TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Mariam Mikava 195355IVSM

Conversion from Stateflow to Uppaal Model

Master's thesis

Supervisor: Tonu Naks
MSc

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Mariam Mikava 195355IVSM

Stateflow mudelite teisendamine Uppaal'i mudeliteks

Magistritöö

Juhendaja: Tonu Naks
MSc

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mariam Mikava

10.05.2021

Abstract

Model Driven Engineering (MDE) is widely used in software development, in particular for development of safety-critical systems. Simulink and Stateflow are often used for developing control systems. While providing excellent, easy to use tools for composing the models and for simulation, the formal analysis capabilities are limited both in Simulink and Stateflow.

This thesis, investigates the possibility of formal verification of Stateflow models by converting them into the Uppaal model checking tool. Firstly, we do an experiment for demonstrating the advantages of the Uppaal verification language. Then, we create a Uppaal EMF metamodel as an intermediate layer for manipulating Uppaal elements during the translation and, finally, map Uppaal elements to the Stateflow elements. The thesis also provides the translation of different Stateflow modelling patterns and investigates how to represent fully deterministic Stateflow semantics in essentially non-deterministic of Uppaal models. The approach is verified by two example models.

This thesis is written in English and is 66 pages long, including 6 chapters, 42 figures and 6 tables

List of abbreviations and terms

Table of contents

List of figures

List of tables

1 Introduction

Modelling real-time safety-critical systems have a vital role in developing high quality, safe and liveness systems with minimal cost. Model-Driven Engineering (MDE) is widely used to create safety-critical designs, for example, in avionics, air traffic management, autonomous vehicle, etc. MDE is particularly useful because it supports the designing, validation and verification of safety-critical systems [1]. Simulink and Stateflow graphical tools are very popular MDE tools for modelling critical systems [2]. Simulink for verification uses Simulink Design Verifier and Simulink Polyspace [3]. They are responsible for detecting design and implementation errors. As a result, they can successfully reveal blocks in the model that result in integer overflow, dead logic, array access violations, and division by zero. Simulink design verifier can formally verify that the design meets functional requirements. However, it is not able to deal with real-time complex temporal properties [4][5]. That is why developing new verification technologies for the Simulink/Stateflow models has a vital role in maintaining a system safe, liveness and reduce the production cost using early-stage verification.

The popularity of MDE is increasing, and therefore the systems modelled using MDE tools are getting more complex. For instance, the Stateflow graphical modelling language semantics is given by 886 pages long document [6][7]. Stateflow and Simulink are often used in the modelling of safety-critical systems where potential failure can have fatal results. That's why the needs for better verification methods for Simulink/Stateflow models are increasing. Formal methods are preferred given that unlike simulation-based verification, they provide guarantees for the correctness of checked properties [8].

Timed-automata is a powerful concept for modelling and verifying real-time systems, providing a good basis for formal verification. Timed automata is an automation containing a finite set of nodes and is extended with real-valued variables. One of the modelling and verification tools providing model checking capabilities based on the theory of time automata is Uppaal [9].

Uppaal is a model checking tool used for modelling, validation and verification of real-time systems. The tool contains two main parts a graphical user interface (GUI) and a model checker engine. The GUI of the Uppaal version used in this thesis (4.1.24) provides the support of the following elements [10][11]:

- Editor – Construction of the model
- Simulator – Provides both guided and random simulation with displayed trace and updated values of global and local variables
- Concrete simulator – Each step of the simulation is better shown at a specific time unit
- Verifier - The main power of the tool provides the possibility to verify essential properties of the model and can detect and generate counter example which significantly simplifies the error identification process

. This thesis is seeking for possibilities of verifying time-based requirements. In Stateflow such verification can be done by simulation which has the same limitations as software testing. The verifier has to find scenarios that may lead to requirement violation and compose test/simulation according to this scenario. In the Uppaal model-checking tool, the verification of temporal properties is naturally available. The Uppaal query language simplifies describing the safety and liveness properties of the model and verifying them using model-checker. This thesis assumes that automatic conversion from Stateflow models into a Uppaal model checking tool will make the timing analysis available for Simulink users.

1.1 Research goal

The main goal of this thesis is to create a converter that automatically translates Stateflow models into the Uppaal model checking tool and gives the possibility for property verification using the Uppaal query language. For achieving the primary goal, it is essential to address the following concrete goals:

- Analyze the usefulness of translation from Stateflow to Uppaal models by running the experiments to compare the verification capabilities of both modelling tools.

- Create a Uppaal EMF metamodel representing element types of the Uppaal model [2]. The metamodel will be used for creating an intermediate layer for manipulating the model elements before recording them in the file for the Uppaal model checking tool. The approach makes it simpler to extract data from Uppaal and map it to Stateflow's API, respectively.

- Develop a mapping between selected subset of Stateflow elements and corresponding Uppaal model constructs.

- Implement an automatic translation from Stateflow to Uppaal model checking tool by developing an algorithm for the mapping.

- Verify the mapping rules on selected example models. Full verification of all the mapping element remains outside of the scope of this thesis.

1.2 The verification language of Uppaal

The main reason for using a model-checking is to verify whether a finite-state model of a system meets a given specification [12]. The language for defining specifications should be descriptive enough to express the requirements sufficiently. Uppaal uses the query language based on computational tree logic (CTL) [13] and uses mathematical notations to provide path evaluations using specific symbols. In Uppaal we have statistical, state, and temporal properties. Based on statistical properties UPPAAL can estimate the probability of statistical expression values. There are four types of statistical properties: quantitative, qualitative, comparison and probable value [14]. This work is more concerned about the other too property groups: temporal and state properties. When p and q are state properties, then there are five temporal properties that can be applied to them [15]:

1. E<>p - Possibly : there exists a path where eventually p holds.

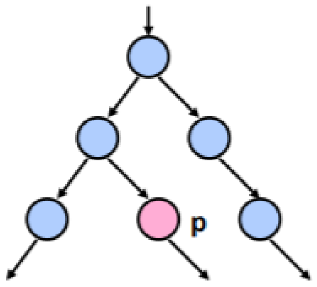


Figure 1. Possibly

2. E[]p - Potentially always: there exists a path where p always hold

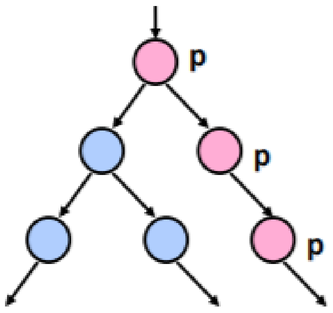


Figure 2. Potentially

3. A<>p - Eventually: for all path p eventually holds

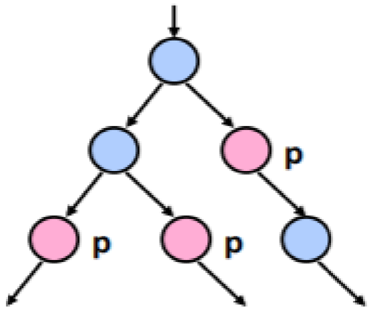


Figure 3. Eventually

4. A[]p - Invariantly: for all path p always holds

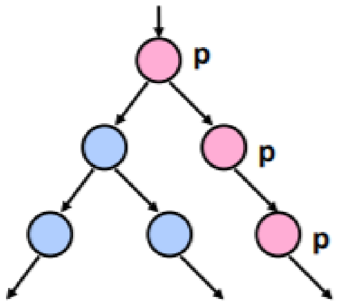


Figure 4. Invariantly

1. p-->q - Leads to: for all path if p becomes true, also q will eventually become true

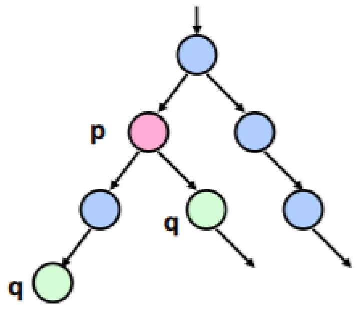


Figure 5. Leads to

Using these five notations, it is possible to easily verify state and temporal properties that are essential while building real-time safety-critical systems. The following three statements given below are the combination of state and temporal properties which are represented using the notations above.

- Safety properties are the most important property for proving that the system never will be in a state where damage can happen. One of the most substantial property that Uppaal verifier can check is if the system is deadlock-free or not. In avionics, traffic control in aerospace, autonomous vehicle production etc., it is critically important to verify that the systems are deadlock-free.

- Liveness properties are used to prove that the system eventually reaches the desired state. An example of using liveness properties could be the elevator that is supposed to arrive eventually after a button press.

- Reachability properties are used to verify the behaviours of the model. Uppaal for communication uses the channel synchronisation between sender and receiver. Moreover, using reachability properties, it is easy to check how synchronisation is happening.

1.3 Motivation:

The verification is an essential part while creating real-time safety-critical systems. Non-verified specific property can cause delayed testing, production, and in some cases, it could be a reason for developing an unsafe system. Stateflow is one of the languages that is a widely accepted tool for large-scale system development and the possibility of providing formal analysis while allowing the user to stick to the tool thy are used to is the main motivation for this thesis. Stateflow is easy-to use and simulation capabilities within Simulink make it a powerful design tool, however verification through simulation has the same limitations as testing in software development.

1.4 Challenges

There are several important differences between the semantics of Stateflow and UPPAALlanguages which make the translation complicated.. Stateflow semantics is very complex. It uses an event stack mechanism, while Uppaal does not support it. The execution of Stateflow models is entirely deterministic, where each transition has priority and guard represents the execution condition. On the contrary, Uppaal models contain non-determinism, and the guard represents only an enabling condition.

1.5 Research questions

1) What are the benefits using Uppaal verification language over statflow models?
2) How does the semantics of Stateflow elements trantslate to the semantics of UPPAAL model?
3) How to synchronize the execution of translated elements in a way to maintain the behaviour of the model?
4) How to represent deterministic behaviour of Statelflow model in Uppaal semantics, which is essentially probabilistic?

2 Verification frameworks and theoretical background

2.1 Verification Frameworks

The paper [11] is one of the most popular Uppaal introductory tutorial which explores every element of Uppaal and explains the execution characteristics using several models.

As the stability of the embedded software becomes more critical, the software verification tools are gaining more and more attention. The objective of the paper [16] is to compare model checking tools performance with respect to time. Authors focused on four different model checkers for study purpose: NuSMV, SPIN, UPAAL and PES. NuSMV is a model checking tool that uses binary decision diagrams. SPIN model checker uses linear temporal logic (LTL) [17] formulas for software models verification.

