after event is shown in Figure 6.9.

This model should be able to reach the `END1` state, but it is also allowed to stay in the `A` for multiple cycles. This behaviour is tested with the following requirements:

- `E<> stm.A_Simple`
- `E<> stm.END1_Simple`
- `E[] !stm.END1_Simple`

These requirements simply test for the reachability of all the states, and for the possibility that the `END1` state is not reached. This ensures that the behaviour of the state machine possibly leaving state `A` is present in the example model.

To also test that an after event can interrupt a composite state we created the model shown in Figure 6.10.

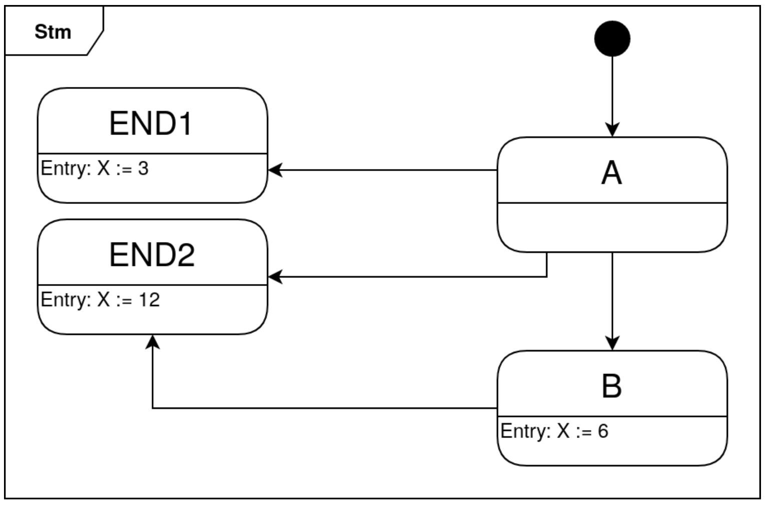


Figure 6.11: Test model for entry behaviour

The behaviour of the model shown in Figure 6.10 is verified with the following UPPAAL requirements:

- `E<> stm.A_Composite`
- `E<> stm___A_0.B_Simple`
- `E<> stm.END1_Simple`
- `E[] !stm.END1_Simple`
- `A[] stm.END1_Simple imply stm___A_0.COMP_WAIT`

6.2.6 Entry and exit behaviour

As the names specify, the entry and exit behaviour of a state is code that is executed either on entry of a state or on exit of a state. There are some tricky orderings when it comes to the exit behaviour nested states with composite states, but this is handled in the next section.

First we will look at the entry behaviour. The test model for this is shown in Figure 6.11. We want to test that the code is always executed before entering the state. In order to do this we modify the property `X` on entry of most states. Because the variable is altered in no other place, we can check whether it is always true that for example `X == 12` in the `END2` state.

We test for this behaviour with the following UPPAAL properties:

- `E<> stm.B_Simple`
- `E<> stm.END1_Simple`
- `E<> stm.END2_Simple`
- `A[] stm.B_Simple imply str_stm.X == 6`
- `A[] stm.END1_Simple imply str_stm.X == 3`
- `A[] stm.END2_Simple imply str_stm.X == 12`
- `stm.B_Simple --> stm.END2_Simple`

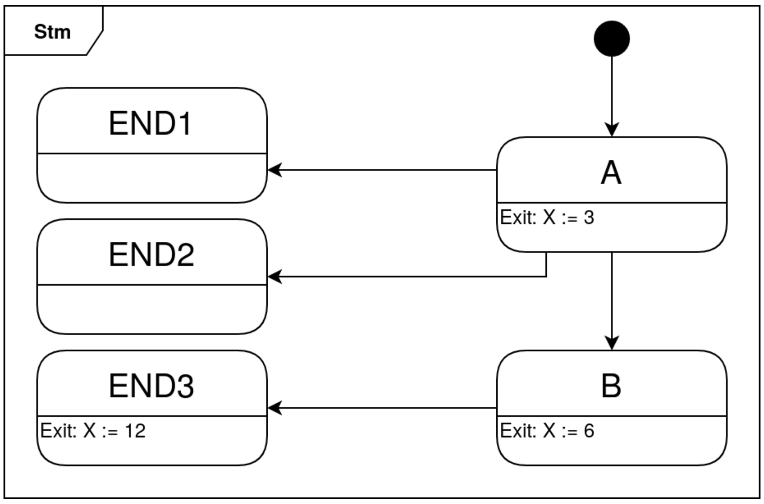


Figure 6.12: Test model for exit behaviour

In the first three properties we check for the reachability of the states. Then, we check if when they are in the state, the correct entry behaviour has executed. Lastly, we verify if state `END2` was also reachable from state `B`. It is important that we check that the same happens in `END2` if we enter it from `A` or `B`.

For testing the exit behaviour we created a model in a similar way. The test model is shown in Figure 6.12. Here we can test for example that in all the successor states of `A`, the property `X == 3` should hold. Another one is that `X == 12` should never be true, because it is not possible to leave state `END3`.

The UPPAAL properties we used to test for this behaviour are:

- `E<> stm.B_Simple`
- `E<> stm.END1_Simple`
- `E<> stm.END2_Simple`
- `E<> stm.END3_Simple`
- `A[] stm.A_Simple imply str_stm.X == 0`
- `A[] stm.B_Simple imply str_stm.X == 3`
- `A[] stm.END1_Simple imply str_stm.X == 3`
- `A[] stm.END2_Simple imply str_stm.X == 3`
- `A[] stm.END3_Simple imply str_stm.X == 6`
- `A[] str_stm.X != 12`

Lastly we test if the entry behaviour, transition effect, and exit behaviour is executed in the correct order. We test this with the model shown in Figure 6.13. A few simple calculations are used that give a different result if their order is changed. The calculations are:

a) $x := (x+2)*2$
b) $x := (x+3)*3$
c) $x := (x+5)*5$

The different outcomes when the order is mixed up is displayed in Table 6.2. When the calculations are done in the correct order (first a, then b, and finally c) the result is 130. This result is different from all other permutations of the calculation order, therefore we

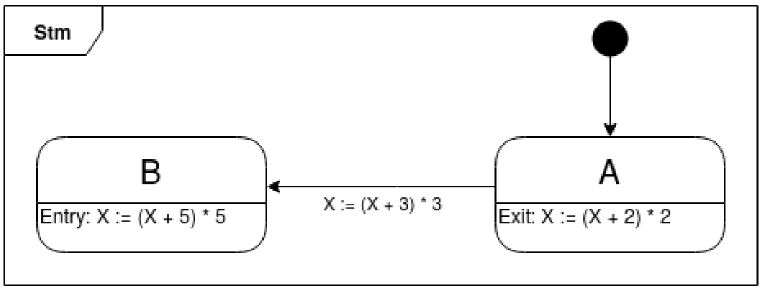


Figure 6.13: Test model for the order of exit behaviour, entry behaviour, and transition effects

Permutation	**Result**
a -> b -> c	130
a -> c -> b	144
b -> a -> c	135
b -> c -> a	144
c -> b -> a	172
c -> a -> b	171

Table 6.2: Results of the different calculation order permutations.

can detect if the different lines of code are executed in the correct order.

In order to check this we need only three properties:

- `E<> stm.B_Simple`
- `A[] stm.A_Simple imply str_stm.X == 0`
- `A[] stm.B_Simple imply str_stm.X == 130`

6.2.7 Composite states

Composite states give the possibility for concurrent behaviour in a single state machine. This gives many possible interleavings, which result into a lot of edge cases. In this section we separate them and put them under test with a couple of example models. The things we want to test are the initialisation of the regions when a composite state is entered, the termination of the regions when a composite state is exited, and the transition to a different state from the composite state. We test these on two example models. One has two regions, and the other has only one region. In the latter we also nest composite states and test the order of the exit behaviour of those nested states.

The first example model is shown in Figure 6.14. Here we model a bridge that is only opened when the two barrier poles are closed. It starts in the `CLOSED` state and transitions to the `CLOSING_POLES` composite state when the `Open_Pressed` event occurs. Then the two regions are initialised. It can only transition to the `OPEN` state when both regions have reached their final state. While the state machine is in the `CLOSING_POLES` state, it can transition at any time to the `CLOSED` state if the `Cancel_Pressed` event is triggered. This should terminate the sub regions.

The behaviour of the model is tested with the following UPPAAL properties:

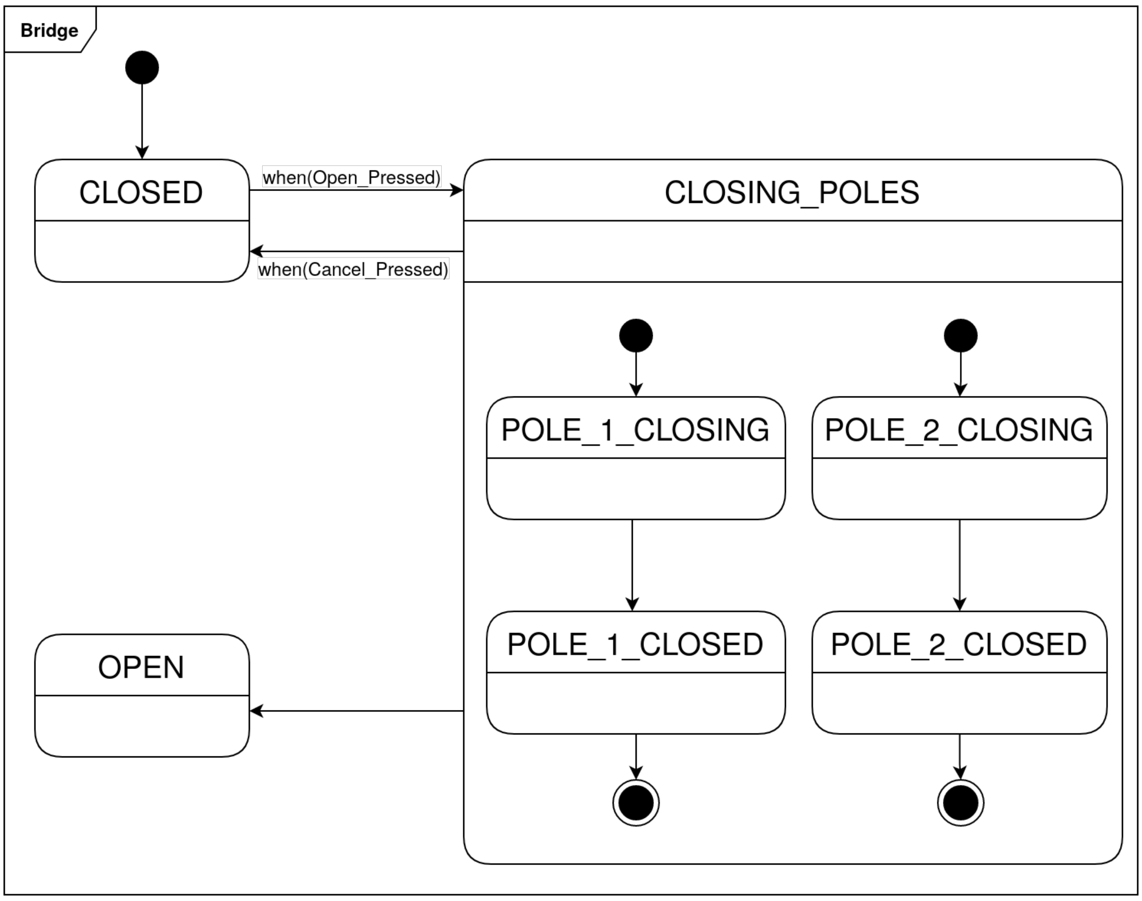


Figure 6.14: Test model for composite state behaviour with two regions

- `E<> bridge.OPEN_Simple`
- `A[] bridge.OPEN_Simple imply (bridge___CLOSING_POLES_0`
  `.COMP_WAIT && bridge___CLOSING_POLES_1.COMP_WAIT)`
- `A[] bridge.CLOSED_Simple imply (bridge___CLOSING_POLES_0`
  `.COMP_WAIT && bridge___CLOSING_POLES_1.COMP_WAIT)`
- `E<> (bridge.CLOSING_POLES_Composite`
  `&& !bridge___CLOSING_POLES_0.COMP_WAIT`
  `&& !bridge___CLOSING_POLES_1.COMP_WAIT`
  `&& str_bridge.Cancel_Pressed)`
- `(bridge.CLOSING_POLES_Composite`
  `&& !bridge___CLOSING_POLES_0.COMP_WAIT`
  `&& !bridge___CLOSING_POLES_1.COMP_WAIT`
  `&& str_bridge.Cancel_Pressed)`
  `--> (bridge.CLOSED_Simple`
  `&& bridge___CLOSING_POLES_0.COMP_WAIT`
  `&& bridge___CLOSING_POLES_1.COMP_WAIT)`
- `(bridge.CLOSED_Simple && str_bridge.Open_Pressed)`
  `--> (bridge.CLOSING_POLES_Composite`
  `&& bridge___CLOSING_POLES_0.Initial1_Initial`
  `&& bridge___CLOSING_POLES_1.Initial2_Initial)`

In the first property we check if the `OPEN` state is reachable. The second property verifies that when the `OPEN` state is reached, the UPPAAL processes of the regions are in their waiting state. This means that sub regions have been properly terminated when entering the `OPEN` state. The same is checked for the `CLOSED` state in the third property. Next we want to verify that the regions are terminated when the `Cancel_Pressed` event is triggered. In order to do that we first check if there is a reachable model state where the regions are activated and the pulse property `Cancel_Pressed` is set to true. This is done with the fourth property in the list. The following property checks that when this is true, eventually the model will end up in a state where the current active state is the `CLOSED` state and the regions have been terminated. In the last property we verify that the regions are always properly activated when the composite state is entered. Specifically it checks that when the model is in the `CLOSED` state and the `Open_Pressed` event is triggered, it eventually reaches a state where the UPPAAL processes of both regions have transitioned from the waiting state to their initial states.

The model shown in Figure 6.15 contains two nested composite states. Once the state machine has entered the composite state `B` it can go any time to state `END1` if the `Button_Pressed` event is triggered. The state `END2` is reached from state `D`. Both these transitions should always terminate the regions. In the termination process of the composite states, the exit behaviour of the innermost state should be executed first, then the exit behaviour of the parent, and lastly the exit behaviour of the topmost state. We use the same calculations as previously used in this section, of which the possible results can be found in Table 6.2. The desired behaviour is tested with the following properties:

- `E<> stm.END1_Simple`
- `E<> stm.END2_Simple`
- `A[] stm.A_Simple imply (stm___B_0.COMP_WAIT`
  `&& stm___B_0___C_0.COMP_WAIT)`
- `(stm.B_Composite && str_stm.Button_Pressed) -->`

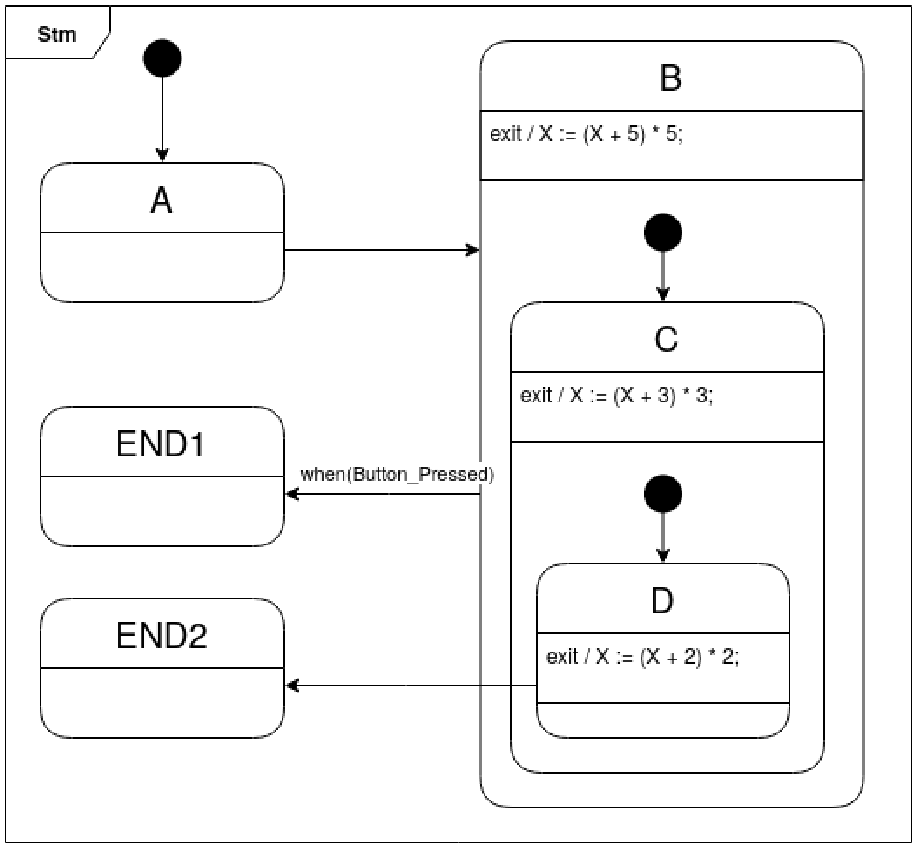


Figure 6.15: Test model for composite state behaviour with one region including nested regions