The study uses eight different translation schemes from UML [18] activity model checker for comparing above mentioned tool performances. Paper presents algorithms for each type of the translation and graph analysis of the results. Authors focus on how much verification time was spent by each translation, they also present mean and median of verification times for every activity and emphasize offset variations for determining the best tool for UML activity model checker.
Three different SPIN translations were used in the process of research. The study shows that all of three translations presented small difference between median and mean.

Overall process showed that channel based translation is less optimal in comparison with flat and variable-based ones, which have more or less similar performance. Two translations were defined for the model checker UPAAL: a centralized-control and distributed-control translation. It is notable that mean and median of the verification time are more similar in the distributed translation. Graphs show that the verification time is as well affected by the size of the activities. Results of this experiment shows that distributed translation has a better performance than a centralized one.
As for the PES model checker, study showed that mean and median values are very similar, but unlike others the offset value remains constant in spite of the activity size. The verification time for the translation were twice longer that the UPAAL's best time performance.
Two translation schemes were used to evaluate the performance of the NuSMV model checker. Turned out that there is significant difference between flat and modular translations, as flat translation takes a lot less time for action completion. Overall in comparison with other model checkers NuSMV showed lowest performance.
Authors present UPAAL distributed translation compared to SPIN variable based and PES translations. The difference is demonstrated in graph, displaying verification times for three of the abovementioned checkers. In overall performance of the UPAAL is the best compared two others.

2.2 Related Work

Several works are done using the Uppaal model checking tool for formal analysis and verification of safety-critical systems. The main emphasis on most of the papers is transforming Simulink models that are not containing the Stateflow tool. However, few works still address formal verification based on state machines, including Stateflow with a limited number of model elements.

The paper [19] presents the approach of transforming Simulink blocks into UPPAAL Statistical model checker by illustrating two Brake-by-Wire and Adjustable Speed Limiter systems. They translated computational blocks and the other blocks which are defining the structure of the models are eliminated during the transformation. They presented pattern for discrete and continues blocks using three: Start, Offset, and Operate states. Where the start represents the initial starting point, offset is used to model the delays of the system and the operate state is used to display the output of the system and also it represents the last state for the system. With combination of Uppaal property language they also used the temporal logic extension weighted metric to measure the probabilities of property satisfaction. Their work was motivated by the industrial need of developing brake and speed limiter systems with better verification possibilities. The paper covers important part of the Simulink tool and shows the usage of property language. The exact details of the transformation method are not described – and also not expected given that it is a paper rather than a research report. The translation of Stateflow models transformation is not investigated. Compared to this work, in our research we address the Stateflow model transformation and provide the mapping with an implementation for automatic translation.

The paper [20] presents the approach of transforming the Safechart models into extended time automata (ETA). Safechart is one of the variants of Statecharts which is specifically created to maintain risk analysis for safety critical systems [21]. The presented research objectives are to understand:

- How Safecharts can be translated in ETA in a way to maintain the specific safety semantic of Safecharts.
- How safety properties can be chosen to be used for verification of Safechart models.
- How to deal with the safety non-determinism that could be presented in Safecharts but isn’t desirable for safety-critical systems.

To address these objectives, they created input language for defining complex hierarchical models and for decomposition of hierarchical states they developed flattening algorithm. To deal with the non-determinism they developed algorithm for risk level evaluation, and checked if there were any transition with unknown level of rick and implement transition prioritization. The SGM model-checking tool which for verification uses Time computational tree logic (TCTL) was used during the transformation. Similarly to this paper, in the current work, the topic of translation between deterministic and non-deterministic is models raised. However, because of syntactic differences among used tools, the approach of dealing with none determinism in our paper is different from what is presented above.

The verification approach of a real-time train controller design presented in [22] is among the few articles which focus on translation from Simulink/Stateflow to Uppaal. The paper introduces the Uppaal runtime verification, explains importance of it and concentrates on translating six Stateflow elements: states, transitions, junction, actions, timers, and events. They identify two main challenges for their translation:

1) Stateflow transition is driven by events. Execution of every event is in deterministic sequential order, and interruptible with stack. At the same time timed automata is executed in parallel, and driven by the channel synchronization without the support of stack.

2) Stateflow supports hierarchy structure which is combined with recursive activation-deactivation mechanism, transitional action, and conditional action very closely. At the same time timed automata supports a single state.

The paper addresses these complications by:

- Creating a virtual stack
- Implementing the state transformation rule: For a regular simple state without decomposition or attached actions, the transformation is straightforward. They just directly map simple Stateflow states to Uppaal timed automata. But for those complex Stateflow state with decomposition or attached actions, they used parallel cooperative templates.

Overall, this article covers an important part of Stateflow elements. However, there are several missing elements like the different kinds of compositions, function box, temporal logic inside the actions and most importantly, there is no investigation of one of the main characteristic of Uppaal non – determinism [23]. One of the most significant differences between Stateflow and Uppaal models is that the execution of Stateflow models is entirely deterministic. The transitions have priorities, and a guard is considered an execution condition, while in Uppaal, a guard only represents the enabling condition. In figure 6, the change of a state from s1 to s2 is triggered when the guard condition is satisfied and time is equal to 10 units. In figure 7 corresponded Uppaal model is presented. If we neglect none determinism, we will get that when time is equal to 10, it is possible to go from s1 to s2, but it is not a strict execution condition. Furthermore, it gives different verification results rather than what is expected for the Stateflow model. To force the system to go from one state to another, it is needed to eliminate non-determinism from the model.

The example:

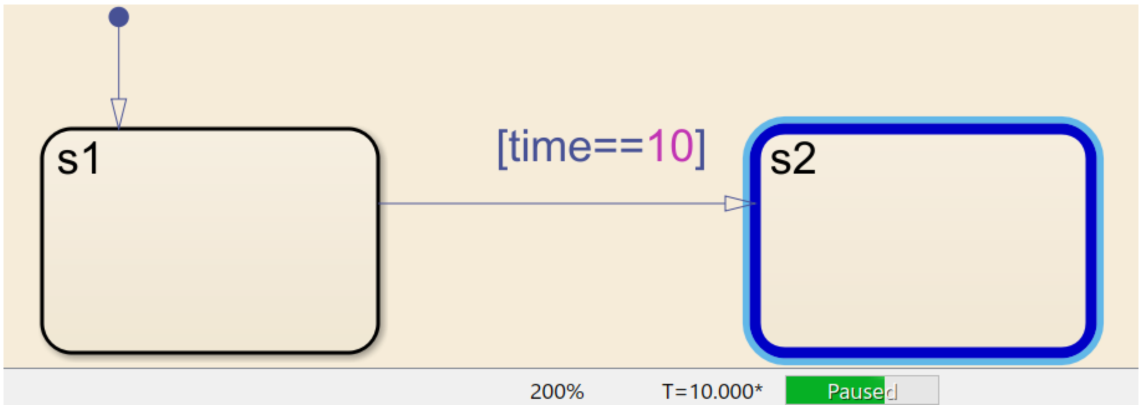


Figure 6. Determistic Stateflow model

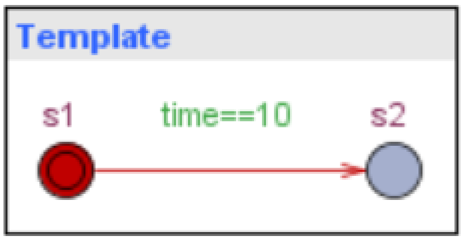
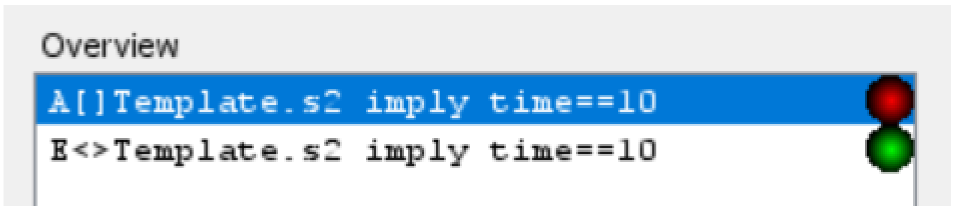


Figure 7. Non-deterministic Uppaal model

The first property is not satisfied because we have non-determinism in the above model (see Figure 7). It is possible for the system to stay in s1 state after 10 time units that's why for all path the first condition is not satisfied (see Figure 8).



```
A[]Template.s2 imply time==10
E<>Template.s2 imply time==10
```

Figure 8. Verification in Uppaal

Compared to this paper, we address the translation of additional Stateflow elements. We go deeper in Stateflow compositions, develop mapping for or, and, and flow graph compositions, explore temporal actions in states and provide a pattern for representing deterministic behaviour in Uppaal.

2.3 Descriptions of the formalisms

2.3.1 Description of the Stateflow language

Stateflow is a modeling language and a tool in the MATLAB toolset for the modeling and simulation of decision logic using state machines [24] [25].

2.3.1.1 State modelling concepts

Concept	**Definition**
Machine	Top-level container of Sateflow elements within one model
Chart	A chart is the top-level object that has an explicit interface (data and events). In Stateflow there exist also Subchart
State	A state describes an operating mode of a reactive system. In a Stateflow chart, states are used for sequential design to create state transition diagrams. States can be active or inactive. The activity or inactivity of a state can change depending on events and conditions. The occurrence of an event drives the execution of the state transition diagram by making states become active or inactive. At any point during execution, active and inactive states exist.
State label	The label for a state appears on the top left corner of the state rectangle with the following general format: State Name entry:entry actions during:during actions exit:exit actions

	on event_name:on event_name actions on message_name:on message_name actions bind:events
Transition	A graphical element in a chart that can be used for connecting The states in char is transition
Transition label	A transition label can consist of an event or message, a condition, a condition action, and a transition action. Each part of the label is optional. The ? character is the default transition label. Transition labels have this overall format: event_or_message[condition]{condition_action}/transition_action
Explicit event	An event is a Stateflow object that can trigger actions in one of these objects and the explicit event is an event that you define explicitly
Implicit event	An implicit event is a built-in event. These events are implicit because there is no need to define them explicitely
Actions	State actions are instructions written inside a state and defines how a chart behaves during simulation
Guard	Guard is a condition added on the transition label. Condition should be satisfied to execute the transition and if the condition is satisfied the transition is executed.
Junction	Connective junctions are decision points in the system.