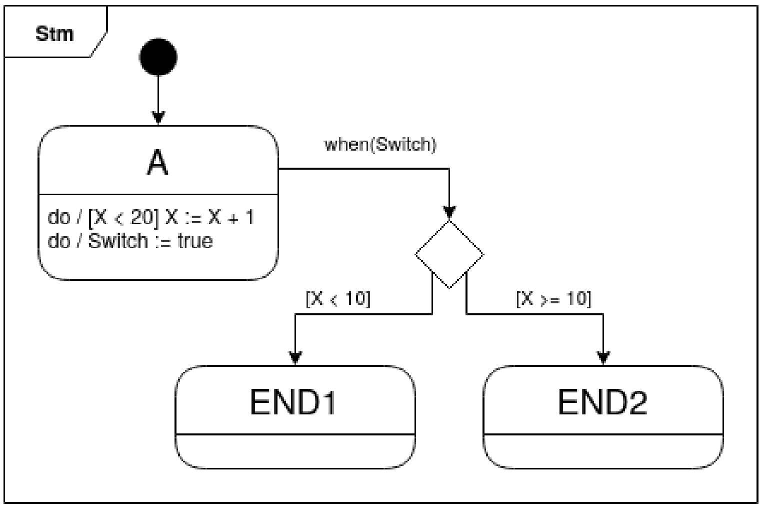


Figure 6.16: Test model for junction vertices

```
  (stm.END1_Simple && stm___B_0.COMP_WAIT
  && stm___B_0___C_0.COMP_WAIT)
```

- `A[] stm.END1_Simple imply (str_stm.X == 130 || str_stm.X == 70 || str_stm.X == 25)`
- `A[] stm.END2_Simple imply str_stm.X == 130`
- `A[] stm.END2_Simple imply (stm___B_0.COMP_WAIT && stm___B_0___C_0.COMP_WAIT)`

First we check for the reachability of the end states. Then, we check if the regions are in their waiting states when the state machine is still in state `A`. The fourth property checks if the state machine always reaches state `END1` where all the sub regions have been terminated if the `Button_Pressed` event is triggered while state `B` is the active state. There are a few different state configurations possible when `B` is the active state of the state machine. It could be that the region of `B` is still in its initial state, or that it has reached `C` and the region of `C` is still in its initial state, or that it has reached `C` and its regions active state is `D`. Only the exit behaviour of the active states is executed, so there are three possible values for `X` in state `END1`. These are checked in the fifth property. In state `END2` there is only one possible value for `X`. This is verified with the sixth property. The last property checks that when `END2` is reached, all the regions have been terminated.

6.2.8 Junction vertices

Junction vertices are used to concatenate transitions. These transitions can be seen as a single transition. All the guards along the transitions need to evaluate to true, and the transition effects are executed in order.

The model that tests this functionality of EULYNX is shown in Figure 6.16. It starts in state `A` where it has as internal behaviour two actions. It can either increment the variable `X` by one, or it can set the variable `Switch` to true.

The UPPAAL properties that test for the correct behaviour in this example model are listed below.

- `E<> stm.END1_Simple`
- `E<> stm.END2_Simple`
- `A[] (stm.END1_Simple imply str_stm.X < 10)`
- `A[] (stm.END2_Simple imply str_stm.X >= 10)`
- `A<> stm.END1_Simple || stm.END2_Simple`
- `stm.A_Simple && str_stm.X < 10 && str_stm.Switch --> stm.END1_Simple`
- `stm.A_Simple && str_stm.X >= 10 && str_stm.Switch --> stm.END2_Simple`

We first check for the reachability of the end states. After that we see if the variable `X` has the correct value in both end states. The state machine should eventually end up in one of the two end states, because the state machine is forced to take a step each cycle. This results in it taking internal behaviour steps until it sets the variable `Switch` to true. Lastly we explicitly define which scenarios should lead to which end states.

7. Usability

In this chapter we evaluate how well we are able to do model checking on the output UPPAAL models of the translation. In doing so we answer research question c). Because the translation is intended for models of components in railway systems, we will be using the Point model [28] as case study. Starting with small subcomponents of this model, and adding more connected parts later on.

For every submodel that we want to validate we first create a list of requirements we want to check. After that, we see if these requirements can be written as UPPAAL properties, and write them if it is possible. Finally, we check the result of these properties and evaluate if they give the expected outcome.

7.1 Point overview

The Point model represents the controlling system of a railway junction. A diagram showing the context of the Point subsystem can be found in Figure 7.1. It communicates with a subsystem called 'Electronic Interlocking'. This is where the model gets its instructions from, and which it needs to report to. The 'Point Machine' moves the switch blades in the railway junction the system is controlling. It reads the current state of the system and gives it instructions.

The complete model contains many blocks, including standard reusable EULYNX blocks. For simplicity, we only look at the SP, FP, and P3 components. An overview of the basic structure of how these components are connected is shown in Figure 7.2. Both SP and FP function as a two way proxy, forwarding commands to the P3 component, and sending updates back. The instructions are either that the junction should move to the left, or that the junction should move to the right. The P3 block is the most complex out of the three.

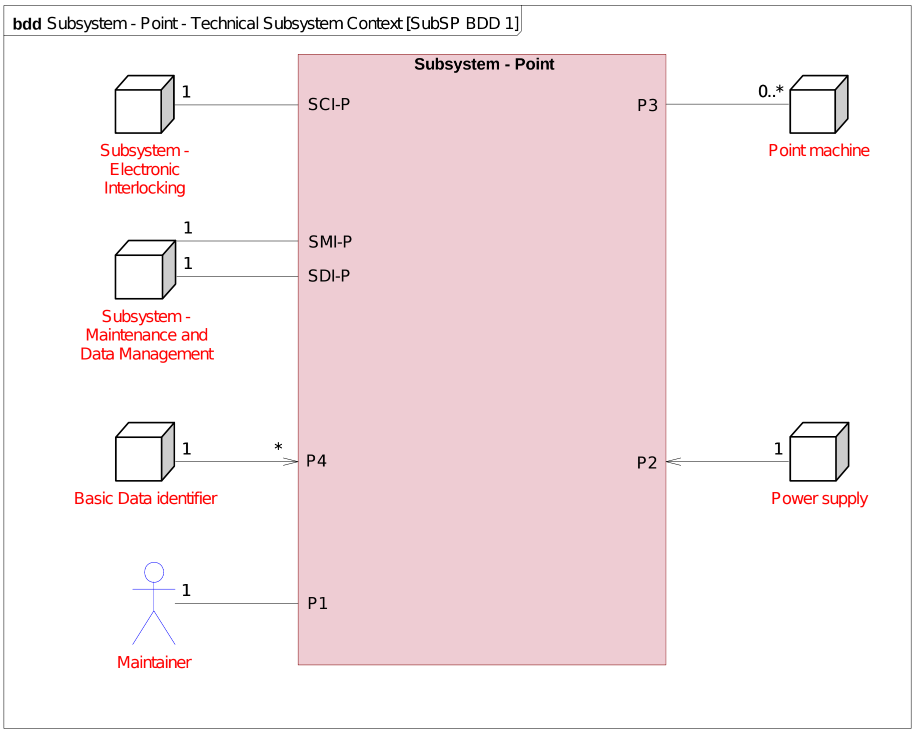

bdd Subsystem - Point - Technical Subsystem Context [SubSP BDD 1]
Subsystem - Point
SCI-P
P3
0..*
1
Subsystem - Electronic Interlocking
Point machine
SMI-P
SDI-P
Subsystem - Maintenance and Data Management
P4
*
P2
Basic Data identifier
Power supply
P1
Maintainer

Figure 7.1: The technical context diagram (Eu.P.950) of Point [28]

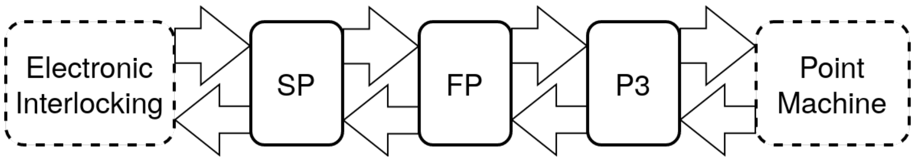


Figure 7.2: Simplified structure of main Point model components

It reads and keeps track of the current status of the junction, and sends the appropriate instructions to the junction based on the given commands. These two tasks are divided over two separate regions.

More detailed diagrams describing the structure of the Point model are included in Appendix C. This also includes the state machine specifications of the SP, FP, and P3 components.

7.2 Requirement specification

In creating the list of requirements we try to think of things a railway engineer would want to verify in their system. It is difficult to ask a railway engineer for this, because they are generally not used to this kind of verification, and it would require more time than is available for this project. We therefore collaborated with FormaSig researchers who have had discussions with railway engineers about this topic.

In verifying the behaviour of EULYNX blocks, the one thing that is focused most upon is that the block gives the correct output values given certain input values. How the block

ID	Requirement	Sub-model
R1	When SP detects a new position, this position is reported to the EIL.	SP
R2	When port D21 of SP reads `"ESTABLISHED"`, SP goes to the `PDI_CONNECTION_ESTABLISHED` state.	SP
R3	When port D21 of SP reads something other than `"ESTABLISHED"`, SP transitions to the `RECEIVING_STATUS_REPORT` state.	SP
R4	When port D21 of SP reads `"ESTABLISHED"`, SP can forward a move command.	SP
R5	SP should only send a new position command to FP if that command was given by the EIL.	SP
R6	When FP detects a new position, this is forwarded to the EIL.	SP+FP
R7	When a new position is commanded by the EIL, this position is forwarded to P3	SP+FP
R8	If the Point machine status is `"LEFT"` and P3 receives a command to go right, it is possible that P3 will instruct the Point machine to go right.	P3
R9	If the Point machine status is `"LEFT"` and P3 receives a command to go right, P3 will instruct the Point machine to go right if no problems occur.	P3
R10	If redrive is enabled, the Point machine reads `"LEFT"`, and is no longer `"LEFT"`, P3 should instruct to go left.	P3
R11	The DT20 port of P3 can only have the values `"LEFT"`, `"RIGHT"`, `"NO_END_POSITION"`, or `"TRAILED"`.	P3
R12	When DT20 reads `"LEFT"` or `"RIGHT"`, D6 should read `"END_POSITION"`	P3

Table 7.1: Requirements for the Point model

implements this is not necessarily important. As long as the observable behaviour is correct, the model can be considered correct. By all means, this is not the only use for EULYNX model specification, but it is the only thing we will focus on in this chapter. This way of testing is similar to black-box testing.

In some cases it is not possible to derive a certain state of the block from its output. In those cases we will reference the state machines state names directly in the requirements if it is necessary.

The list of requirements we created is shown in Table 7.1. For each requirement is also specified what sub-model of Point is needed in order to check this requirement. These components can then be isolated such that UPPAAL does not have to simulate the whole model in order to check these requirements.

A more detailed description of R1 is shown in Table 7.2. The in-depth descriptions of the other requirements can be found in Appendix D in Tables 12.2 to 12.12.

ID	R1
Requirement	When SP detects a new position, this position is reported to the EIL.
Clarification	One of the functions of SP is that it should forward new positions received from FP to the Electronic Interlocking system. That a new position is made available by FP can be detected from the pulse port `T2_Msg_Point_Position`. The actual position can be read from the port `DT2_Point_Position`. This value should be written to SP's output port `DT20_Point_Position`. This port is connected to the EIL.

Table 7.2: Detailed description of R1

Most requirements have the form of, when 'this' holds, 'that' should hold in the future. The requirements R1, R2, R3, R6, and R7 have this form. R4 and R8 add variety by specifying the possibility of something happening as a result of something else instead of it definitely happening. It is useful to add variety to the set of requirements, because this can give us more insights in different types of requirements that are possible to verify and types that are not. R5 turns the logic around by specifying 'this' can only happen if 'that' happened before. R9 has also the structure of if 'this', then eventually 'that', but it adds that 'that' is only guaranteed if another thing does not happen. R10 specifies that a sequence of two events should lead to another state. Lastly, R11 and R12 specify a simple global invariant.

7.3 UPPAAL property specification

The translated UPPAAL requirements are shown in Table 7.3 for as far as they were possible to translate. Additional notes and remarks can be found in Appendix D. In this section we will discuss a few special cases among these requirements in further detail and cover how they represent the original requirement, or why they were not translatable.

Looking at the first requirement R1, we have a clear causal property where if SP receives a position report, it should correctly forward this report to the EIL. For this, we can utilise the 'eventually' operator of the UPPAAL requirement language. In `A --> B`, if A occurs, then in any case B should eventually occur. In our model that could be the next cycle, or even multiple cycles. Focussing on the fact that the forwarded position should always be the same as the received position, we find that we cannot express this in a single UPPAAL requirement. For that you would need a free variable that could be used in the requirement to signify an unknown value, but constant in the requirement. For example: `var_x == X -> var_y == X`. As a workaround we simply duplicate the requirement for every possible position that can be reported.

In R4 we specify that SP should be able to forward a move command if a certain status port is set to `"ESTABLISHED"` (`4` in the translation). To express this in an UPPAAL requirement we can use the 'eventually' operator again, and specify in the left hand side a state where the status port is set to `"ESTABLISHED"` and that it is currently receiving a move command. The right hand side should specify the state in which SP is forwarding the move command. To accurately cover that this holds for every message, we also need to repeat this requirement for every possible message.

ID	UPPAAL requirement
R1	`(str_sp.T2_Msg_Point_Position && str_sp.DT2_Point_Position == <X>) --> (str_sp.T20_Point_Position && str_sp.DT20_Point_Position == <X>)`
R2	`(str_sp.D21_S_SCI_EfeS_Gen_SR_State == 4) --> sp.PDI_CONNECTION_ESTABLISHED_Simple`
R3	`(str_sp.D21_S_SCI_EfeS_Gen_SR_State != 4) --> sp.RECEIVING_STATUS_REPORT_Composite`
R4	`(str_sp.D21_S_SCI_EfeS_Gen_SR_State == 4 && str_sp.T10_Move_Point && str_sp.DT10_Move_Point == <X>) --> (str_fp.T1_Cd_Move_Point && str_fp.DT1_Move_Point_Target == <X>)`
R5	✗
R6	`(str_fp.T20_Point_Position && str_fp.DT20_Point_Position == 2) --> (str_sp.T20_Point_Position && str_sp.DT20_Point_Position == 2)`
R7	`(str_sp.T10_Move_Point && str_sp.DT10_Move_Point== <X>) --> (str_fp.T10_Move && str_fp.DT10_Move_Target == <X>)`
R8	`A[] ((p3___cOp2_All_Left() && str_p3.T1_Move && str_p3.DT1_Move_Target == 9) imply E<> str_p3.D11_Move_Right)` *
R9	✗
R10	`(p3___cOp9_Redrive_Enabled && str_p3.Mem_Current_Point_Position == 3 && !p3___cOp4_All_Left) --> str_p3.D10_Move_Left`
R11	`A[] str_p3.DT20_Point_Position == 3 || str_p3.DT20_Point_Position == 9 || str_p3.DT20_Point_Position == 0 || str_p3.DT20_Point_Position == 8`
R12	`A[] (str_p3.DT20_Point_Position == 3 || str_p3.DT20_Point_Position == 9) == (str_p3.D6_Detection_State == 13)`

Table 7.3: UPPAAL version of the requirements for the Point model

We found that R5 is not possible to translate into an UPPAAL requirement. It is mainly not possible to specify that a certain state did not occur in the past. Another approach is to specify the possibility of the erroneous behaviour and take the negation of that property. In that case we would want to specify that there is a trace where no message has been received (i.e. port T10_Move_Point is `FALSE` the whole time), and at some point it sends a message to FP (by setting port T1_Cd_Move_Point to `TRUE`). This can be specified in CTL by writing `!E (!str_sp.T10_Move_Point U str_sp.T1_Cd_Move_Point)`. UPPAAL does not have an operator that provides the same semantics as the 'until' operator of CTL.

For R8 we are able to specify a property that is theoretically correct, however it cannot be checked with UPPAAL because it contains nested path expressions which are not supported in the current version of UPPAAL. We needed to specify that when a certain property was true, there always existed a path from that in which a different property eventually became true. This is theoretically described with the formula `A[] this imply (E<> that)`. This is slightly different from the 'eventually' operator (`-->`) of UPPAAL, which specifies that something will always eventually happen, given a certain previous state. That operator can be written as `A[] this imply (A<> that)`.

R9 is the last requirement we could not find an UPPAAL translation for. The main difficulty lies in specifying that property 'b' always will become true, given property 'a' is true on the path before that. Even in CTL is this hard to write as a requirement. The property `E a U b` is too loose, because it only says that there exists a path where 'a' leads to 'b'; and the property `A a U b` is too restrictive, because it specifies that 'a' holds on all paths until 'b' holds. This leaves us with `AG message_received imply (E no_problems U command_given)` as the closest approximation.

The requirements R11 and R12 are easily translated, as they are simple invariants. The operator `A[]` in the UPPAAL requirement language can be used here. What we do need to keep in mind is that the invariant does not have to hold when the EULYNX state machine is 'between states'. If for example one variable is altered in the exit behaviour of a state and the other variable is altered in the entry behaviour of the next state, then it could be that in our UPPAAL model there is a 'between' state in which this property does not hold. This can be dealt with by specifying that the requirement should hold in states where all the processes are in a stable state. This can be written as `A[] all_processes_are_stable() imply requirement`.

The rest of the requirements were translated without issues. With only three requirements that could not completely be translated, we believe that the output model and the tool UPPAAL provide enough features to specify simple to moderately complex requirements for a EULYNX model. For slightly more advanced requirements like R5, R8, and R9, the UPPAAL requirement language is a bit too limited.

7.4 Results

In this section we will go over the verification results of the requirements using UPPAAL. We compare the outcome with the result we expected beforehand, and evaluate the feedback UPPAAL gives with regard to counter examples. The requirements are tested in

UPPAAL on a PC with an Intel i7-4710MQ processor, 16 GB RAM, and running Ubuntu 18.04.5 LTS.

It was found that the output models generate a very large state space. Verifying some properties in the SP model take almost five minutes. Three properties of the P3 model cannot be checked at all, because after five minutes UPPAAL gives an ‘Out of memory’ error.

A few measures have been applied to decrease the state space. A big factor which can cause a big state space increase is the event queue size. The SP model has 6 different event types, which if the queue size is three can give a factor 120 $(6*5*4)$ increase in state space. It was necessary to decrease the queue size to one, in order to be able to check all properties of SP. Next to that we set UPPAAL’s state space reduction to ‘aggressive’, selected ‘Compact Data Structure’ for state representation, and tried both the smallest and largest hash table size. However, these settings did not give a enough improvement in performance to be able to check more properties. We give more possibilities to improve performance in Section 10.1. These improvements could significantly increase the performance, but they cost too much time to implement to include in the scope of this project.

R1: When SP detects a new position, this position is reported to the EIL

For the first requirement of the SP component, we expected it to evaluate to false. This should be the case, because there is a state in which SP does not react to a received message report. This is when the active state of the SP component is the RECEIVING_STATUS_REPORT composite state, and the active state of its region is the simple state STATUS_REPORTED.

Checking the requirement using the verify tool of UPPAAL resulted in the property being ‘not satisfied’. When we add the diagnostic trace option we get a counter example which proofs that the model does not satisfy the property.

To show what kind of feedback UPPAAL provides, we created a simplified version of the graphical simulation trace view of UPPAAL for this requirement. It can be found in Figure 7.3. This figure uses the style of the simulator tool in UPPAAL. We simplified this trace by removing some ‘between’ states, to make it easier to read. Each UPPAAL process has a vertical ‘lifeline’ on which the state history is depicted. The figure is read from top to bottom. When one process has a new location and the others do not have a location on that same horizontal line, they are still in the same location as specified earlier in the figure.

The first thing that happens in the model is that SP moves from its initial state to the RECEIVING_STATUS_REPORT state, which is triggered by the orchestrator process. Upon entering this state it signals the process of its region to start, so it moves from its waiting state to the initial state of that region. Next, the environment process sets port T2_Msg_Point_Position to TRUE, and port DT2_Point_Position to 2 (which is the translation for "LEFT"). After that, the orchestrator process signals the event manager to check for new events. The event manager sees the change in T2_Msg_Point_Position and puts the corresponding event in the queue. Directly after that, it pops it from the queue and dispatches the event by sending a broadcast signal. However, because SP is

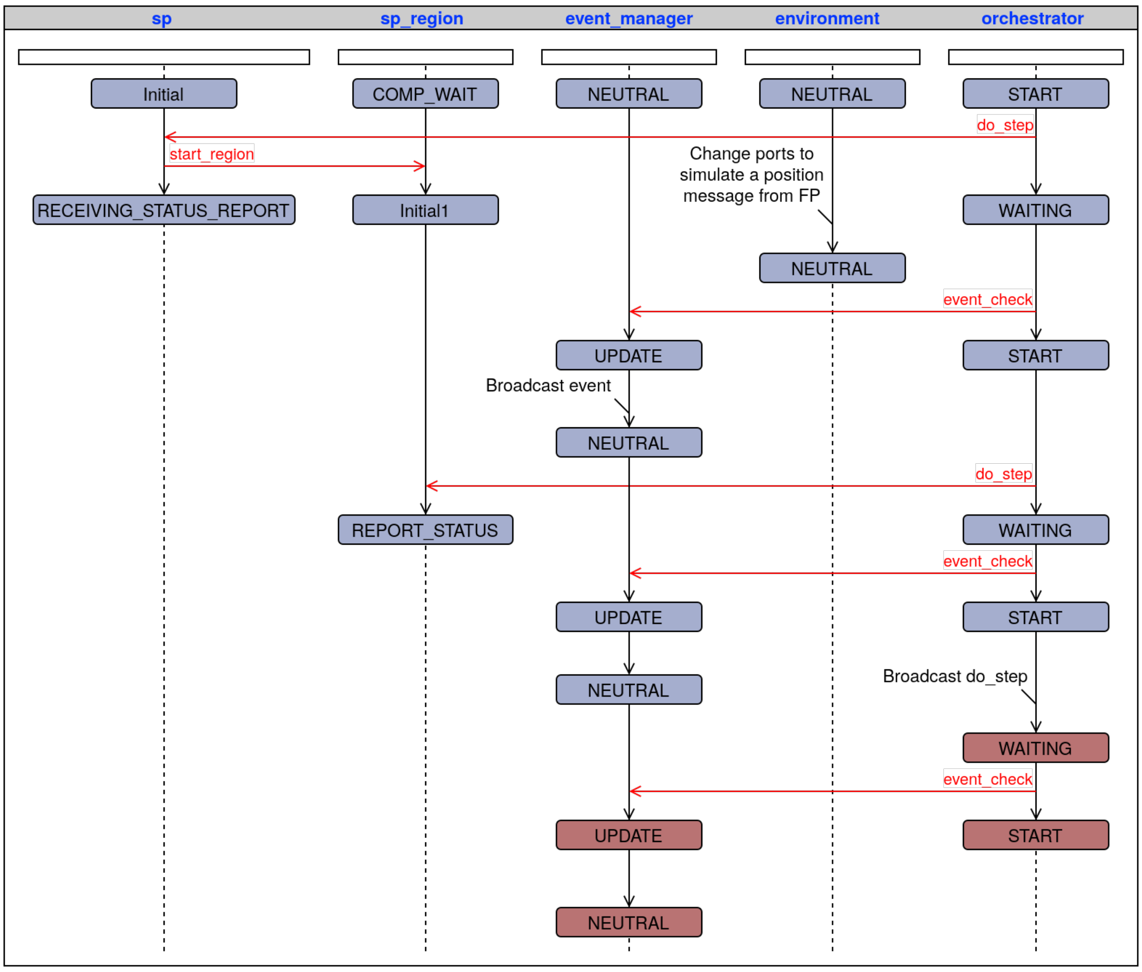


Figure 7.3: Simplified simulation trace of the counter example given by UPPAAL for R1

in `RECEIVING_STATUS_REPORT` and its region is still in `Initial1` there is no process that can handle the event. Therefore, the event is discarded. Next, the orchestrator process makes the region process move from its initial state to `REPORT_STATUS`. Finally the orchestrator process goes through one last cycle before UPPAAL detects that it has reached the same state as before this cycle. This is detected as a loop, and therefore it is possible that the right hand side of the requirement is never satisfied. UPPAAL marks this loop by colouring the repeated locations red.

R2: When port D21 of SP reads "ESTABLISHED", SP goes to the PDI_CONNECTION_ESTABLISHED state

We expect a similar result from this requirement as from R1. When the state machine has as active state configuration `<RECEIVING_STATUS_REPORT, REPORT_STATUS>` it is not able to handle the event. This is also the case if any of the initial states are active.

When we check the requirement using UPPAAL, we find that our expected result was correct. The counter example given by UPPAAL shows the case where the event cannot be handled while the region of `RECEIVING_STATUS_REPORT` is in its initial state.

The simulation trace looks similar to the trace described in the previous section. First the model transitions to the `<RECEIVING_STATUS_REPORT, REPORT_STATUS>` state configuration, then the environment variables are changed, then the event queue is updated, followed by the event being discarded, and finally the model completes one more cycle while nothing has changed. The last cycle is detected by UPPAAL as an infinite loop, which results in the requirement not being satisfied.

R3: When port D21 of SP reads something other than "ESTABLISHED", SP transitions to the RECEIVING_STATUS_REPORT state.

For this requirement we expect that it is satisfied when we check it using UPPAAL. The right hand side of the requirement specifies that `RECEIVING_STATUS_REPORT` is the active state of SP. The left hand side specifies an event that should result in the model transitioning to `RECEIVING_STATUS_REPORT`. Because the SP component has only two states (three including the initial state), it is easy to derive what will happen in both states when the event is triggered. `RECEIVING_STATUS_REPORT` will stay the active state, unless the opposite of the event specified in the requirement occurs. `PDI_CONNECTION_ESTABLISHED` will transition to `RECEIVING_STATUS_REPORT` if the event occurs. Therefore the requirement should be satisfied.

Checking the requirement in UPPAAL resulted in the property not being satisfied. The counter example that UPPAAL produced showed a situation where multiple events were triggered, and that the event queue was filled with other events. This resulted in the event that was relevant for this requirement being discarded. Therefore it is not guaranteed that setting the port `D21_S_SCI_EfeS_Gen_SR_State` to something other than `"ESTABLISHED"` will eventually result in the `RECEIVING_STATUS_REPORT` being an active state.

This result is not useful for us, because it only confirms a known shortcoming of the translation. To combat this we can strengthen the requirement by specifying that when the event corresponding to `D21_S_SCI_EfeS_Gen_SR_State != "ESTABLISHED"` is at the head of the queue, the model should eventually reach `RECEIVING_STATUS_REPORT`. This is written as:

```
(sp_event_queue_peek() == 3) -->
sp.RECEIVING_STATUS_REPORT_Composite
```

This requirement is more specific, because the left hand side now specifies only the moment that port D21 becomes something else than `"ESTABLISHED"` instead of it being something else than `"ESTABLISHED"`. For example this requirement does not cover the case where `D21 != "ESTABLISHED"` becomes true, SP transitions to the state `RECEIVING_STATUS_REPORT`, and after that SP transitions to a different state while `D21 != "ESTABLISHED"` is still true. If from that last state `RECEIVING_STATUS_REPORT` is not reachable, R3 is not satisfied, but the requirement above would be satisfied in that case.

The result of verifying this property in UPPAAL is that it is satisfied. To make sure that the event can be triggered in both states we also check the following two properties:

```
E<> (sp.RECEIVING_STATUS_REPORT_Composite &&
sp_event_queue_peek() == 3)
```

```
E<> (sp.PDI_CONNECTION_ESTABLISHED_Simple &&
sp_event_queue_peek() == 3)
```

Both properties are found to be satisfied. The second property also gives an informative trace. A simplified version of this trace is included in Figure 7.4 to help demonstrate the simulation trace feature of UPPAAL. The complete unedited simulation trace is added in Appendix E.

In this simulation trace is visible how the whole model goes through four cycles. First, SP moves to its composite state and its region is activated. Next, the new cycle starts, so the event manager is triggered. However, no relevant events are dispatched. Next, the environment sets port `T2` to `TRUE`, which is followed by the region transitioning from its initial state to the `REPORT_STATUS` state. Thereafter, the event manager is triggered to check for events, sees that `T2` is set to `TRUE`, and dispatches the corresponding event. This triggers the transition from `REPORT_STATUS` to `STATUS_REPORTED`. Next, the environment sets port `D21` to `"ESTABLISHED"`, which triggers an event in the next cycle which in turn triggers the transition from the composite state to `PDI_CONNECTION_ESTABLISHED`. This puts the region process in its waiting state. Just before the final cycle starts, the environment sets `D21` to `"CLOSED"`. This results in the event labelled with ID `3` being added to the queue. This is the point where the state expression of the requirement evaluates to true, so the simulation trace ends here.

Summary

The complete summary of all the requirement results is shown in Table 7.4. The remaining requirements, that were not already discussed in detail in the previous subsections, were all either not satisfied or UPPAAL encountered an out of memory error. The refined version of R3 is also included in this table as R3’.

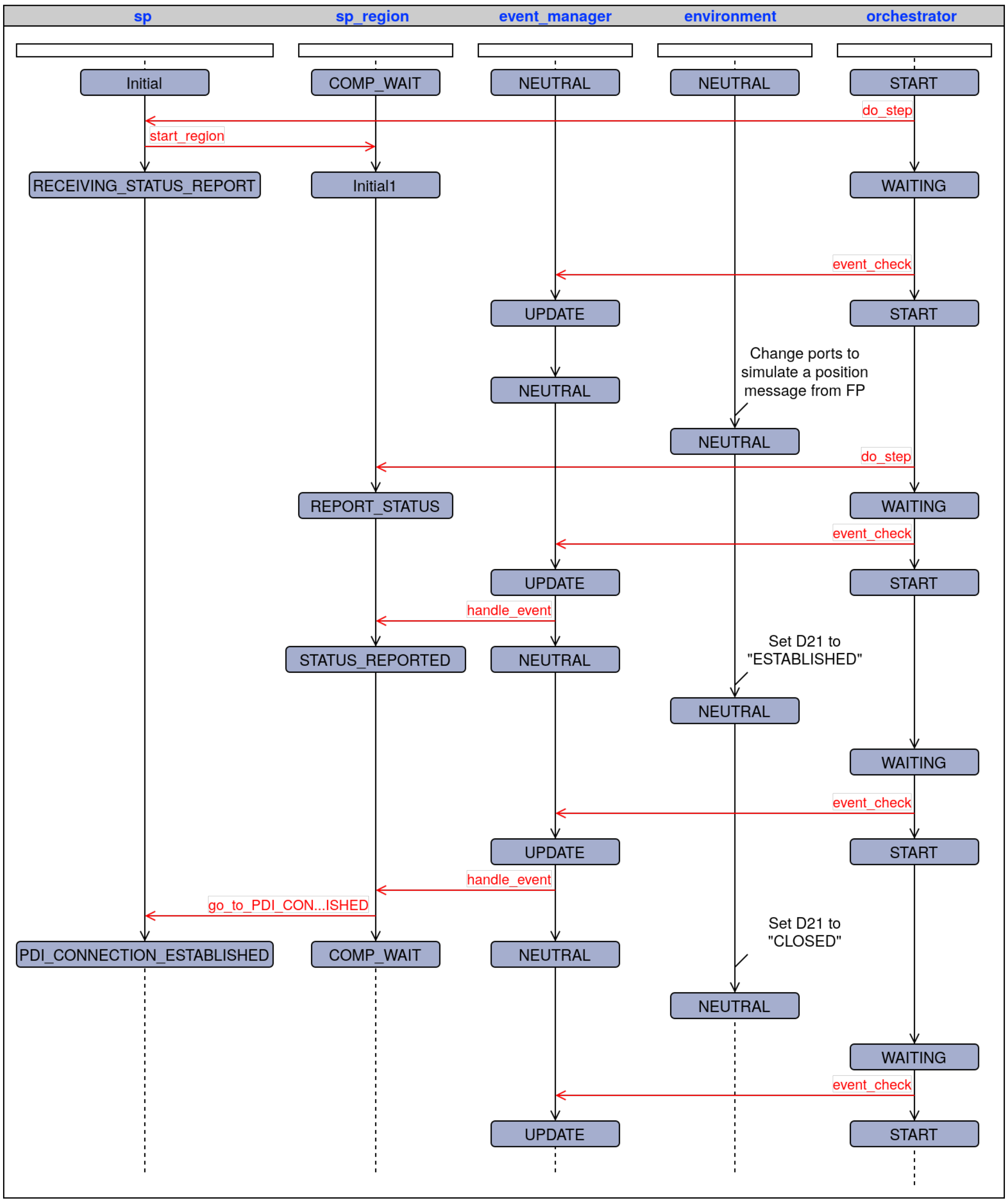


Figure 7.4: Simplified simulation trace for additional UPPAAL requirement

ID	Expected	Actual
R1	🔴	🔴
R2	🔴	🔴
R3	🟢	🔴
R3’	🟢	🟢
R4	🔴	🔴
R5	n/a	n/a
R6	🔴	🔴
R7	🔴	🔴
R8	n/a	n/a
R9	n/a	n/a
R10	🟢	error
R11	🟢	error
R12	🟢	error

Table 7.4: UPPAAL result summary of the requirements for the Point model.