Table 1 –Stateflow language concept

2.3.1.1.1 Temporal logic concepts

Concept	**Definition**
After	Returns true if at least n units of time have elapsed since the associated state became active. Otherwise, the operator returns false.

	Syntax: after (n,time_unit) n is a positive real number or an expression that evaluates to a positive real value. Time_unit is sec, msec, or usec.
At	Returns true if exactly n seconds have elapsed since the associated state became active. Otherwise, the operator returns false. Syntax: at (n,time_unit) n is a positive real number or an expression that evaluates to a positive real value. Time_unit is sec, msec, or usec.
Before	Returns true if fewer than n units of time have elapsed since the associated state became active. Otherwise, the operator returns false. Syntax: before (n,time_unit) n is a positive real number or an expression that evaluates to a positive real value. Time_unit is sec, msec, or usec.

Table 2 – Stateflow temporal logic concept

2.3.2 Description of the Uppaal modelling language

2.3.2.1 Concepts

Concept	**Definition**
Template	Parallel processes, is used to model small parts of a system
Location	Locations are the states of the system

Initial location	State where the process starts
Urgent location	Time is not allowed to pass when the system is in this state
Committed location	The most restrictive location type. There must be at least one outgoing transition enabled and the transition is taken immediately.
Invariant	Conditions on locations which allows system to stay in the location until condition becomes false
Rate of exponential	Rate of clocks given by the general expression, used for statistical model checking.
Edge	Transition between locations
Selection	Allows to select a value from a range non-deterministically
Guard	Guard is a condition added on the transition. Condition should be satisfied to execute the transition, guard represents only enabling condition
Synchronization	Synchronisation is used for communication between processes, channels are the labels for synchronisation they send or receive signals.
Update	Update allows initializations and update of variables and function calls on the transition
Weight	Automata support branching edges where weights can be added to give a distribution on discrete transitions.
Clock	Clocks are real-valued integers, measured in real time units

Table 3 – Uppaal language concept

3 An experiment to demonstrate the power of Uppaal query language

Before starting the translation from Stateflow to Uppaal model checking tool, important part is to perform the experiments and demonstrate the advantages of verification using a query language. This section demonstrates usage of Uppaal and Stateflow for modelling a periodic traffic-light system.

3.1 Problem statement for the experiment

Model two traffic lights: three-color periodic traffic light for cars and a two-colour on-demand traffic light for pedestrians.

Demonstrate how to model and verify the following requirements:

1. Green light for vehicles should be at least X ticks

2. Vehicle should get a green light at least every Y ticks

3.Pedestrian shall get green light no later than Z ticks after the button press

4.The period for switching the lights is P (unless some other condition interferes, a colour stays on for P time units and is then switched)

It is assumed that X <= P.

3.1.1 The model of non-periodic traffic light systems.

Scenarios to cover in the model:

- Vehicle green
    - Pedestrian pressed the button, time passed from switching green on is less than X → the system waits until X and then switches the light
    - Pedestrian pressed the button, time passed from switching green on is equal or larger than X → the system switches to yellow immediately

    - Pedestrian pressed the button, time passed from switching green on is less than X, the button was already pressed before → the system waits until X and then switches the light
    - Time X passed, the pedestrian did not press the button → system stays on green, no change
    - Time P passed, the pedestrian did not press the button → system stays on green, no change
- Vehicle yellow
    - Pedestrian pressed the button → no change, the system is already waiting for red
    - Time P passed → system switches to red
- Vehicle red
    - Pedestrian pressed the button → no change, the system is already at red
    - Time P passed → system switches to green

Dataset:

X - minimum time for vehicle green: 20 time units

Y - minimum period between two vehicle greens: 20 time units

Z - maximum delay between button press and pedestrian green: 30 time units

P - the period of each light to stay on: Min green for vehicle - 20 time units, Yellow for vehicle -10 time units, Red for vehicle -10 time units, Min red for pedestrian 30 time units.

3.2 Models

3.2.1 Modelling with Uppaal

Main logic:

The presented non-periodic traffic light system's state change is triggered by environment input when the pedestrian press the button. Depending of the time intervals between pressing there are several different execution paths (see Figure 9):

1. The system gets the input from the environment of button pressed and the vehicle light is on green state and the time of being the green state for vehicle is less or equal to X (20 time units) . The system waits on green state, until the time of green state for vehicle becomes less or equal to 20 and then switches light to yellow.

2. The system gets the input from the environment of button pressed and the vehicle light is on green state and the time of being the green state for vehicle is greater than X (20 time units) the system immediately switches to the yellow state.

3. The system is in the yellow state and the pedestrian pressed the button. When the system is in the yellow state it won't get any massage from the environment and the state stays on green state. After P (10 time units) the system switches to the red state.

4. The vehicle light is in the red state and the pedestrian pressed the button. When the system for vehicle light is in the red state it won't get any massage from the environment and after P (10 time units) the system switches to the green state for vehicle.

5. The vehicle traffic light is on the green state and there is no input from the environment. The vehicle light stays on the green state.

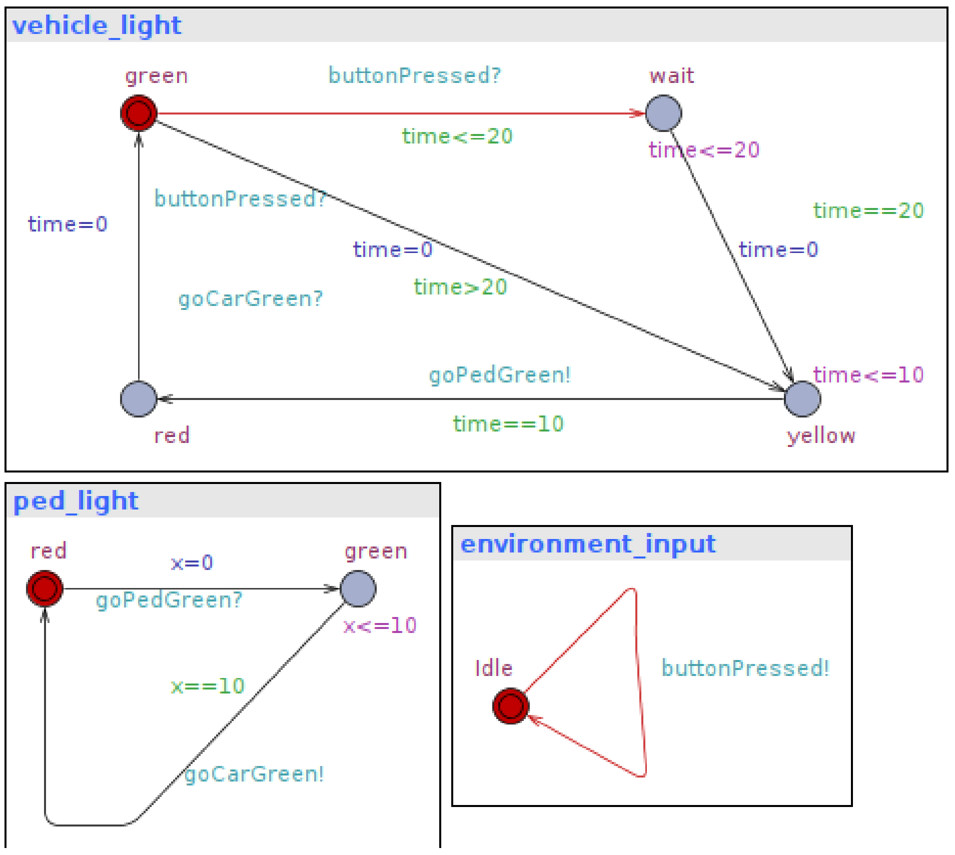


Figure 9. Non-periodic traffic light system model in Uppaal

3.2.1.1 Verification conditions

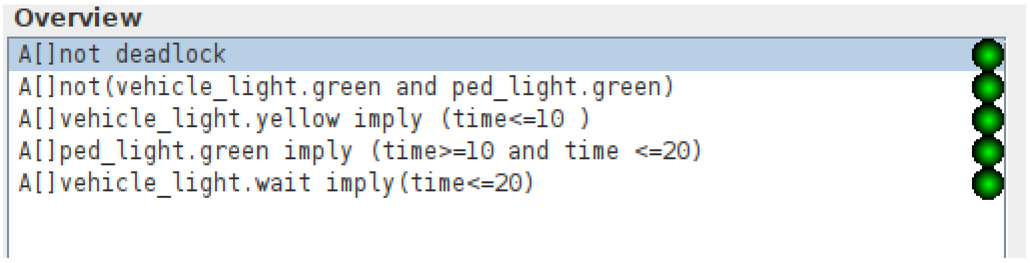
The idea of this experiment is to show that the verification of safety and temporal properties using formal methods and specifically using Uppaal query language is efficient way (see Figure 10).

1. A[]not deadlock: there is no deadlock in the model.
2. A[] (not (vehicle_light.green and ped_light.green)) : There is never green for pedestrians and cars at the same time.
3. A[]ped_light.green imply (time>=10 and time <=20) : Green for pedestrian will stay during 10 time units.
4. A[]vehicle_light.wait imply (time<=20) : The green light for the vehicles stays on at least during 20 time units. If the pedestrian press in less than 20 time unit

interval the system goes to waiting state until 20 time units passes and then it goes from green to yellow state.

5. E<>vehicle_light.wait and time>=20: There is no path where after 20 time units, from a pedestrian pressed the button, still will be the green state for vehicles.