The requirements R1, R2, R4, R6, and R7 are all not satisfied. The last three return similar counter examples to the requirements R1-R3. Just like with those requirements, the events cannot be handled in all possible state configurations. The counter examples are also as clear, readable, and informative as the counter examples that were discussed in the previous subsections.

The requirements R10-R12 could not be checked, because UPPAAL encountered an out of memory error while checking the requirement. This means that UPPAAL ran out of resources before it could completely check the whole model. This is due to the large state space of the model. A reason for why the other properties were possible to check is that UPPAAL does not always have to go through the whole state space if it finds a counter example, because at that point the requirement is disproved. That is if the requirements specifies something that should hold for all traces. If the requirement specifies that something holds for at least a single trace, then disproving it will require the exploration of the whole state space.

8. Discussion

In this chapter we will go over the results of the research, to what extent they answer the research questions, and what the challenges were. First we discuss each supporting research question one by one, followed by the main research question. After that we give some recommendations for future work based on the results of this research.

8.1 Sub research questions

8.1.1 a) Which interpretations of EULYNX fit the goals of the translation to UPPAAL best?

The different interpretations of EULYNX that should be taken into account were discussed in Section 5.1. We found that there were three facets of EULYNX for which we had to decide on a certain interpretation.

Regarding model complexity and usability, the cycle based synchronisation was the best choice, because this allowed us to keep the logic of the different state machines separate by managing them with additional processes. Having the state machines communicate with each other directly through shared memory was a decision that followed naturally, because the publishing of port values to other state machines is a simple operation that only has to read the values from the output port variables and write them to the input port variables. This keeps the UPPAAL processes which are generated from EULYNX state machines clear and readable, by adding a minimal amount of extra logic that is needed to manage the model. However, the generated UPPAAL models are not formatted, and put all the states in a cluttered grid. It takes some manual effort to position the model elements such that it is readable.

With respect to time constraints, we did not include this in the scope of the current project.

It was kept in mind while creating the translation, but it might still be hard to implement this with the current cycle interpretation. The timed elements of EULYNX are also not fully included in the current interpretation of the translation to mCRL2, which put our focus on completing the translation of the other features of EULYNX.

8.1.2 b) Is the translation to UPPAAL correct?

The correctness of the translation is discussed in Chapter 6. We tested the different features of EULYNX with small and simple example models. Each model made use of a specific feature or combination of features, and its behaviour was verified with properties using the UPPAAL verifier tool. Apart from the simple way events are dispatched (Sect. 6.2.4) and the fact that initial states are treated as simple states instead of pseudo states, we are fairly confident that the translation is correct. The two aforementioned defects were discovered during the validation process, and were found to have not enough impact to be added to the scope of the current project.

While our testing strategy covers the translation systematically, it is always difficult to fully verify the complete translation. There are more combinations of features and edge cases possible. In order to be able to say with strong confidence that the translation is correct, a formal correctness proof should be created. This did not fit in the scope of this project, but it is a thing that could be done in the future.

8.1.3 c) How well can the output models be used to verify model requirements?

This research question is discussed in Chapter 7. There we first created a list of requirements that are valuable to check on the EULYNX model Point. Next, we evaluated how we could specify those requirements as UPPAAL properties. For those that were translatable, we created UPPAAL properties that could be checked on the output models of the translation. Finally, we tested those requirements by running them in UPPAAL.

We composed a list of twelve requirements. It was a challenge to compose a list with enough variety to evaluate the model checking capabilities of the translation combined with UPPAAL as thorough as possible. We solved this issue by building the list in collaboration with one of the developers of the translation from EULYNX to mCRL2 [3] who has had discussions with railway engineers about this topic. It might be useful to also collaborate with actual railway engineers to get different insights, but for this project the current approach was sufficient, because the translation is still in its prototype phase and accurately evaluating the ease of use by railway engineers was not part of the main goal of this project. The different requirements focus on separate types of requirements with respect to structure. These cover the basic requirement types a railway engineer would use. It is easy to think of more complex requirements by layering or combining multiple types, but for this research the fundamental requirements are enough. This list gives us enough variety to determine if the translation can be of some use in validating models of the EULYNX standard.

From these twelve requirements we were able to translate nine to UPPAAL properties. The most complex three added an extra layer to their requirement that could not be expressed in UPPAAL. This was the case because the requirement language of UPPAAL is too limited. It would have been useful if these requirements were also possible to verify in UPPAAL, but based on the other nine requirements that were possible to translate we can say that UPPAAL (combined with the output model of the translation) gives us enough expressiveness to do basic model checking. Especially the A 'leads to' B requirements are useful, because EULYNX components are mainly input-output focused. In those cases it is interesting for a railway engineer to be able to specify how a components outputs should react to certain inputs. Regarding the requirements that could not be translated, it might be possible to check them if the requirement is taken into account in the translation and they are partly encoded in the output model. However, this is out of the scope of this project.

Testing the nine translated requirements on their corresponding UPPAAL models resulted in six requirements evaluating to true and three requirements encountering an error during the verification. That most of the requirements would not be satisfied was as expected. The counter examples given by UPPAAL were useful, as they demonstrated clear scenarios where the requirement did not hold. The behaviour was also as we could have predicted looking at the effective semantics of EULYNX state machines. The requirements that were supposed to be false are important to have, because they produced counter examples which gave insight in how useful the feedback is that UPPAAL gives in combination with the output models.

For one requirement we found that the reason it was not satisfied was due to the fact that events were discarded because the finite event queue was full in some cases. To still be able to check this requirement, we rewrote it to only look at the cases where the event did end up in the queue. This requirement did evaluate to true in the UPPAAL verifier. This can become a problem when multiple components are involved, because then there are multiple points where events can be discarded. To work around this, it is possible to split requirements that specify a reaction through multiple components (like SP and FP forwarding a message) into two separate requirements. For example "SP and FP forward a message from P3 to the EIL." can be split up into "FP forwards a message from P3 to SP" and "SP forwards a message from FP to the EIL".

The three remaining requirements could not be checked, because UPPAAL encountered an 'out-of-memory' error before it could finish the verification process. This was due to the large state space of the models under test. To be able to check these requirements on those models we need to improve the performance of the output models by minimising the state space. The fact that these requirements could not be verified is not a big problem in this part of the research. It would have been useful to have more counter examples to evaluate if they turned out to be false, but that we could translate and execute them in UPPAAL is also a positive result.

Next to the case study on the Point model we can also take the verification process of the translation using small example EULYNX models in Chapter 6 into account. We were able to test the behaviour of the different EULYNX elements sufficiently.

All in all we can say that the output models and UPPAAL provide enough capabilities to do basic model checking of EULYNX models. However, the UPPAAL requirement language is not expressive enough for more advanced requirements.

8.1.4 d) How well can the input models of the translation scale in size and complexity such that UPPAAL is still able to check them?

The limitations of the output models generated by the translation in terms of performance were encountered during usability testing in Chapter 7. Here we found that we could not completely model check P3 and the combination of SP and FP. To be able to check all the specified requirements on SP we had to limit the event queue size to one. While it is still useful that we are able to model check individual components of the Point model, the performance of the output models in UPPAAL is not sufficient enough to model check the Point system. While this proves that the translation is not ready to validate complete models of the EULYNX specification, we do believe the performance is sufficient given the prototype status of the translation.

Because we already encountered performance issues while testing for usability, we did not perform specific scalability tests. Instead we put our focus on finding ways in which we could improve the performance of the model.

8.2 Translation improvements

The main issue we encountered while testing that is limiting the use of this translation is performance. Therefore we also looked at how we could increase the performance of the output models. We found three ways in which we could improve translation. These are discussed in more detail in Section 10.1.

The first improvement removes unnecessary ‘between’ locations and combines the sequential steps surrounding that location into a single step. This reduces the state space, and also the number of locations which increases the readability of the output model. This improvement can be safely implemented without changing the model behaviour with respect to the semantics of EULYNX state machines.

The second improvement synchronises the behaviour of separate state machines more strictly such that their run-to-completions happen in sequential order without interleaving. This does not effect the model readability, but only decreases the number of possible model states. Because of the separation of the memory between state machines this change does not affect the behaviour of the model with respect to the semantics of EULYNX state machines. This was not implemented from the start because it was not a must-have and it requires more code to orchestrate the processes more strictly.

The last improvement focuses on how model checkers search through the state space for specific model requirements. In this improvement we propose a change that drastically limits the strength of the ‘leads to’ operator in the requirement language of UPPAAL, but in return it could improve the performance of UPPAAL when checking simple requirements that only specify behaviour that should happen within one cycle. Therefore, it is intended as an optional feature that can be turned off while checking other requirements.

On a final note, there are two EULYNX elements that do not work as they should, but are implemented in a simplified form. These elements are the initial states and the hierarchical dispatching of events. For this research it was sufficient to implement them in this simple form, but these should be implemented correctly in order to consider the translation complete.

8.3 Main research question

How can we create a jEULYNX to UPPAAL translation which can be used on models of railway signalling equipment interfaces?

With the help of the sub questions a) to d) we can conclude that the translation described in Chapter 5 is a good translation that can be used to model check railway signalling equipment interfaces. Most of the modelling possibilities of EULYNX are shown to be correctly translated. UPPAAL provides useful insights in the model by providing detailed traces and a robust requirement language. The output models of the translation are built in such a way that these features of UPPAAL can be utilised. However, the performance of the output models generated by the translation is a limitation. This can be improved by implementing the changes discussed in the upcoming Section 10.1.

8.4 General remarks

A point that also needs to be addressed is the extent to which we managed to take advantage of UPPAAL-specific features, such as time and probability. While we achieved to create a translation that is able to generate UPPAAL models from EULYNX diagrams and get useful insights in those models using UPPAAL, we are not using UPPAAL to its full potential. UPPAAL excels in creating elegant timed automatons and providing insights in those models like variable values over time, average values, and probabilities. Out of those model evaluation features, we only use the path and state expressions of the requirement language. Regarding the model specification, we eventually created a framework with boilerplate model elements to recreate the EULYNX model behaviour in UPPAAL. This was more than anticipated at the start of the project. However, this might be necessary to get EULYNX state machine behaviour in any general model checker, because their behaviour is very domain specific. This makes UPPAAL less suited as target language for the translator. Nonetheless, the graphical model view, accessible requirement specification language, and simulation trace tool are still a great benefit for model checking EULYNX state machines in UPPAAL.

9. Conclusion

In this research we successfully created a translation from EULYNX state machines modelling the behaviour of railway signalling equipment interfaces to the modelling language of UPPAAL. It provides additional aid in model checking these systems because UPPAAL uses different model checking techniques and is more accessible than mCRL2. The main research question that guided this research is:

How can we create a jEULYNX to UPPAAL translation which can be used on models of railway signalling equipment interfaces?

Our success in answering this question is supported by the following conclusions that are drawn from this research. In order to create a translation from EULYNX to UPPAAL the following things are required.

Because the target model of the translation is in a formally defined domain, a clear interpretation of the source model needs to be chosen in order to translate from an informal domain. For this research the best choice was to synchronise the state machines with an execution loop and have them communicate with direct memory sharing. Changes in the memory are handled via an event queue per state machine. This interpretation worked well with model synchronisation and memory sharing features of UPPAAL.

Next, the translation needs translation patterns that create a framework in the target model in order to facilitate certain features that are not directly available in the target modelling language. In our translation to UPPAAL this included for example the orchestrator process and the event manager processes. This provided the right features and it kept the state machine processes simple and close to the original structure of the EULYNX state machines, which satisfies our goal that the target model should be easy to read.

To ensure the quality of the translation, it is necessary to validate it both in terms of correctness and usability. In this research the translation was tested by translating example models and verifying their behaviour with properties in UPPAAL. This systematic approach gave us strong confidence in saying that the translation is correct, even though it is not able to completely proof the absence of errors.

The technique that fit our research the best to validate the usability of the translation was evaluating the checkability of a set of requirements on models part of the EULYNX standard. The set of requirements was created with the needs of railway engineers in mind. This provided enough evidence that the translation can be used successfully by railway engineers. However, there are some intricate but realistic requirements that could not be translated.

As part of the usability of the translation, performance is also an important factor as it determines the scale of the models that can be verified using the translation. We found that the performance of the translation can be improved by decreasing the number of intermediate locations and synchronising the processes more strictly. This is discussed in further detail in Chapter 10. While the translation is not yet suitable for large scale models, we are confident that it can be improved for larger input models by implementing the proposed alterations.

To conclude we can say we succeeded in creating a useful translation that helps with the verification of railway signalling equipment interfaces in the EULYNX standard. As part of the research we tested it for correctness and usability, which gave a positive result. We did find some problems in the usability as a result of state space explosion. To help increase the usability we proposed some improvements on the translation in its current state.

10. Future work

As first recommendation for future work we have three performance improvements. These are further detailed in Section 10.1. To extend the usability of the translation it is important that the performance of the output models are improved such that the translation can be used for larger models.

To get stronger confidence in the correctness of the translation it is also possible to try to prove the translation correct with a formal proof. In this way each facet of the translation can be directly traced to a direct relation between EULYNX model elements and UPPAAL model elements. This is a difficult task as the semantics of EULYNX state machines have no formal definition. However, the modelling language of UPPAAL does have a formal definition.

There are two known shortcomings of the translation in terms of coverage of EULYNX elements. For a more complete translation the hierarchical event dispatching and initial states need to be properly implemented. The current state of implementation is sufficient in simple cases.

Finally it is interesting to look at implementing the timed elements of EULYNX using the clocks of UPPAAL. This enables the use of additional model checking features in UPPAAL such as calculating the average time it takes until a certain state is reached, and therefore make the translation more valuable. However it does also require work on the modelling side, because the models of the EULYNX standard contain very little time information.

10.1 Performance improvements

In this section we propose three improvements on the translation that could increase the performance of the output model. These improvements could not yet be implemented

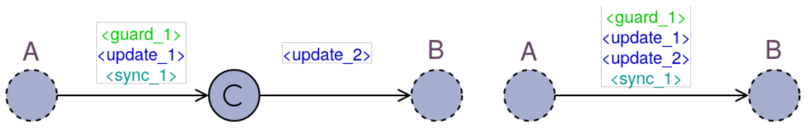


Figure 10.1: Removal of a committed location with a single incoming edge

because they did not fit in the scope of this project. As shown in Chapter 7, not all the requirements of the components and component combinations of the Point model could be checked because the state space of these output models are too large. The improvements in this chapter can help towards the checkability of these components in UPPAAL.

10.1.1 Static location reduction

At some points in the translation we add extra 'between' locations which are not always necessary. This can occur for example at the point where the entry behaviour of a EULYNX state is placed. In specific cases the extra location can be removed and its incoming and outgoing transition can be combined into one. When this is not performed, the model generates more unnecessary states during the simulation. These states are unnecessary because in EULYNX we are only interested in the model states where the state machines are done with their run-to-completion of travelling between states. Removing a single 'between' state might seem like a minor performance improvement, but depending on the update statements on both transitions and the possible interleavings with other processes the removal could greatly impact the state space of the output model. Additionally it also increases the readability of the output model by removing extra locations. This makes it easier to link UPPAAL elements back to EULYNX elements and the vise versa.

It is difficult for the translator to determine if the behaviour of the next transition can be added to the previous transition. Implementing this could increase the complexity of the translator code. Therefore we propose a post-processing step that searches through the output model and removes between locations by combining the incoming and outgoing transitions of those locations.