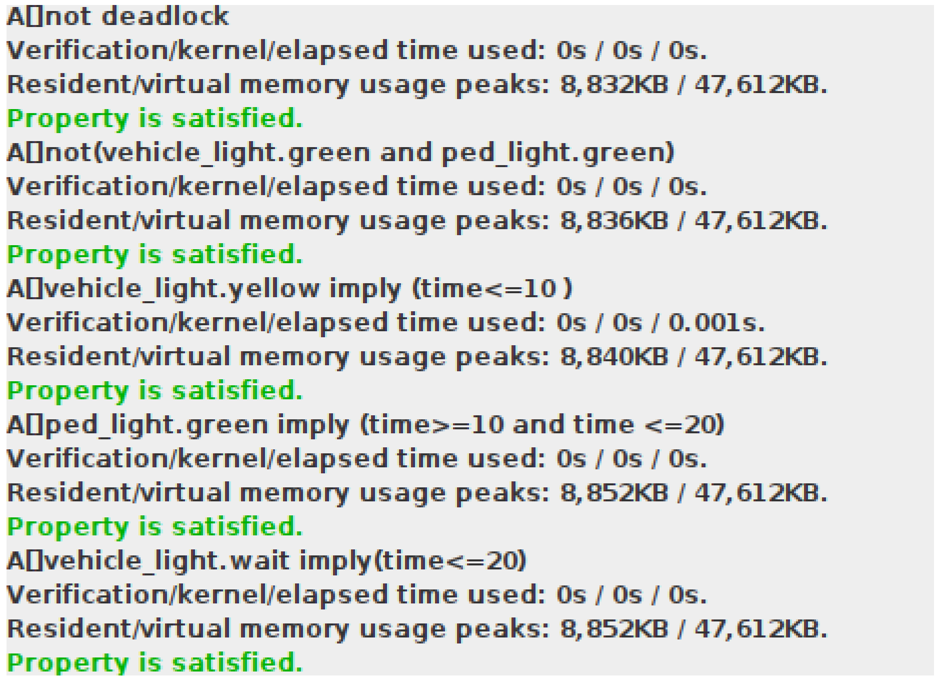
6. A[]vehicle_light.yellow imply (time<=10) : This condition with outgoing guard time==10 states that for all path after pedestrian press there will be yellow light exactly 10 time units.



```
Overview
A[]not deadlock                                              ●
A[]not(vehicle_light.green and ped_light.green)              ●
A[]vehicle_light.yellow imply (time<=10 )                    ●
A[]ped_light.green imply (time>=10 and time <=20)            ●
A[]vehicle_light.wait imply(time<=20)                        ●
```

Figure 10. Verification in Uppaal

Here is the log of the verification performance of Uppaal model checking. For example, to verify that the whole system is deadlock free it took less than one millisecond (see Figure 11) .



```
A[]not deadlock
Verification/kernel/elapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 8,832KB / 47,612KB.
Property is satisfied.
A[]not(vehicle_light.green and ped_light.green)
Verification/kernel/elapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 8,836KB / 47,612KB.
Property is satisfied.
A[]vehicle_light.yellow imply (time<=10 )
Verification/kernel/elapsed time used: 0s / 0s / 0.001s.
Resident/virtual memory usage peaks: 8,840KB / 47,612KB.
Property is satisfied.
A[]ped_light.green imply (time>=10 and time <=20)
Verification/kernel/elapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 8,852KB / 47,612KB.
Property is satisfied.
A[]vehicle_light.wait imply(time<=20)
Verification/kernel/elapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 8,852KB / 47,612KB.
Property is satisfied.
```

Figure 11. Performance of verification in Uppaal

3.2.2 Stateflow model

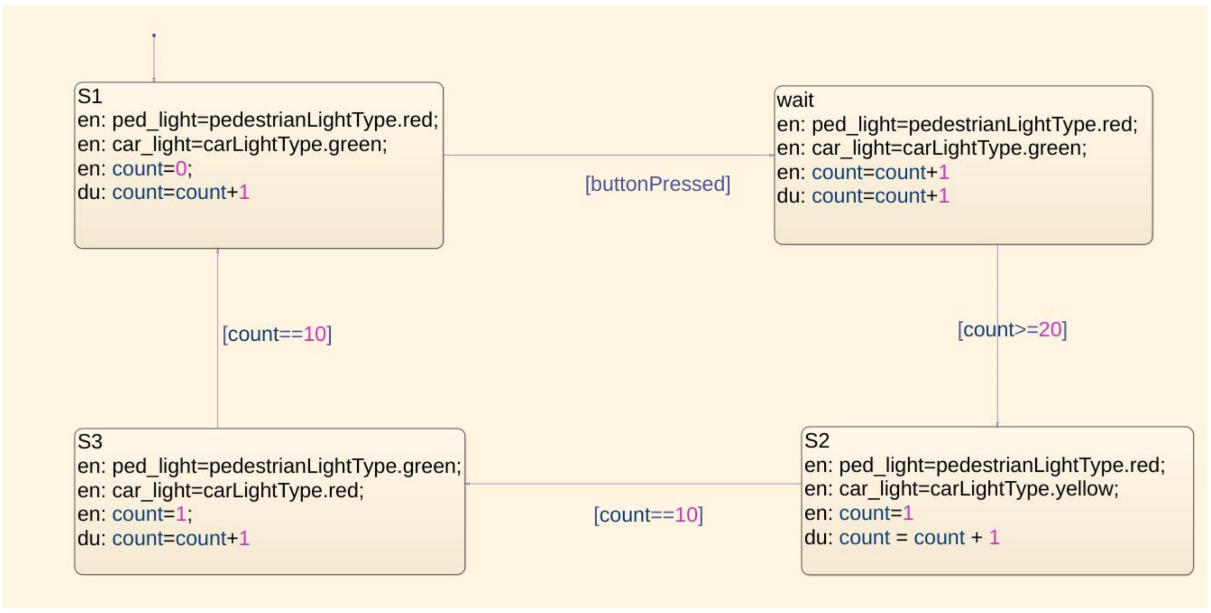

S1
en: ped_light=pedestrianLightType.red;
en: car_light=carLightType.green;
en: count=0;
du: count=count+1
[buttonPressed]
wait
en: ped_light=pedestrianLightType.red;
en: car_light=carLightType.green;
en: count=count+1
du: count=count+1
[count==10]
[count>=20]
S3
en: ped_light=pedestrianLightType.green;
en: car_light=carLightType.red;
en: count=1;
du: count=count+1
[count==10]
S2
en: ped_light=pedestrianLightType.red;
en: car_light=carLightType.yellow;
en: count=1
du: count = count + 1

Figure 12. Stateflow non-periodic traffic light system

3.2.2.1 Verification conditions

1) Light for pedestrians and for cars never is the green at the same time

The inputs 1 and 2 on the graph represents vehicle and pedestrian traffic lights respectively.

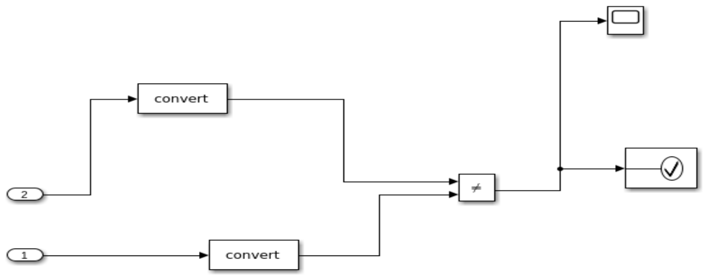


Figure 13. Simulink observer model

2) Green light for pedestrian will stay during 10 time units

To prove this property there is need to create an additional subsystem in Simulink.

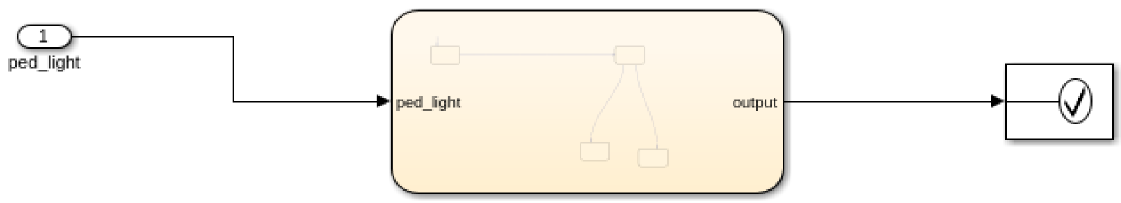


Figure 14. Simulink observer model

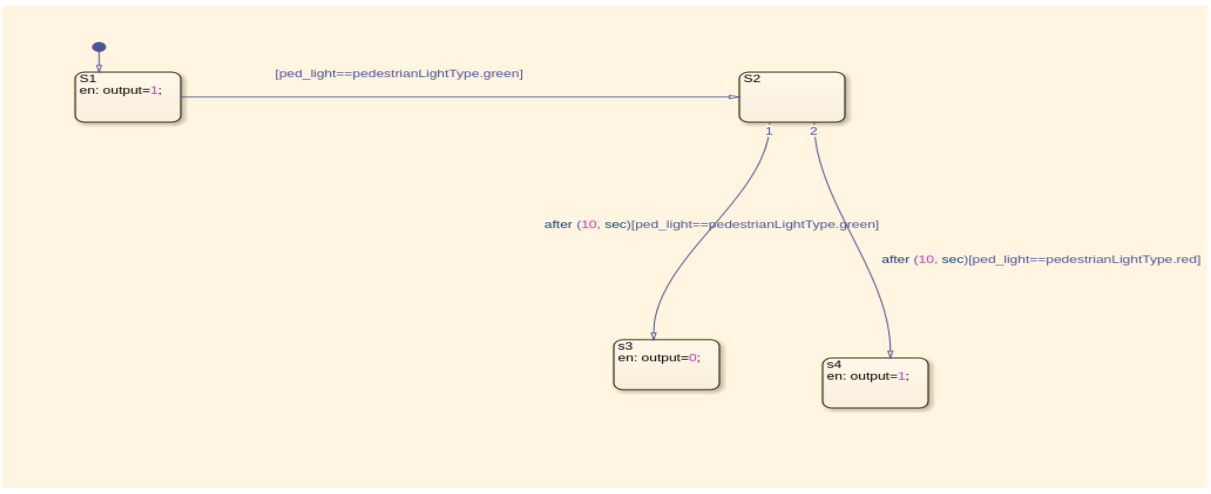

S1
en: output=1;
[ped_light==pedestrianLightType.green]
S2
1
2
after (10, sec)[ped_light==pedestrianLightType.green]
after (10, sec)[ped_light==pedestrianLightType.red]
s3
en: output=0;
s4
en: output=1;

Figure 15. Stateflow model inside the observer

Scope block for the verification:

Scope contains three inputs

1. Global clock
2. Vehicle light types
3. Pedestrian light types

From the scope by manual counting we can verify:

Previous two properties:

1. Light for pedestrians and for cars never is green at the same time: from the graphs it is shown that when the light for pedestrian is green the light for cars is red thought the whole sample time and visa-versa.

2. When the light for the pedestrian gets the green it stays on green during 10 seconds and then switches to red: from the graphs it is shown that after 30 seconds the pedestrian light gets green and after 10 seconds it gets red.

- At least during 20 time units there will be green for cars

- Pedestrian gets the green light in at least 30 seconds

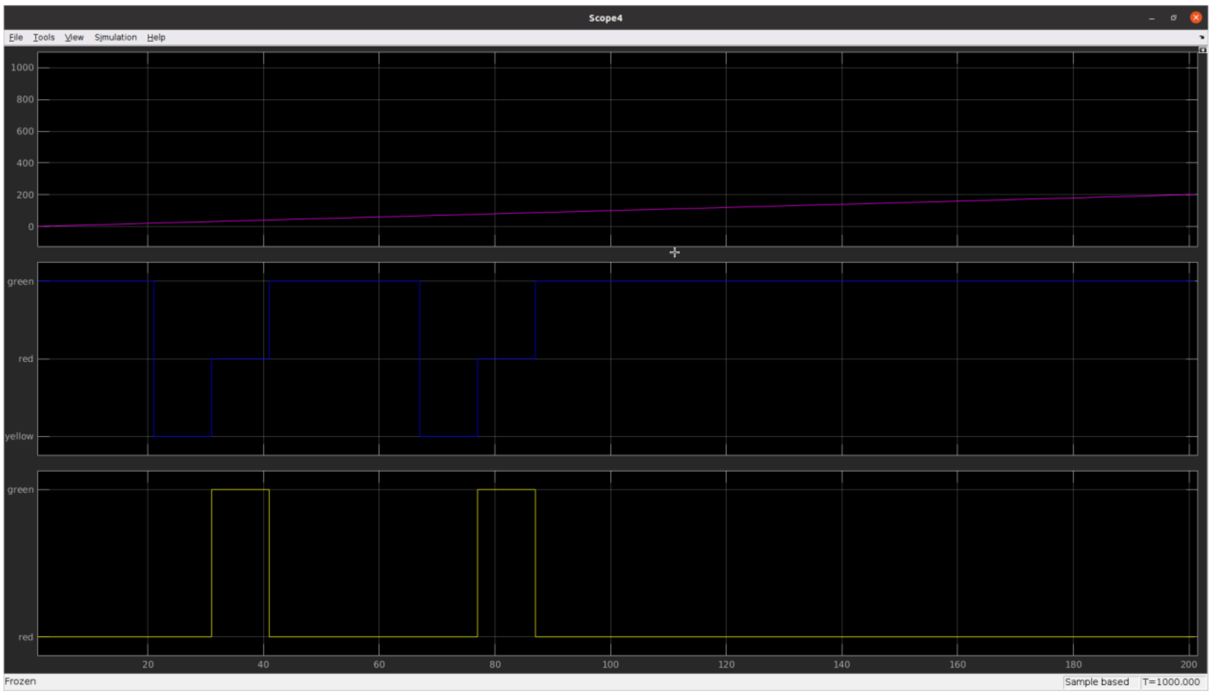


Figure 16. Simulation in scope block

The whole model contains a Stateflow chart where the main logic of traffic light system is specified. Additionally, there are created two subsystems for property verification and scope blocks to visually display the input and output values at a time.

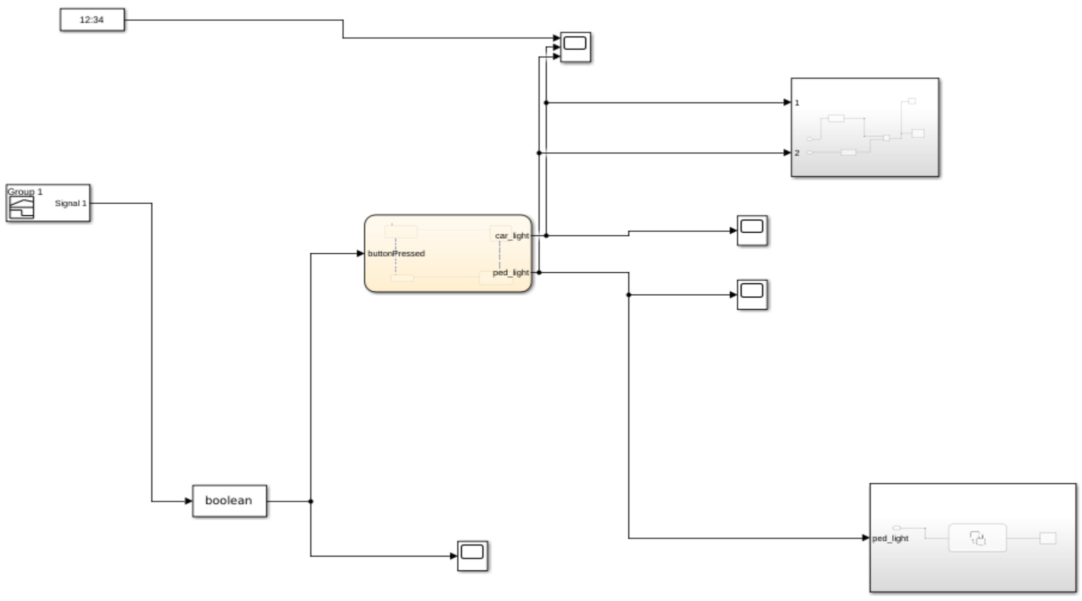


Figure 17. Simulink whole none-periodic traffic light system

3.3 Analysis of an experiment:

In the presented experiment, the non-periodic traffic light system is constructed in both Simulink/Stateflow and in Uppaal modelling tools. The main focus of the experiment is to compare the verification performance of modelling tools on a simple non-periodic traffic light system.

In the Stateflow model we used two methods to verify properties: first using observers conaining assertion blocks and the second using visually observable scope blocks.

1) The Assertion block checks whether any of the elements of the input signal are zero. If all of the elements are nonzero, the assertion is accurate, and the block does nothing. If not, the block halts the simulation and returns an error message by default. To implement the verification using assertions, subsystem blocks were created in which the verification logic using different Simulink blocks were modelled (see Figure 14 and figure 15) . The main disadvantage of verification using assertion blocks is that it is laborious – you have to construct a model for each situation. Verifying one single property requires creating a subsystem with logic inside, which sometimes is more complex than creating the model itself.

2) The second method using scope blocks hardly can be considered as a verification. Scopes are mainly used to visualize simulation data and observe it. They are a helper tool for verification by human – as you do not describe what you want from the system you cannot expect that there is an error message or trace provided.

Using UPPAAL, the verification is much efficient and straightforward. As there is no dedicated query language for describing properties in Simulink, the only possibility to describe the verified properties is to construct another model – an observer. Compared to the Simulink/Stateflow, Uppaal can verify safety and temporal properties using a one-line query language containing mathematical notations. Based on the experiment, Uppaal showed better capabilities of verifying the safety and temporal properties.

4 The proposed Solution

4.1 EMF metamodel for Uppaal

In developing the translation from Stateflow to the Uppaal model, one of the subtasks was to develop an EMF metamodel (see Figure 18) for manipulating elements of the Uppaal model. Similar metamodel for reading Stateflow model was existing before and the goal was to use similar technology for working with both Stateflow and Upaal. The EMF framework provides full automation for generating the Java for working with model elements and for reading/writing XMI files with the model.

As a starting point, we used an open-source metamodel given in in the Uppaal documentation [26]. Unfortunately, the published version of the metamodel was not compatible with the current version of Uppaal (the last commit to the repository was made at 2016). In order to make generated XMI files readable by Uppaal, all the classes except *NamedElement*, *Transition, and Template* classes were newly added. Also some of the features in the metamodel needed special tuning to get them serialized exactly in the format Uppaal expected while reading the file.