The first example in which we can simplify the model by removing a between state is shown in Figure 10.1. In this example the committed location in the middle has a single incoming and a single outgoing edge. The outgoing edge can contain zero or more update statements, while the incoming edge can contain zero or more guards, update statements, and channel synchronisations (both sending and receiving). It is important that whether or not the outgoing edge is enabled is dependent on the behaviour of the incoming edge. This is not the case in this example, because the outgoing edge has no guards and no channel synchronisations. Therefore, the two transitions can be combined into the single transition shown on the right side of the figure. It is important that the update statements of the outgoing edge are placed after those of the incoming edge to preserve the execution order of these statements.

The same principle can be applied to a scenario where the committed location has multiple incoming edges. This example is shown in Figure 10.2. If the outgoing edge of the committed location has only update behaviour, then the committed location can be eliminated completely by connecting the incoming edges to the successor location and adding the update behaviour to those edges.

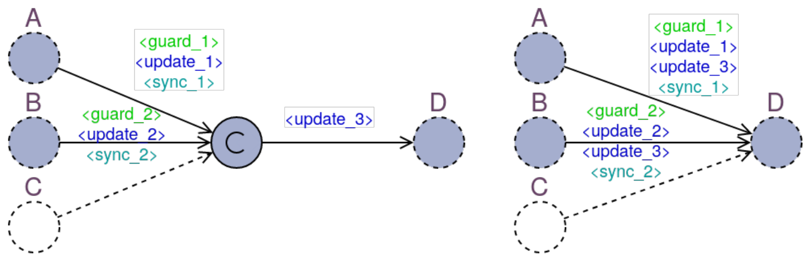


Figure 10.2: Removal of a committed location with multiple incoming edges

Finally, there are some more complex cases where the committed location can be removed. When both the incoming and outgoing edges have guards and update statements we need to check for concurrency conflicts. We can assume that the transition conditions specified in EULYNX are already atomic and are only placed on outgoing transitions of non-committed locations, so we only need to take into account the conditions and update behaviour that are added during the translation to help synchronise the processes. These are references to the variables `..._is_stable` and `..._is_stopped`, and the function call `all_processes_are_stable()`. When only the incoming edge or only the outgoing edge contain some of these expressions, then the two transitions can be combined into a single transition. This scenario can also be applied to the situation where the outgoing edge has a channel synchronisation. It can be combined with the incoming edge if the incoming edge does not contain one of the aforementioned expressions.

The post-processing step can use these scenario descriptions to find committed locations and refactor the model in order to remove them. This step can then be repeated until it does not change the model anymore.

10.1.2 Process synchronisation

When multiple processes run in parallel their different interleaving possibilities can result in an increase in size of the state space. Our translation creates a separate process for each state machine and region in the EULYNX model. Every EULYNX cycle, the output model orchestrates every process to do one step. This step can for a single process contain multiple ‘between’ states where the process handles different tasks sequentially that are part of this run-to-completion. In the current version of the translation, all the processes are triggered simultaneously to start a run-to-completion and how these processes interleave until they are all done is non-deterministic. For some processes it does not matter how they interleave, so we can try to run them sequentially to eliminate some interleaving possibilities.

Processes that are created from different EULYNX state machines operate on separate parts of the memory. They only access their own struct, which contains the input ports, output ports, and properties of that state machine. At the end of each cycle the values of the ports are published to the other state machines connected to those. Therefore it does not matter in what order the processes of the state machines do their run-to-completion of a cycle. The sub-processes of a state machine do operate on the same memory as the main process of a state machine, so their interleaving behaviour should remain non-deterministic.

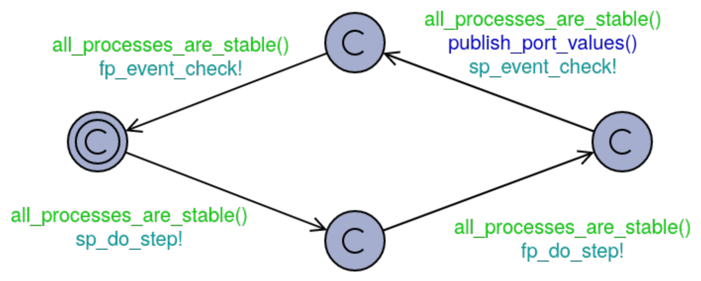

all_processes_are_stable()
fp_event_check!
all_processes_are_stable()
publish_port_values()
sp_event_check!
all_processes_are_stable()
sp_do_step!
all_processes_are_stable()
fp_do_step!

Figure 10.3: Revised orchestrator process for improved process synchronisation

To execute the run-to-completions of different state machines separately, we need to change the orchestrator process. Instead of having all processes listen on the general `do_step` and `event_check` channels, they should listen on state machine specific channels like `<stmName>_do_step` and `<stmName>_event_check`.

An example of how the orchestrator would look with two state machines called SP and FP is shown in Figure 10.3. It starts in the leftmost location where it waits for all the processes to be in a stable state. This is always the case at the start of the simulation of a model generated with the current version of the translation. Therefore, it takes the first transition which sends a broadcast signal to all the processes of SP to take a step if they are able to. Then, it waits until all the processes are in a stable state again, and it does the same for FP. When FP has also finished its run-to-completion the orchestrator transitions to the location at the top of the figure. This publishes the values of the ports to the other state machines, and simultaneously signals the event manager of SP to check for events and dispatch them if there are any. When all processes are in a stable state again, it also triggers the event manager of FP to do the same, which brings the orchestrator to the location it started in.

10.1.3 Separating cycles

During the verification of the requirements for the Point model in UPPAAL we thought of a different way in which we could make the output model easier to verify for UPPAAL. Specifically for the requirements that use the ‘leads to’ operator (`-->`). Lets take as example a requirement in the form of `A --> B`. During the verification if UPPAAL encounters a state in which `A` holds, then it has to explore all the possible traces starting from that state to see if `B` will eventually hold in all of those traces. Because in the current version of the translation the model is in a committed state at all times, the depth of this trace tree can become very large if UPPAAL does not encounter a loop early in the trace.

It could be that for some requirements you are only interested in the direct response of the model to a certain input, meaning for example that a component is presented with a certain input and that it should give a specific output right after a single cycle. If this is the case UPPAAL could stop the search right after a single cycle.

This would remove the possibility of checking other requirements that use the ‘leads to’ operator specifying behaviour that should happen after more than one cycle. Therefore, this should be implemented as a feature that can be toggled on and off.

To realise this, the model should be in a non-committed state between cycles. Because

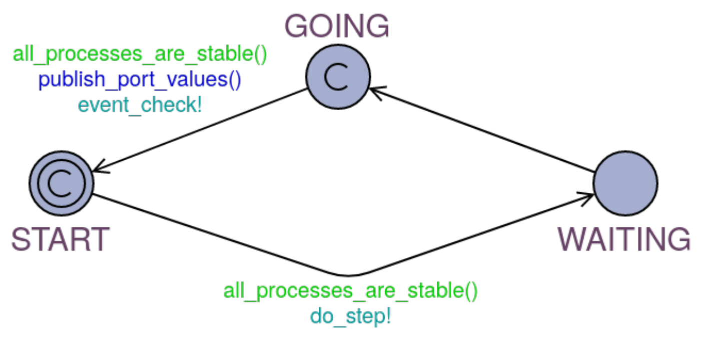


Figure 10.4: Revised orchestrator process for simple ‘leads to’ requirements

the output model does not currently use time constraints, UPPAAL will detect this state as a state in which it can stay indefinitely as it is not triggered to leave that state. All the main locations of EULYNX states are already non-committed as well as the initial states of the event managers. The only non-committed state that needs to be added is one before triggering the event managers in the orchestrator process. The altered orchestrator process is shown in Figure 10.4.

An UPPAAL requirement specifying behaviour of one cycle would have the following form:

```
A && orchestrator.GOING --> B
```

11. References

[1] *A Vending Machine*. (Visited on 11-2-2021). Technische Universiteit Eindhoven. URL: `https://www.mcrl2.org/web/user_manual/tutorial/machine/index.html#first-variation` (cited on page 33).

[2] *ASAL structured action language (SySim)*. (visited on 19-11-2021). PTC Inc. URL: `https://support.ptc.com/help/modeler/r9.0/en/index.html#page/Integrity_Modeler/sysim/SySim_Atego_structured_action_language.html` (cited on page 21).

[3] Mark Bouwman, Bas Luttik, and Djurre van der Wal. “A Formalisation of SysML State Machines in mCRL2”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. 2021, pages 42–59 (cited on pages 12, 18, 48, 88).

[4] Julian Bradfield and Igor Walukiewicz. “The mu-calculus and model checking”. In: *Handbook of Model Checking*. Springer, 2018, pages 871–919 (cited on page 32).

[5] *COMMISSION IMPLEMENTING DECISION (EU) 2017/863*. (Visited on 25-10-2021. European Union, 2017. URL: `https://eur-lex.europa.eu/eli/dec_impl/2017/863/oj` (cited on page 108).

[6] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009, pages 549–552 (cited on page 48).

[7] K. Cui et al. “Unifying Modeling and Simulation Based on UML Timing Diagram and UPPAAL”. In: *2010 Second International Conference on Computer Modeling and Simulation*. Volume 1. 2010, pages 26–30. DOI: `10.1109/ICCMS.2010.125` (cited on page 17).

[8] Alexandre David. *Uppaal Timed Automata Parser library*. (Visited on 10-2-2021). URL: `http://people.cs.aau.dk/~adavid/utap/syntax.html` (cited on page 24).

[9] *EULYNX*. (Visited on 9-2-2021). EULYNX Consortium. 2020. URL: https://eulynx.eu (cited on pages 12, 19).

[10] *File Formats*. (Visited on 10-2-2021). UPPAAL. 2020. URL: https://docs.uppaal.org/toolsandapi/file-formats (cited on page 24).

[11] *FormaSig: Formal methods in railway signalling infrastructure standardisation processes*. (Visited on 29-11-2021). University of Twente. Apr. 2020. URL: https://www.utwente.nl/en/eemcs/fmt/research/projects/formasig/ (cited on page 12).

[12] J.F. Groote et al. "The mCRL2 toolset". English. In: *Informal proceedings of the International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008, Paphos, Cyprus, July 8, 2008; co-located with ECOOP)*. 2008, pages 5–1/10 (cited on pages 11, 30).

[13] Jan Friso Groote et al. "The Formal Specification Language mCRL2". In: *Methods for Modelling Software Systems (MMOSS)*. Edited by Ed Brinksma et al. Dagstuhl Seminar Proceedings 06351. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. URL: http://drops.dagstuhl.de/opus/volltexte/2007/862 (cited on page 30).

[14] K. Havelund et al. "Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL". In: *Proceedings Real-Time Systems Symposium*. 1997, pages 2–13. DOI: 10.1109/REAL.1997.641264 (cited on page 18).

[15] Xiaopu Huang et al. "MDE-based verification of SysML state machine diagram by UPPAAL". In: *International Conference on Trustworthy Computing and Services*. Springer. 2012, pages 490–497 (cited on page 17).

[16] Henrik Jensen, Kim Larsen, and Arne Skou. "Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL". In: *BRICS Report Series* 3.24 (Jan. 1996). DOI: 10.7146/brics.v3i24.20005. URL: https://tidsskrift.dk/brics/article/view/20005 (cited on page 18).

[17] Stuart Kent. "Model Driven Engineering". In: *Integrated Formal Methods*. Edited by Michael Butler, Luigia Petre, and Kaisa Sere. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 286–298. ISBN: 978-3-540-47884-3 (cited on page 48).

[18] Alexander Knapp and Stephan Merz. "Model checking and code generation for UML state machines and collaborations". In: *Proc. 5th Wsh. Tools for System Design and Verification* (2002), pages 59–64 (cited on pages 17, 40).

[19] Kim G Larsen, Paul Pettersson, and Wang Yi. "UPPAAL in a nutshell". In: *International journal on software tools for technology transfer* 1.1-2 (1997), pages 134–152 (cited on pages 11, 18, 23).

[20] FREDRIK LARSSON, PAUL PETTERSSON, and WANG YI. "Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction". In: *Real-Time Systems* 1 (), page 28 (cited on page 28).

[21] Mathias Legrand and Vel. *The Legrand Orange Book (LaTeX Template) v2.4*. LICENSE: CC BY-NC-SA 3.0 (http://creativecommons.org/licenses/by-nc-sa/3.0/). Sept. 2018. URL: http://www.LaTeXTemplates.com (cited on page 2).

[22] Magnus Lindahl, Paul Pettersson, and Wang Yi. “Formal design and analysis of a gear controller”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 1998, pages 281–297 (cited on page 18).

[23] S.P. (Bas) Luttik. “What is the Point: Formal Analysis and Test Generation for a Railway Standard”. English. In: *Proceedings of the 30th European Safety and Reliability Conference and the 15th Probabilistic Safety Assessment and Management Conference*. Edited by Piero Baraldi, Francesco Di Maio, and Enrico Zio. Nov. 2020, pages 921–928. ISBN: 978-981-14-8593-0. DOI: `10.3850/978-981-14-8593-0_4410-cd` (cited on pages 12, 18).

[24] Jasen Markovski and Michel A Reniers. “Verifying liveness in supervised systems using UPPAAL and mCRL2”. In: *International Conference on ICT Innovations*. Springer. 2012, pages 295–304 (cited on pages 18, 35).

[25] *mu-Calculus*. (Visited on 11-2-2021). Technische Universiteit Eindhoven. URL: `https://www.mcrl2.org/web/user_manual/language_reference/mucalc.html` (cited on page 32).

[26] André LN Muniz, Aline MS Andrade, and George Lima. “Integrating UML and UPPAAL for designing, specifying and verifying component-based real-time systems”. In: *Innovations in Systems and Software Engineering* 6.1 (2010), pages 29–37 (cited on pages 17, 40).

[27] *Published documents*. (Visited on 25-10-2021). EULYNX Consortium. 2020. URL: `https://www.eulynx.eu/index.php/documents/published-documents` (cited on page 108).

[28] *Requirements specification for subsystem Point*. Eu.Doc.36 v2.7 (0.A). EULYNX Partners. URL: `https://www.eulynx.eu/index.php/documents/published-documents` (cited on pages 16, 18, 75, 76, 108–112).

[29] *SySim model execution loop*. (Visited on 2-9-2021). PTC Inc. URL: `https://support.ptc.com/help/modeler/r9.0/en/index.html#page/Integrity_Modeler%2Fsysim%2FSySim_model_execution_loop.html%23` (cited on pages 38, 39).

[30] *The Rope Bridge*. (Visited on 11-2-2021). Technische Universiteit Eindhoven. URL: `https://www.mcrl2.org/web/user_manual/tutorial/ropebridge/index.html` (cited on page 30).

[31] Yann Thierry-Mieg and Maximilien Colange. “Experiments using XTA and ITS-tools”. In: (2020) (cited on page 24).

[32] JE Wiggelinkhuizen, GJ Tretmans ESI, and E Hendriksen. “Feasibility of formal model checking in the Vitatron environment”. In: (2007) (cited on page 11).

12. Appendices

A EULYNX functions of Torch

Listing 12.1: The helper functions of the Torch IBD in EULYNX.

```
init() {
  torch_location := "NEAR";
  v1_torch_time := 0;
  v2_torch_time := 0;
  v3_torch_time := 0;
  v4_torch_time := 0;
}
setTorchTime() {
  torch_time := 0;
  if v1_requests_torch and v1_location == torch_location then torch_time :=
       max(torch_time, 1); end if
  if v2_requests_torch and v2_location == torch_location then torch_time :=
       max(torch_time, 2); end if
  if v3_requests_torch and v3_location == torch_location then torch_time :=
       max(torch_time, 5); end if
  if v4_requests_torch and v4_location == torch_location then torch_time :=
       max(torch_time, 10); end if
  if v1_requests_torch and v1_location == torch_location then v1_torch_time
       := torch_time; end if
  if v2_requests_torch and v2_location == torch_location then v2_torch_time
       := torch_time; end if
  if v3_requests_torch and v3_location == torch_location then v3_torch_time
       := torch_time; end if
  if v4_requests_torch and v4_location == torch_location then v4_torch_time
       := torch_time; end if
}
getRequestCount(): Integer {
  requestCount := 0;
  if v1_requests_torch and v1_location == torch_location then requestCount
      := requestCount + 1; end if
  if v2_requests_torch and v2_location == torch_location then requestCount
      := requestCount + 1; end if
  if v3_requests_torch and v3_location == torch_location then requestCount
      := requestCount + 1; end if
  if v4_requests_torch and v4_location == torch_location then requestCount
      := requestCount + 1; end if
  return requestCount;
}
allVikingsMoved(): Boolean {
  if v1_torch_time > 0 and v1_requests_torch then return FALSE; end if
  if v2_torch_time > 0 and v2_requests_torch then return FALSE; end if
  if v3_torch_time > 0 and v3_requests_torch then return FALSE; end if
  if v4_torch_time > 0 and v4_requests_torch then return FALSE; end if
  return TRUE;
}
isDone(): Boolean {
  return v1_location == "FAR" and v2_location == "FAR" and v3_location == "
      FAR" and v4_location == "FAR";
}
moveTorch() {
  v1_torch_time := 0;
  v2_torch_time := 0;
  v3_torch_time := 0;
  v4_torch_time := 0;
  if torch_location == "NEAR" then
    torch_location := "FAR";
  else
    torch_location := "NEAR";
  end if
}
```

B Output UPPAAL model for the guard test example

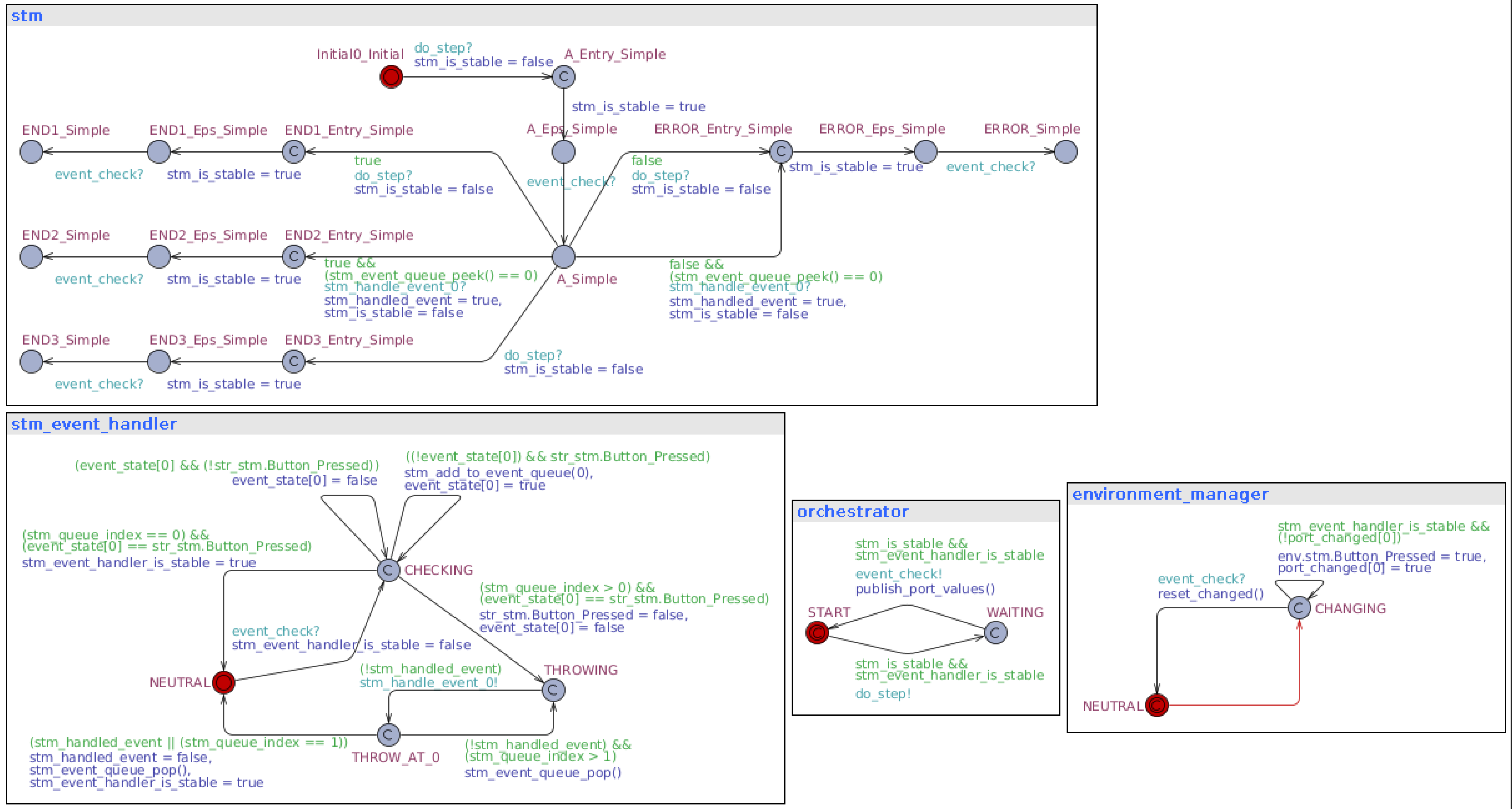


Figure 12.1: Output UPPAAL model for the guard test example

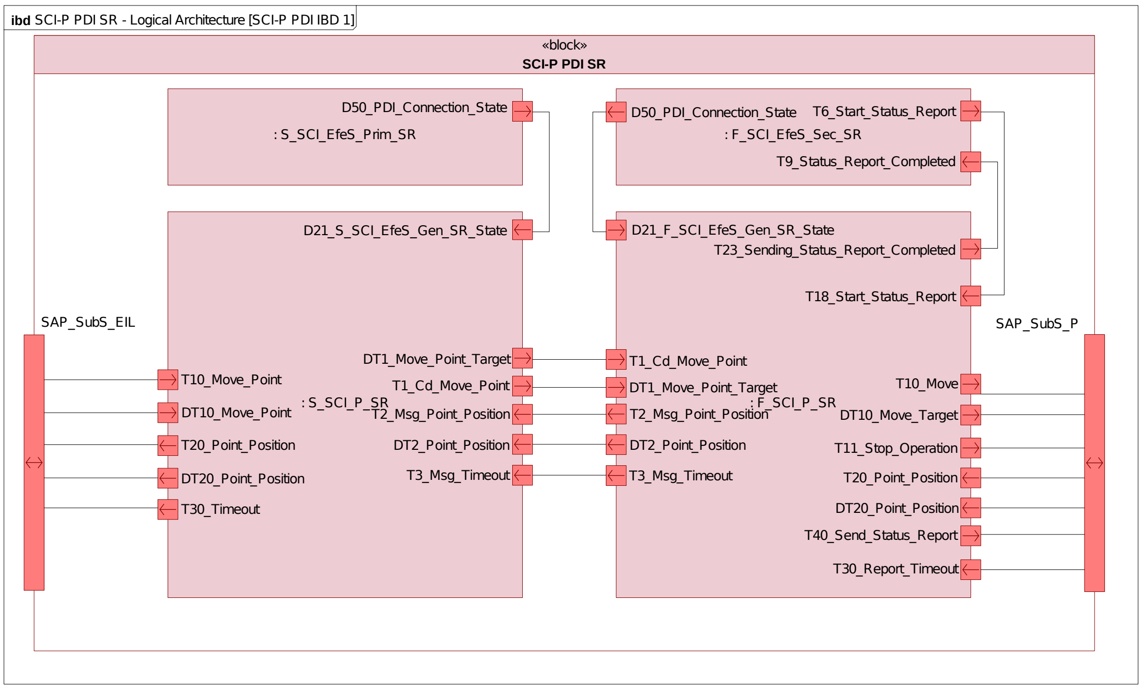


Figure 12.2: Logical architecture of the SP and FP component (Eu.P.3286) in Point [28]

C State machine diagrams of the Point model

The following diagrams are direct copies from the Point model document [28], which can be found in the published documents of EULYNX [27]. These documents are licensed under the EU Public License [5].

How the SP (S_SCI_P_SR) and FP (F_SCI_P_SR) blocks are connected is shown in Figure 12.2. The SP component is the large block on the left, and the FP component is the large block on the right. The connection between the FP and P3 (F_P3_SR) blocks is shown in Figure 12.3. Here, FP is the large block on the left, and P3 is the large block on the right.

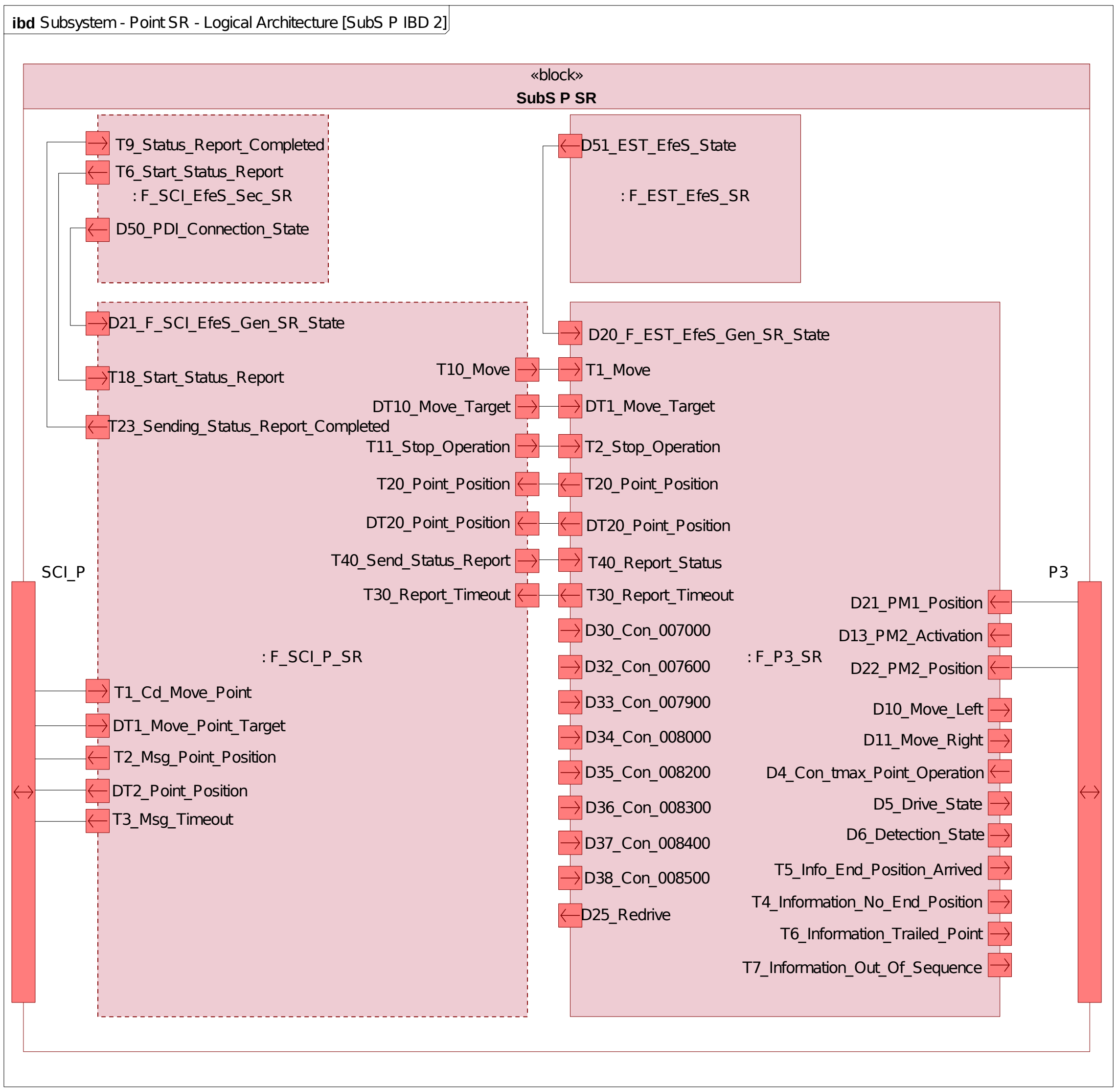


Figure 12.3: Logical architecture of the FP and P3 component (Eu.P.3296) in Point [28]

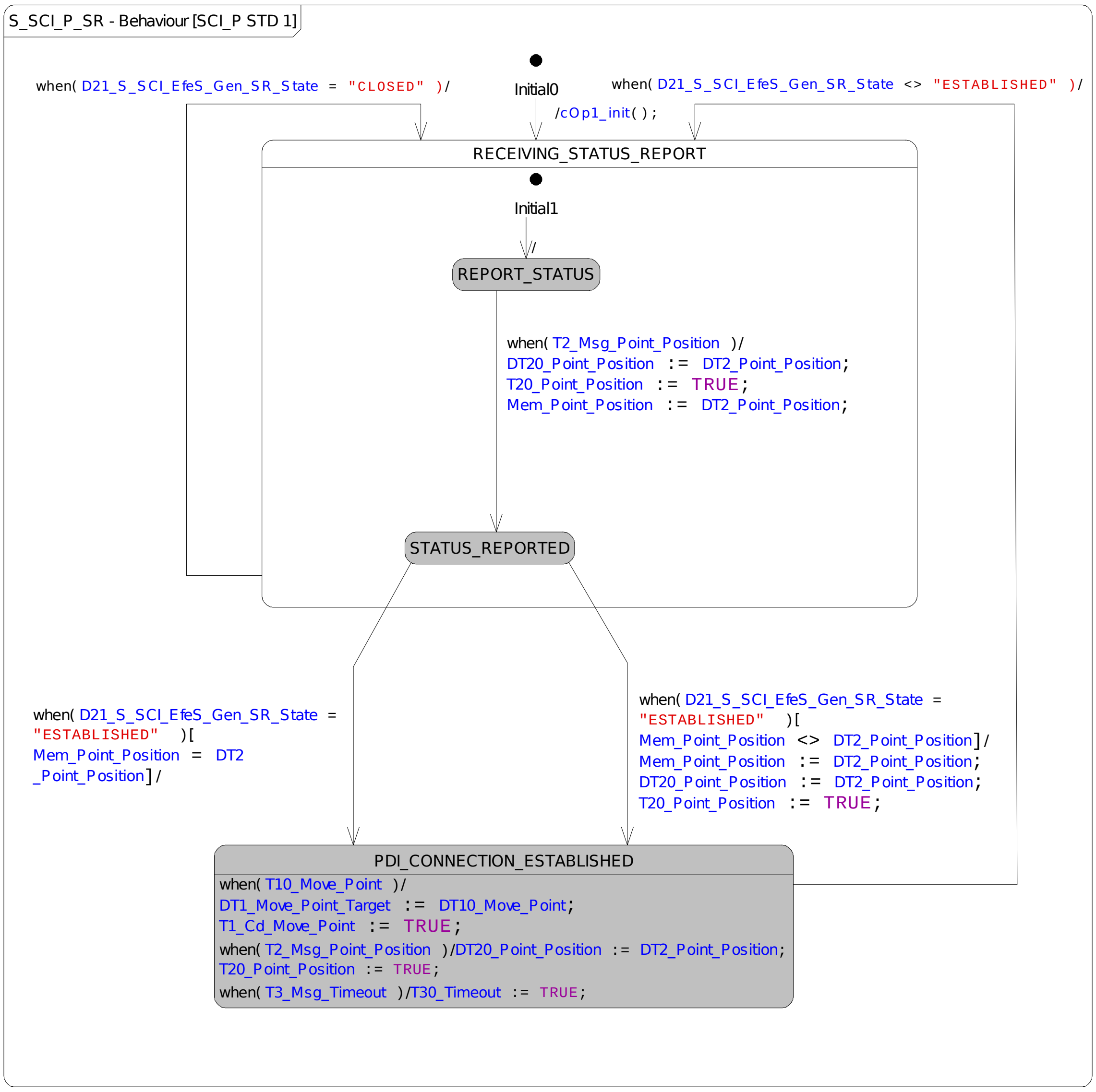

S_SCI_P_SR - Behaviour [SCI_P STD 1]
Initial0
/cOp1_init( );
when(D21_S_SCI_EfeS_Gen_SR_State = "CLOSED")/
when(D21_S_SCI_EfeS_Gen_SR_State <> "ESTABLISHED")/
RECEIVING_STATUS_REPORT
Initial1
/
REPORT_STATUS
when(T2_Msg_Point_Position)/
DT20_Point_Position := DT2_Point_Position;
T20_Point_Position := TRUE;
Mem_Point_Position := DT2_Point_Position;
STATUS_REPORTED
when(D21_S_SCI_EfeS_Gen_SR_State = "ESTABLISHED")[
Mem_Point_Position = DT2_Point_Position]/
when(D21_S_SCI_EfeS_Gen_SR_State = "ESTABLISHED")[
Mem_Point_Position <> DT2_Point_Position]/
Mem_Point_Position := DT2_Point_Position;
DT20_Point_Position := DT2_Point_Position;
T20_Point_Position := TRUE;
PDI_CONNECTION_ESTABLISHED
when(T10_Move_Point)/
DT1_Move_Point_Target := DT10_Move_Point;
T1_Cd_Move_Point := TRUE;
when(T2_Msg_Point_Position)/DT20_Point_Position := DT2_Point_Position;
T20_Point_Position := TRUE;
when(T3_Msg_Timeout)/T30_Timeout := TRUE;