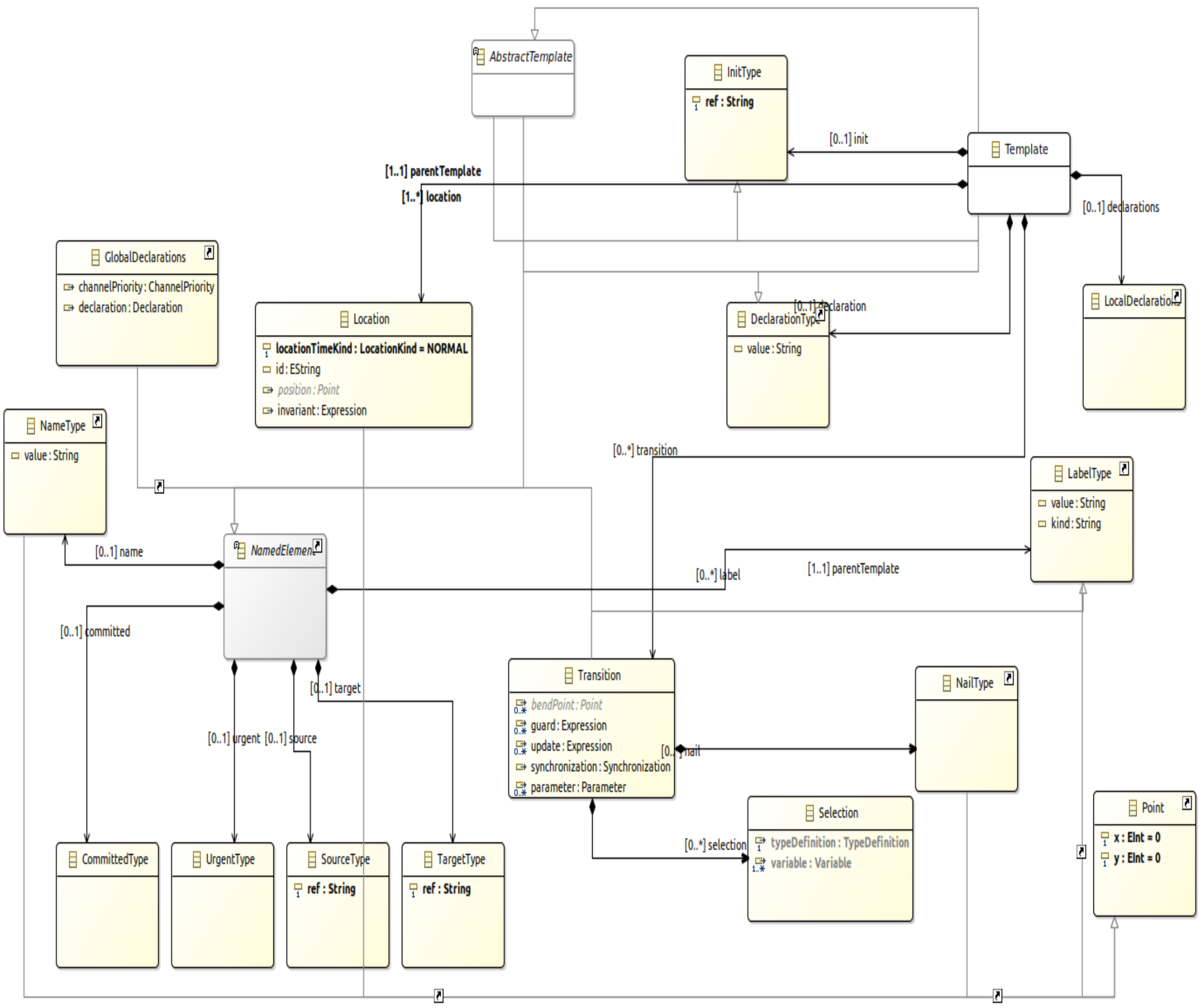


Figure 18: EMF metamodel for Uppaal

4.2 The translation workflow

The translation from Stateflow to Uppaal models has been developed beseed on EMF metamodels. For both tools, Java classes and methods have been generated. The process flow is the following: export Stateflow model in xmi form, import it in eclipse modelling framework, read the model, modify elements and map them to Uppaal elements using classes and methods generated from metamodel. Finally, export the data in XML form (See Figure 19) .

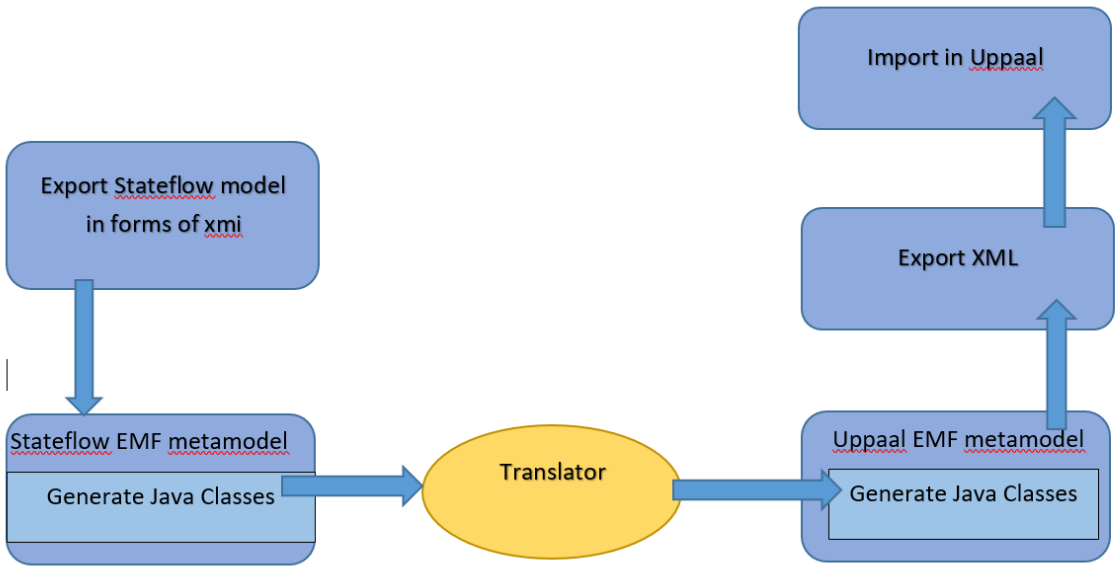


Figure 19. Translation block schema

4.3 Mapping from Stateflow to Uppaal

4.3.1 State

4.3.1.1 Mapping

Stateflow concept	Uppaal concept	Example
Empty state	Location	State_1
Entry action	Update on incoming edge of the location	State_1
During action	Self-transition with an update	State_1
Exit action	Update on outgoing edge from the location	State_1
Composite state	Parallel templates	State_2

Table 4 – State Mapping

4.3.1.2 Empty state

Empty Stateflow states can be translated as locations in the Uppaal model

4.3.1.3 State actions

Stateflow state can contain, entry, during, and exit while Uppaal is not supporting the same representation.

1. Translate entry action
    a. To translate the entry action of state, one of the ways is to add the entry condition as an update in the incoming transition of the Uppaal model.
2. Translate during action
    a. During action is executed when the state is active and there is no valid outgoing transition. To translate the during action in Uppaal first it's needed to split translation into following cases:
        i. Outgoing transition with empty condition - Transition without any condition or event is always valid outgoing transition. In that case

action is never executed. That's why if outgoing transition is free from conditions and events, translating during action can be omitted.

   ii. Outgoing transition with condition: To translate during action when outgoing transition has condition one of the ways is to create self-loop with committed location. Using self-loop, the value in during action will get updated until the outer transition will get valid. When the condition of outer transition will get valid the state still will stay with two valid outer transitions. Compared to Stateflow in Uppaal transitions are not prioritized and they can execute non-deterministically. To avoid executing self-transition it's needed to add the negation of guard condition that got enabled after executing during action.

3. Translate exit action
   a. Exit action in Stateflow is executed when the system leaves the state. So, when outer transition from state is executed the value of exit action is also executed. To represent this behaviour in Uppaal model one of the ways is to update the value on outer transition of the state

Example: State_1

Description

When s1 state is enabled output value is incremented and because there is no valid outgoing transition enabled for s1 state the during action is executed. When count is 3 and condition is satisfied, the transition between s1 and s2 is executed which also means that the exit action is executed and the system goes from s1 to s2 with updated values.

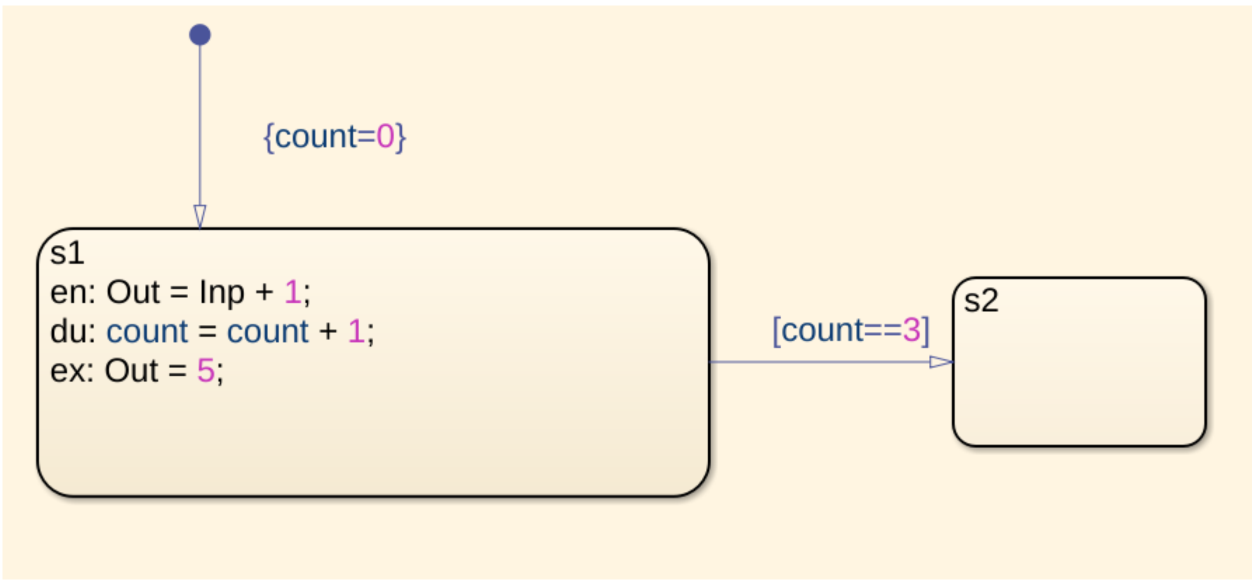


Figure 20. Stateflow example model for states

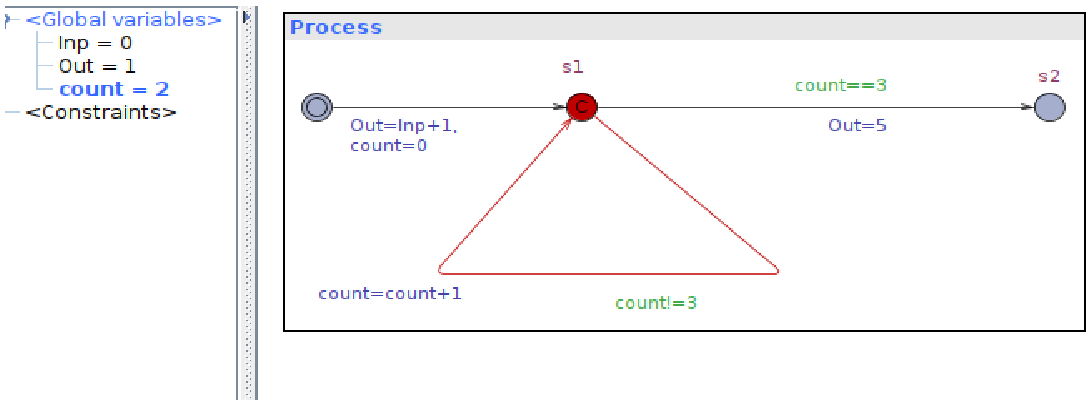


Figure 21. Uppaal example model for states

4. Superstates
   a. Stateflow models are often presented with composite states. Composite state is composed of super-states and sub-states. In Uppaal there are no composite states but they can be represented using parallel templates and channel synchronisation. States which are directly connected to each other create parallel templates and communication between them is based on channel synchronization.

Example: State_2

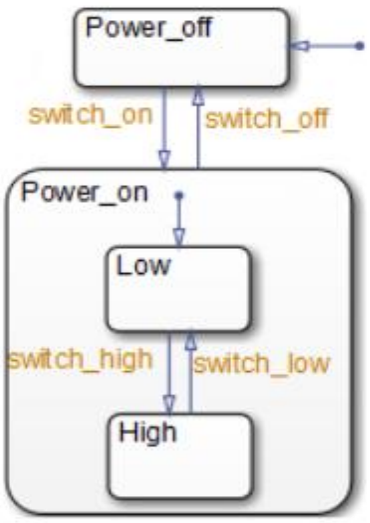


Figure 22. Stateflow example model for Superstates

Every state that are directly connected to each other create new templates.

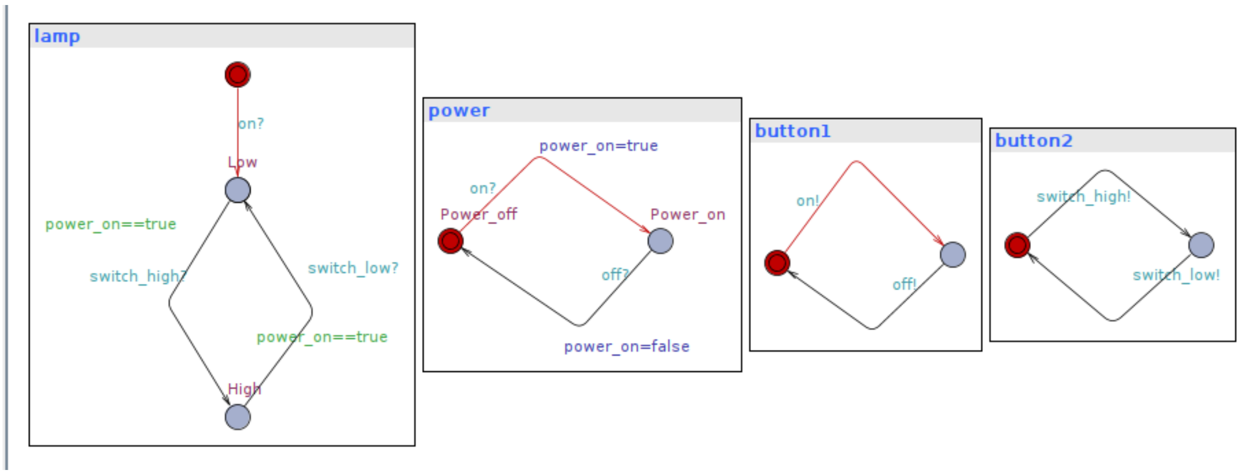


Figure 23. Uppaal example model for Superstates

4.3.2 Transition

4.3.2.1 Mapping

Stateflow concept	**Uppaal concept**	**Example**
Default transition (no guard or event)	Default location with transition	State_1
Transition with event	New template for event with channel synchronisation	Transition_2
Transition with guard	Guard + Invariant	Transition_1
Transition with action	Assignment	Transition_2

Table 5 – Transition Mapping

4.3.2.2 Default transition

a. The default transition which does not contain any labels (events, guards, assignments) can be translated as an initial location of a state which has an inner default transition and the state pointing to the
   Example:

b. The default transition which contains any labels (e.g assignment) can be represented by adding initial location to the Uppaal model and translating transition between initial location and the location that represents the state with inner default transition.
   Example:

4.3.2.3 Transition between states

c. A Stateflow transition which is between two states and doesn't contain any events or guard conditions, in Stateflow is executed immediately after processing the state actions is completed. In Uppaal the activation of a transition without guard and synchronisation is not determined -- it can happen any time. To present the same behaviour that Stateflow model has we need to force the system to leave the location. One of the ways to execute transition directly in Uppaal is to make the location, from where the outer transition is unconditional, committed.

4.3.2.4 Transition with guard

4.3.2.4.1 Guard with time constraints

In Uppaal, time constraints are represented as clocks. To translate the guard condition with timing behaviour following steps should be done:

1. Clocks should be created in Uppaal
2. Guard should be created with clocks in the corresponded transition
3. To make the model probabilistic, the invariant that is less or equal to the guard condition should be added to the location which has an outer transition with guards of timing constraints.

4.3.2.4.2 Guard without time constraints:

Sometimes in Stateflow the value of guard condition is given by computational value which isn't connected to real value clocks. To translate the guard condition without timing behaviour following steps should be done (see Figure 24):

1. Declare variable type of guard in global declaration
2. Create self-loop with guard representing negation of all guard conditions on outgoing transitions

Example Transition_1

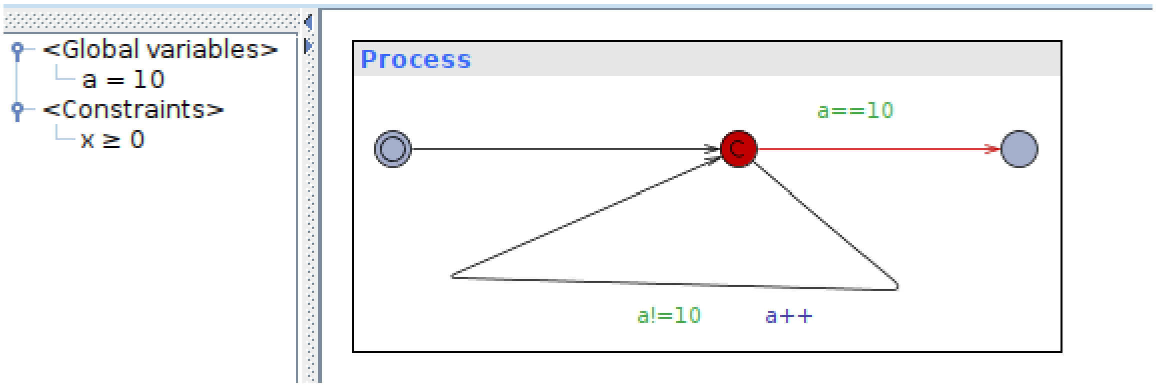


Figure 24. Uppaal example for guard with time constraint

4.3.2.5 Transition with events

In Uppaal the events are modelled through the concept of channel synchronisation. In Stateflow the execution of transitions happens when the event on transition is triggered. To represent the same behaviour in Uppaal one of the ways is to create separate templates for every event presented in Stateflow and make synchronisation between processes.

Example Transition_2

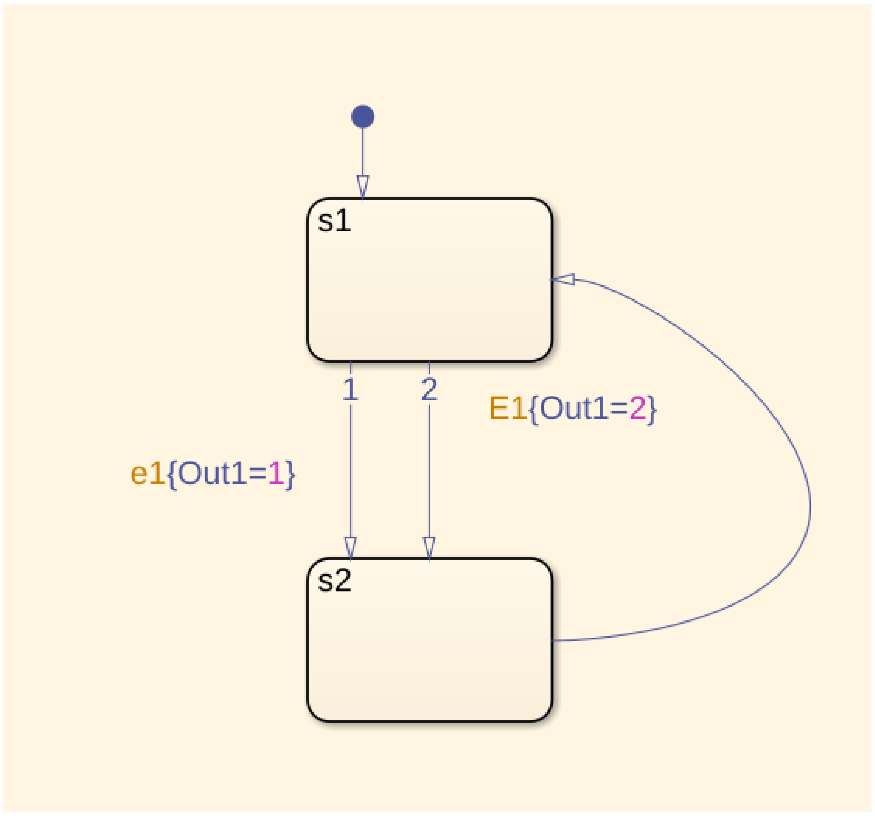


Figure 25. Stateflow example model for Explicit Events

The corresponding Uppaal model with two events template and synchronisation mechanism.

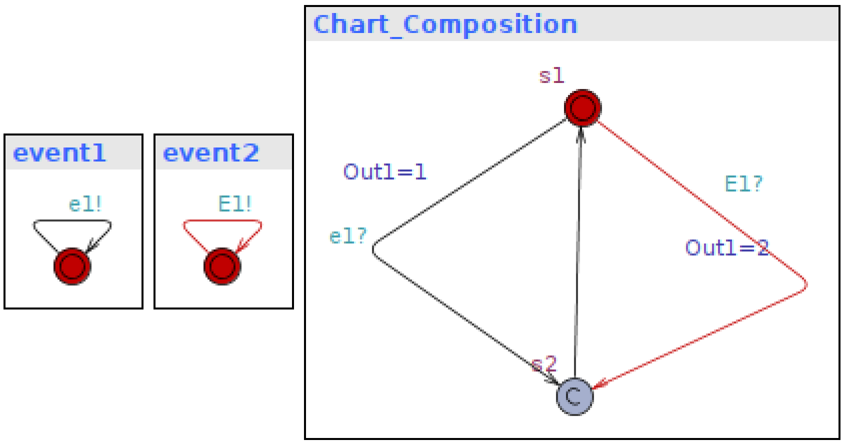


Figure 26. Uppaal example model for Explicit Events

4.3.2.5.1 Complex event logic:

In Stateflow events on the transition can be presented in a complex way. For example: e1|e2|e3. Based on the example presented in part a) events can be translated as separate templates. But in Uppaal it's not supported to represent synchronisation with composite logic. We need to take into account that the transition in case of Stateflow based on the example is executed when one of the following cases is triggered

1. e1
2. e2
3. e3
4. (e1 and e2)
5. (e1 and e3)
6. (e2 and e3)

One of the ways to represent this behaviour in Uppaal is to make templates for all possible scenarios. If the number of events separated with or logic is n to cover all possible scenarios in Uppaal it's needed to create templates which number is calculated from the following formula where the n represents number of events:

$$n + \sum_{i=1}^{n-1} i$$

4.3.2.5.2 Transition with an update

Update element on transition is conceptually the same as Uppaal assignment. That's why the Update can be directly translated as assignment in Uppaal.