Figure 12.4: State machine diagram of the SP component (Eu.P.4729) in Point [28]

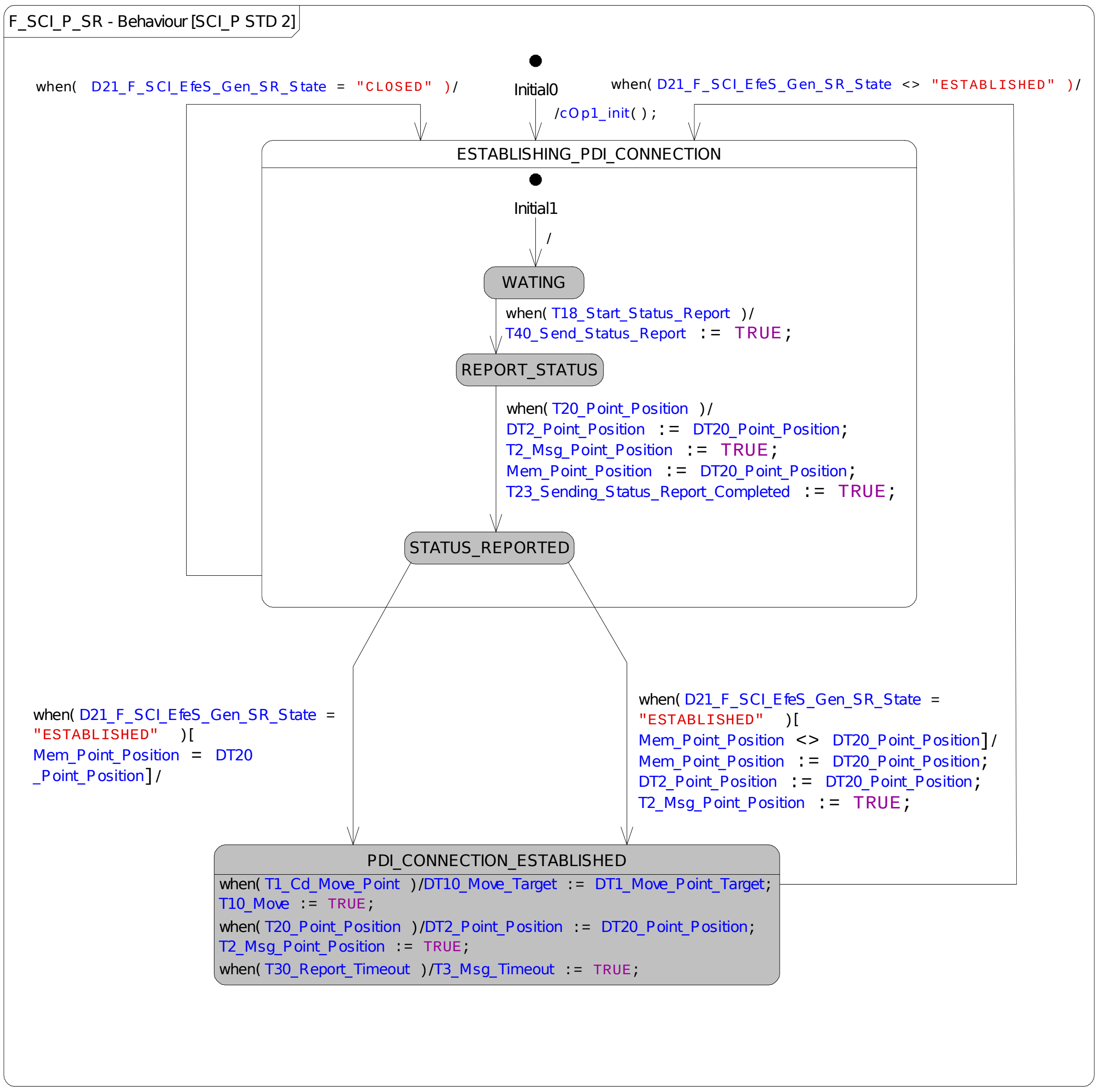

F_SCI_P_SR - Behaviour [SCI_P STD 2]
when(D21_F_SCI_EfeS_Gen_SR_State = "CLOSED")/
Initial0
/cOp1_init();
when(D21_F_SCI_EfeS_Gen_SR_State <> "ESTABLISHED")/
ESTABLISHING_PDI_CONNECTION
Initial1
/
WATING
when(T18_Start_Status_Report)/
T40_Send_Status_Report := TRUE;
REPORT_STATUS
when(T20_Point_Position)/
DT2_Point_Position := DT20_Point_Position;
T2_Msg_Point_Position := TRUE;
Mem_Point_Position := DT20_Point_Position;
T23_Sending_Status_Report_Completed := TRUE;
STATUS_REPORTED
when(D21_F_SCI_EfeS_Gen_SR_State = "ESTABLISHED")[Mem_Point_Position = DT20_Point_Position]/
when(D21_F_SCI_EfeS_Gen_SR_State = "ESTABLISHED")[Mem_Point_Position <> DT20_Point_Position]/
Mem_Point_Position := DT20_Point_Position;
DT2_Point_Position := DT20_Point_Position;
T2_Msg_Point_Position := TRUE;
PDI_CONNECTION_ESTABLISHED
when(T1_Cd_Move_Point)/DT10_Move_Target := DT1_Move_Point_Target;
T10_Move := TRUE;
when(T20_Point_Position)/DT2_Point_Position := DT20_Point_Position;
T2_Msg_Point_Position := TRUE;
when(T30_Report_Timeout)/T3_Msg_Timeout := TRUE;

Figure 12.5: State machine diagram of the FP component (Eu.P.4693) in Point [28]

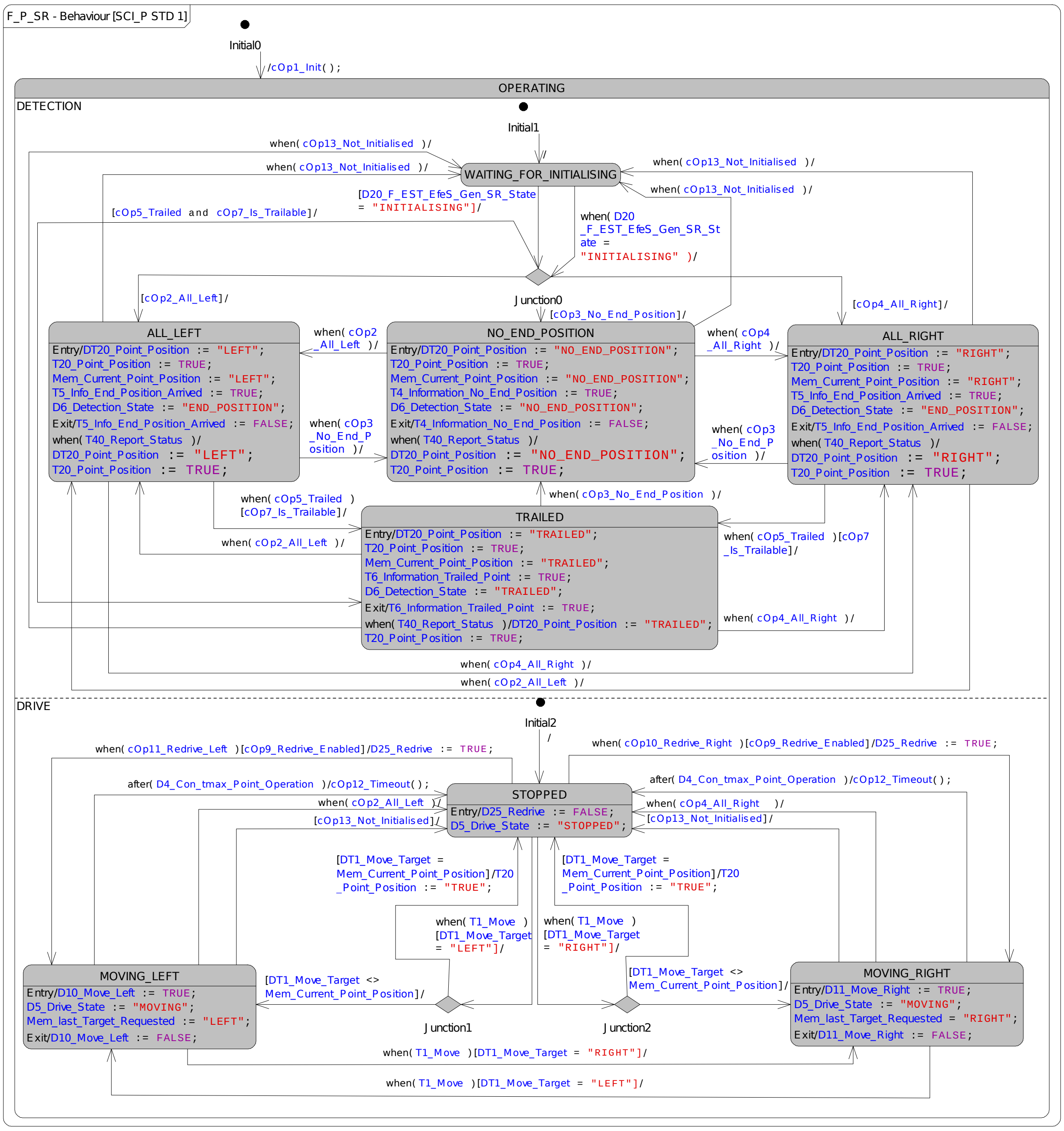

F_P_SR - Behaviour [SCI_P STD 1]
Initial0
/cOp1_Init( );
OPERATING
DETECTION
Initial1
WAITING_FOR_INITIALISING
when(cOp13_Not_Initialised)/
when(cOp13_Not_Initialised)/
when(cOp13_Not_Initialised)/
when(cOp13_Not_Initialised)/
[D20_F_EST_EfeS_Gen_SR_State = "INITIALISING"]/
when(D20_F_EST_EfeS_Gen_SR_State = "INITIALISING")/
[cOp5_Trailed and cOp7_Is_Trailable]/
[cOp2_All_Left]/
Junction0
[cOp3_No_End_Position]/
[cOp4_All_Right]/
ALL_LEFT
Entry/DT20_Point_Position := "LEFT";
T20_Point_Position := TRUE;
Mem_Current_Point_Position := "LEFT";
T5_Info_End_Position_Arrived := TRUE;
D6_Detection_State := "END_POSITION";
Exit/T5_Info_End_Position_Arrived := FALSE;
when(T40_Report_Status)/
DT20_Point_Position := "LEFT";
T20_Point_Position := TRUE;
when(cOp2_All_Left)/
when(cOp3_No_End_Position)/
NO_END_POSITION
Entry/DT20_Point_Position := "NO_END_POSITION";
T20_Point_Position := TRUE;
Mem_Current_Point_Position := "NO_END_POSITION";
T4_Information_No_End_Position := TRUE;
D6_Detection_State := "NO_END_POSITION";
Exit/T4_Information_No_End_Position := FALSE;
when(T40_Report_Status)/
DT20_Point_Position := "NO_END_POSITION";
T20_Point_Position := TRUE;
when(cOp4_All_Right)/
when(cOp3_No_End_Position)/
ALL_RIGHT
Entry/DT20_Point_Position := "RIGHT";
T20_Point_Position := TRUE;
Mem_Current_Point_Position := "RIGHT";
T5_Info_End_Position_Arrived := TRUE;
D6_Detection_State := "END_POSITION";
Exit/T5_Info_End_Position_Arrived := FALSE;
when(T40_Report_Status)/
DT20_Point_Position := "RIGHT";
T20_Point_Position := TRUE;
when(cOp3_No_End_Position)/
when(cOp5_Trailed) [cOp7_Is_Trailable]/
when(cOp2_All_Left)/
TRAILED
Entry/DT20_Point_Position := "TRAILED";
T20_Point_Position := TRUE;
Mem_Current_Point_Position := "TRAILED";
T6_Information_Trailed_Point := TRUE;
D6_Detection_State := "TRAILED";
Exit/T6_Information_Trailed_Point := TRUE;
when(T40_Report_Status)/DT20_Point_Position := "TRAILED";
T20_Point_Position := TRUE;
when(cOp5_Trailed)[cOp7_Is_Trailable]/
when(cOp4_All_Right)/
when(cOp4_All_Right)/
when(cOp2_All_Left)/
DRIVE
Initial2
/
when(cOp11_Redrive_Left)[cOp9_Redrive_Enabled]/D25_Redrive := TRUE;
when(cOp10_Redrive_Right)[cOp9_Redrive_Enabled]/D25_Redrive := TRUE;
after(D4_Con_tmax_Point_Operation)/cOp12_Timeout( );
after(D4_Con_tmax_Point_Operation)/cOp12_Timeout( );
STOPPED
Entry/D25_Redrive := FALSE;
D5_Drive_State := "STOPPED";
when(cOp2_All_Left)/
[cOp13_Not_Initialised]/
when(cOp4_All_Right)/
[cOp13_Not_Initialised]/
[DT1_Move_Target = Mem_Current_Point_Position]/T20_Point_Position := "TRUE";
[DT1_Move_Target = Mem_Current_Point_Position]/T20_Point_Position := "TRUE";
when(T1_Move) [DT1_Move_Target = "LEFT"]/
when(T1_Move) [DT1_Move_Target = "RIGHT"]/
MOVING_LEFT
Entry/D10_Move_Left := TRUE;
D5_Drive_State := "MOVING";
Mem_last_Target_Requested := "LEFT";
Exit/D10_Move_Left := FALSE;
[DT1_Move_Target <> Mem_Current_Point_Position]/
Junction1
Junction2
[DT1_Move_Target <> Mem_Current_Point_Position]/
MOVING_RIGHT
Entry/D11_Move_Right := TRUE;
D5_Drive_State := "MOVING";
Mem_last_Target_Requested = "RIGHT";
Exit/D11_Move_Right := FALSE;
when(T1_Move)[DT1_Move_Target = "RIGHT"]/
when(T1_Move)[DT1_Move_Target = "LEFT"]/

Figure 12.6: State machine diagram of the P3 component (Eu.P.4588) in Point [28]

D Detailed Point requirement descriptions

The the detailed requirement descriptions of the requirements discussed in Section 7.2 are listed in this appendix. As described in Section 5.3.5, the strings are labelled with a unique number ID in the translation. These ID's are clarified in the note of the tables.

ID	R1
Requirement	When SP detects a new position, this position is reported to the EIL.
Clarification	One of the functions of SP is that it should forward new positions received from FP to the Electronic Interlocking system. That a new position is made available by FP can be detected from the pulse port `T2_Msg_Point_Position`. The actual position can be read from the port `DT2_Point_Position`. This value should be written to SP's output port `DT20_Point_Position`. This port is connected to the EIL.
UPPAAL Spec.	`(str_sp.T2_Msg_Point_Position && str_sp.DT2_Point_Position == 2) --> (str_sp.T20_Point_Position && str_sp.DT20_Point_Position == 2)`
Expected res.	Not satisfied. Because there are states in which SP does not listen for a new position report.
Note	The string `"LEFT"` is translated to `2`. This specification only covers forwarding of the message `"LEFT"`. To cover other cases, it needs to be duplicated while changing the string constant ID.

Table 12.1: Detailed description of R1

ID	R2
Sub-model	SP
Requirement	When port D21 of SP reads `"ESTABLISHED"`, SP goes to the `PDI_CONNECTION_ESTABLISHED` state.
Clarification	SP should only forward commands from the EIL to FP when the Prim component (S_SCI_EfeS_Prim_SR) gives the status `"ESTABLISHED"`. SP can read this on its input port D21_S_SCI_EfeS_Gen_SR_State. Once this is the case, SP should move to the `PDI_CONNECTION_ESTABLISHED`. This is the only state in which SP can forward commands from the EIL to FP. Therefore we specify in the requirement that SP should go to this state if port D21 reads `"ESTABLISHED"`.

UPPAAL Spec.	`(str_sp.D21_S_SCI_EfeS_Gen_SR_State == 4) -->`
	`    sp.PDI_CONNECTION_ESTABLISHED_Simple`

Expected res.	Not satisfied. Because there are states in which SP does not listen for a change to `"ESTABLISHED"` in D21.
Note	The string `"ESTABLISHED"` is translated to `4`.

Table 12.2: Detailed description of R2

ID	R3
Sub-model	SP
Requirement	When port D21 of SP reads something other than `"ESTABLISHED"`, SP transitions to the `RECEIVING_STATUS_REPORT` state.
Clarification	The reasoning behind this requirement is similar to the one shown in Table 12.2. The only difference is that SP should move to the `RECEIVING_STATUS_REPORT` state, because this is the state where it cannot forward a command from the EIL. It should first reestablish the connection.

UPPAAL Spec.	`(str_sp.D21_S_SCI_EfeS_Gen_SR_State != 4) -->`
	`    sp.RECEIVING_STATUS_REPORT_Composite`

Expected res.	Satisfied.
Note	The string `"ESTABLISHED"` is translated to `4`.

Table 12.3: Detailed description of R3

ID	R4
Sub-model	SP
Requirement	When port D21 of SP reads `"ESTABLISHED"`, SP can forward a move command.
Clarification	This is in theory the same requirement as described in Table 12.2, but instead of referencing a state name we specifically express that it should be able to forward a command.

UPPAAL Spec.	`(str_sp.D21_S_SCI_EfeS_Gen_SR_State == 4 && str_sp.T10_Move_Point && str_sp.DT10_Move_Point == 2) --> (str_sp.T1_Cd_Move_Point && str_sp.DT1_Move_Point_Target == 2)`
	`(str_sp.D21_S_SCI_EfeS_Gen_SR_State == 4 && str_sp.T10_Move_Point && str_sp.DT10_Move_Point == 5) --> (str_sp.T1_Cd_Move_Point && str_sp.DT1_Move_Point_Target == 5)`

Expected res.	Not satisfied. Because there are states in which SP does not listen for a change to `"ESTABLISHED"` in D21.
Note	The string `"ESTABLISHED"` is translated to `4`, `"LEFT"` to `2`, and `"RIGHT"` to `5`. Two UPPAAL requirements are created to cover both a left and right command.

Table 12.4: Detailed description of R4

ID	R5
Sub-model	SP
Requirement	SP should only send a new position command to FP if that command was given by the EIL.
Clarification	This is a safety property specifying that SP should not send a command to FP when the EIL has not sent the signal for a new command. SP uses pulse port T1_Cd_Move_Point to signal FP that it can read a new move command on its port connected to DT1_Move_Point_Target. On port T10_Move_Point SP listens for a new command given by the EIL. So, we say that SP should not send a pulse to T1_Cd_Move_Point if it did not receive a pulse on T10_Move_Point.
UPPAAL Spec.	Not possible
Note	In the requirement language it is not possible to look back once an event has occurred. A different way to specify that this instance cannot occur would be with the following CTL property: `!E (!str_sp.T10_Move_Point U str_sp.T1_Cd_Move_Point)` However the UPPAAL requirement language is too limited to specify this.

Table 12.5: Detailed description of R5

ID	R6
Sub-model	SP+FP
Requirement	When FP detects a new position, this is forwarded to the EIL.
Clarification	This requirement is similar to R1, but now we take both SP and FP into account. When FP receives a new position from P3, FP forwards it to SP, and SP forwards it to the EIL. We can monitor incoming position reports on port T20_Point_Position and DT20_Point_Position of FP. The position reports sent to the EIL are detected by monitoring port T20_Point_Position and DT20_Point_Position of SP.
UPPAAL Spec.	`(str_fp.T20_Point_Position && str_fp.DT20_Point_Position == 2) --> (str_sp.T20_Point_Position && str_sp.DT20_Point_Position == 2)`
Expected res.	Not satisfied. Because there are states in which SP or FP do not listen for a new position report.
Note	The string `"LEFT"` is translated to `2`. This specification needs to be repeated for each Point position to cover the whole requirement.

Table 12.6: Detailed description of R6

ID	R7
Sub-model	SP+FP
Requirement	When a new position is commanded by the EIL, this position is forwarded to P3
Clarification	This one is similar to R5, but here we take SP and FP into account. When SP receives a command from the EIL, it should forward it to FP, which in turn should forward it to P3. SP receives commands from the EIL on port T10_Move_Point and DT10_Move_Point. FP forwards a command to P3 by publishing it on port T10_Move and DT10_Move_Target.
UPPAAL Spec.	`(str_sp.T10_Move_Point && str_sp.DT10_Move_Point == 2) --> (str_fp.T10_Move && str_fp.DT10_Move_Target == 2)`
Expected res.	Not satisfied. Because there are states in which they do not listen for a move command.
Note	The string `"LEFT"` is translated to `2`, and `"RIGHT"` to `6`. This specification should be repeated for the 'right' command to cover the whole requirement.

Table 12.7: Detailed description of R7

ID	R8
Sub-model	P3
Requirement	If the Point machine status is `"LEFT"` and P3 receives a command to go right, it is possible that P3 will instruct the Point machine to go right.
Clarification	P3 can read the current status of the Point machine (or Point machines, if multiple are supported) from port D21_PM1_Position, D22_PM2_Position D13_PM2_Activation. P3 has an ASAL operation called cOp2_All_Left which takes care of reading out the ports and the logic of dealing with possibly multiple Point machines. The command to move sent by FP to P3 is received on port T1_Move and DT1_Move_Target. To send a command to go right, P3 sets writes true to its port D11_Move_Right. Combining this information, we can check that if cOp2_All_Left evaluates to true, and T1_Move and DT1_Move_Target signal a move command to the right, eventually D11_Move_Right should be true.
UPPAAL Spec.	`A[] ((p3___cOp2_All_Left() && str_p3.T1_Move && str_p3.DT1_Move_Target == 9) imply E<> str_p3.D11_Move_Right)`
Note	The string `"RIGHT"` is translated to `9`. In theory this specification covers the requirement, however UPPAAL does not support nested path expressions, so it will not run.

Table 12.8: Detailed description of R8

ID	R9
Sub-model	P3
Requirement	If the Point machine status is `"LEFT"` and P3 receives a command to go right, P3 will instruct the Point machine to go right if no problems occur.
Clarification	This requirement extends R8 with the exclusion of problematic behaviour. When a train drives through the junction in a certain way it can move the junction to the other side. This state is called trailed. It could be that the junction is already moved to the correct position. In that case P3 should not send a move command to the Point machine. We can check if the Point machine is trailed with the operations cOp5_Trailed and cOp7_Is_Trailable.
UPPAAL Spec.	Not possible
Note	In the UPPAAL requirement language it is not possible to specify that ‘this’ results in ‘that’ given a certain invariant. Due to the nature of the CTL language it is not even possible to specify this requirement in that language.

Table 12.9: Detailed description of R9

ID	R10
Sub-model	P3
Requirement	If redrive is enabled, the Point machine reads `"LEFT"`, and is no longer `"LEFT"`, P3 should instruct to go left.
Clarification	With this requirement we specify that P3 should correct the Point machine if it moves to the wrong side without a command. This could happen when the Point machine is trailed. So if previously cOp2_All_Left was true, but now cOp2_All_Left is false while cOp9_Redrive_Enabled is true and T1_Move has not changed, the port D10_Move_Left should be set to true.
UPPAAL Spec.	`(p3___cOp9_Redrive_Enabled && str_p3.Mem_Current_Point_Position == 3 && !p3___cOp4_All_Left) --> str_p3.D10_Move_Left`
Expected res.	Satisfied.
Note	The string `"LEFT"` is translated to `3`. As it is not possible to specify that event 'A' followed by event 'B' should always result in event 'C', we use one of the memory properties/variables of the P3 component. In this way we can detect that the Point machine is 'no longer left'.

Table 12.10: Detailed description of R10

ID	R11
Sub-model	P3
Requirement	The DT20 port of P3 can only have the values `"LEFT"`, `"RIGHT"`, `"NO_END_POSITION"`, `"TRAILED"`, or `""` (uninitialised).
Clarification	This is a simple safety property that the port DT20_Point_Position can only have the specified values.
UPPAAL Spec.	`A[] str_p3.DT20_Point_Position == 3 || str_p3.DT20_Point_Position == 9 || str_p3.DT20_Point_Position == 0 || str_p3.DT20_Point_Position == 8 || str_p3.DT20_Point_Position == 2`
Expected res.	Satisfied.
Note	The string `"LEFT"` is translated to `3`, `"RIGHT"` to `9`, `"NO_END_POSITION"` to `0`, and `"TRAILED"` to `8`.

Table 12.11: Detailed description of R11

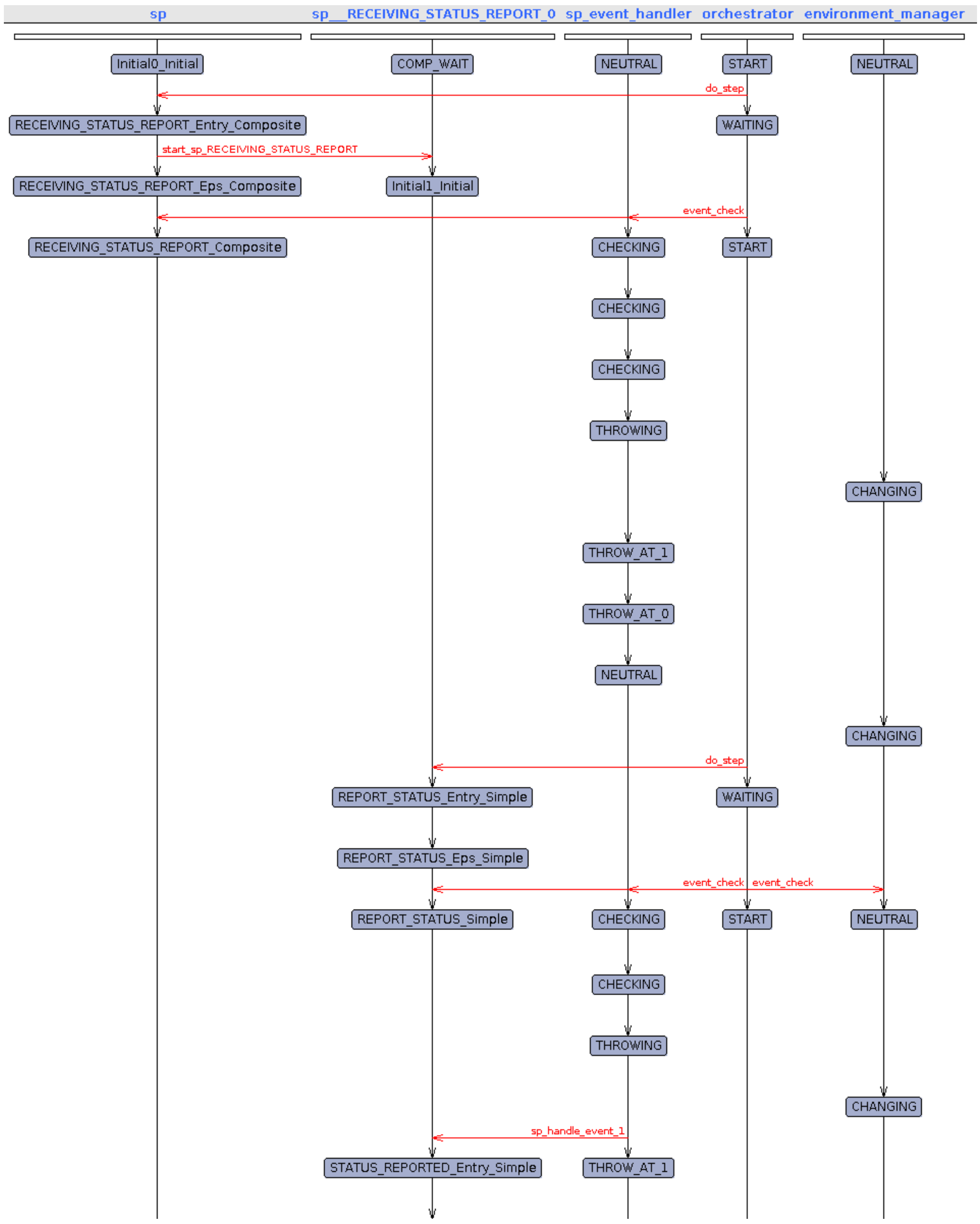
ID	R12
Sub-model	P3
Requirement	When DT20 reads `"LEFT"` or `"RIGHT"`, D6 should read `"END_POSITION"` and vise versa.
Clarification	This is another simple safety property. It specifies a correlation of two output ports of P3. When port DT20_Point_Position is set to `"LEFT"` or `"RIGHT"`, the port D6_Detection_State should be set to `"END_POSITION"`, and the other way around. This is an invariant that should always be true.
UPPAAL Spec.	`A[] (str_p3.DT20_Point_Position == 3 || str_p3.DT20_Point_Position == 9) == (str_p3.D6_Detection_State == 13)`
Expected res.	Satisfied.
Note	The string `"LEFT"` is translated to `3`, `"RIGHT"` to `9`, and `"END_POSITION"` to `13`.

Table 12.12: Detailed description of R12

E Example simulation trace of SP in UPPAAL

The diagnostic trace shown in Figure 12.7 is the result of verifying the following requirement in UPPAAL (with the ‘shortest trace’ option selected):

```
E<> (sp.PDI_CONNECTION_ESTABLISHED_Simple &&
sp_event_queue_peek() == 3)
```



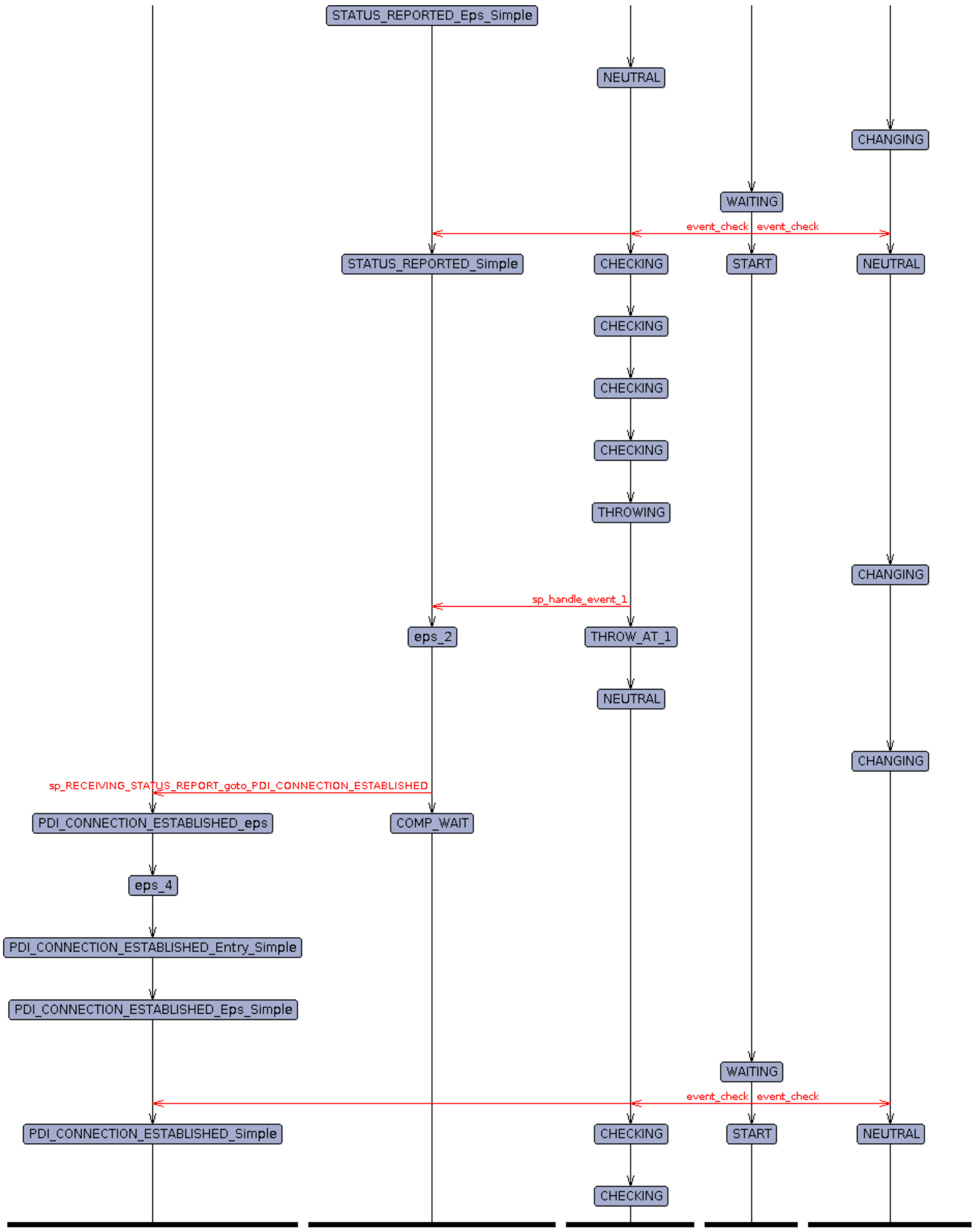


Figure 12.7: Example simulation trace of SP in UPPAAL