4.3.3 Junctions

In Stateflow connective junctions are decision points in the system. When the system is on the junction state the system is not staying on the junction and immediately leaves the junction state.
If outgoing transitions of junctions are not executed junctions have backtrace behaviour which means that the system is staying in the previous state before the junction.
The junctions are executed in the following cases:

1. On outer transition there is no condition
2. On outer transition there is guard condition

In Uppaal there is no direct representation of the junction. One of the possible ways to represent it is using committed location.

To represent the Stateflow junctions in Uppaal the following manipulations should be done:

Translate all junction state as committed location

Check if the junction's outer transition contains the guard or not

1. Junction followed by the transition without guard condition:
    a. There is no need to consider backtrace behaviour of junction because if there is no condition on junction's outer transition executes straightforwardly.
2. Junction followed by the transition with guard condition:
    a. In Stateflow when junction is followed with conditional transition if condition is not true time is freezed and system stays in the previous state before the transition. That's the backtrace behaviour of the junction. To represent the same behaviour in Uppaal one of the ways is to add transition that goes on previous state and on this transition put the negation of all conditions.

Example: Junction_1

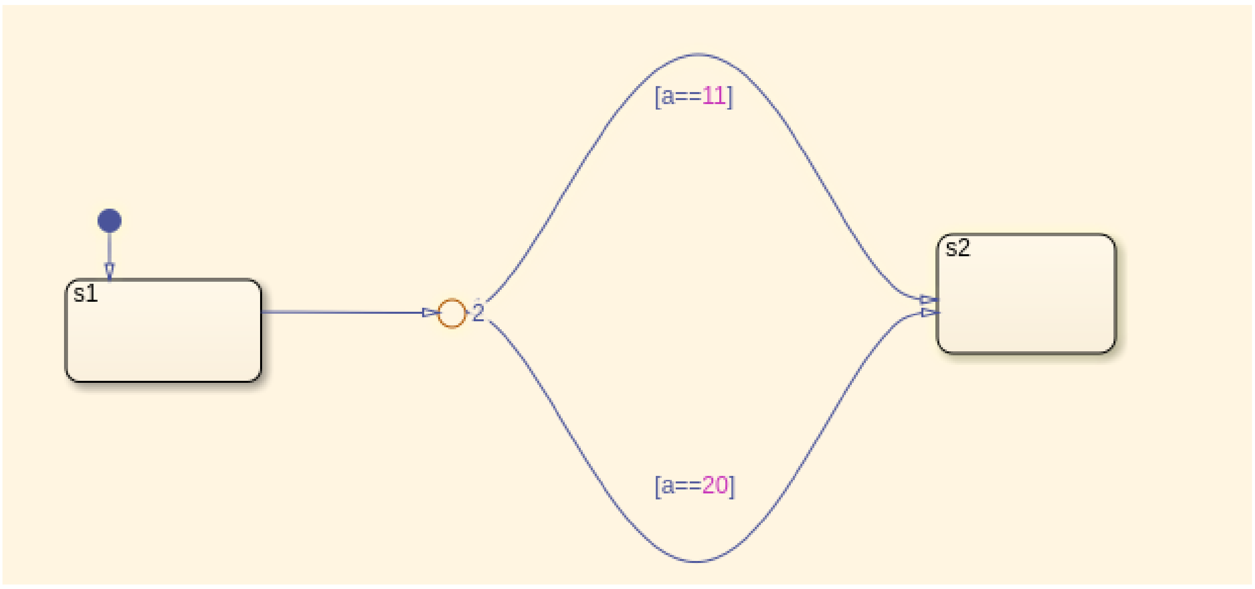


Figure 27. Stateflow example model for a Juncion

In the Uppaal model the committed state represents the junction which goes to the previous state if the conditions on outer transition to the next state are not true (see Figure 28).

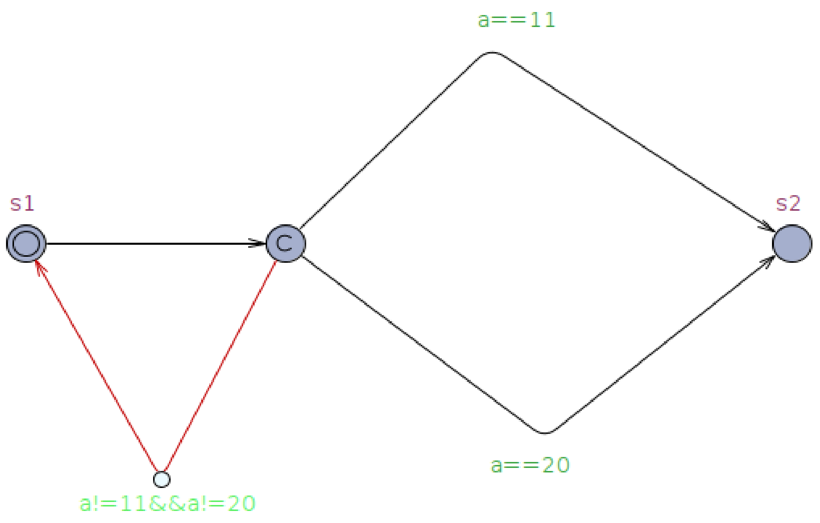


Figure 28. Uppaal example model for a Juncion

4.3.4 Transition prioritization

Stateflow models are fully probabilistic, which also applies to transition execution order. In Stateflow every transition has prioritization number to determine which transition should be taken.

After starting the simulation of bellow Stateflow model s1 state is activated and the systems checks if condition of the transition with the priority number one is satisfied if not only after checks the condition on the second transition (see Figure 29).

To represent the same behavior in Uppaal tool the negated condition value of first transition needs be added as a guard condition to the second transition (see Figure 30).

Example: Transition prioritization1

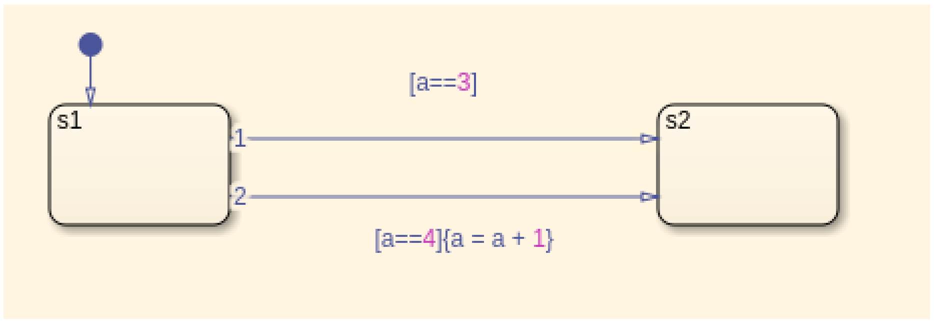


Figure 29. Staateflow example for probabilistic execution flow

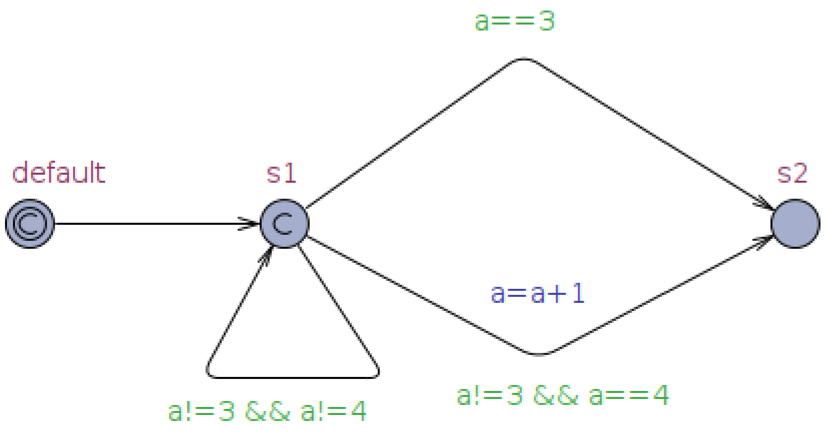


Figure 30. Uppaal example for probabilistic execution flow

4.3.5 Temporal logic

4.3.5.1 Mapping

Stateflow Concept	**Uppaal concept**	**Example**
After (n, time_unit)	Guard “time” as clock (greater than n) + invariant (less or equal to n+1)	Temporal_logic1
Before (n, time_unit)	Guard “time” as clock (less than n) + invariant (less than n)	Temporal_logic1
After (n, implicit_event)	Guard “implicit_event” as clock (greater than n) + invariant (less or equal to n+1)	Temporal_logic2

Before (n, implicit_event)	Guard “implicit_event” as clock (less than n) + invariant (less than n)	Temporal_logic1
At (n, implicit_event)	Guard “implicit_event” as clock (equal to n) + invariant (less or equal to n)	Temporal_logic2
After (n, explicit_event)	Guard variable form “emplicit_event” template (greater than n) + invariant (less or equal to n+1)	Temporal_logic3
Before (n, explicit_event)	Guard variable form “emplicit_event” template (less than n) + invariant (less than n)	Temporal_logic3
At (n, explicit_event)	Guard variable form “emplicit_event” template (equal to n) + invariant (less or equal to n)	Temporal_logic3

Table 6 – Temporal Logic Mapping

4.3.5.2 Absolute time temporal logic

In Uppaal, absolute time constraints can be represented as clocks, but to maintain the same Stateflow behaviour, additional modifications are needed.

4.3.5.2.1 After (n, time_unit)

To translate “After (n, time_unit)” following steps should be done:

1. Create clock variable
2. Transform After (n, time_unit) as a guard condition with clock variable is greater than n
3. To remove time non-determinism, add clock variable less or equal to n+1 as invariant to the location
4. Update clock value to 0 to maintain the feature of incrementing absolute time towards the state.

4.3.5.2.2 Before (n, time_unit)

To translate the Before (n, time_unit) following steps should be done:

1. Create clock variable
2. Transform Before (n, time_unit) as the guard condition with clock variable less than n
3. To remove time non-determinism, add clock variable less then n as invariant to the location
4. Update clock value to 0 to maintain the feature of incrementing absolute time towards the state [6].

Example: Temporal_logic1

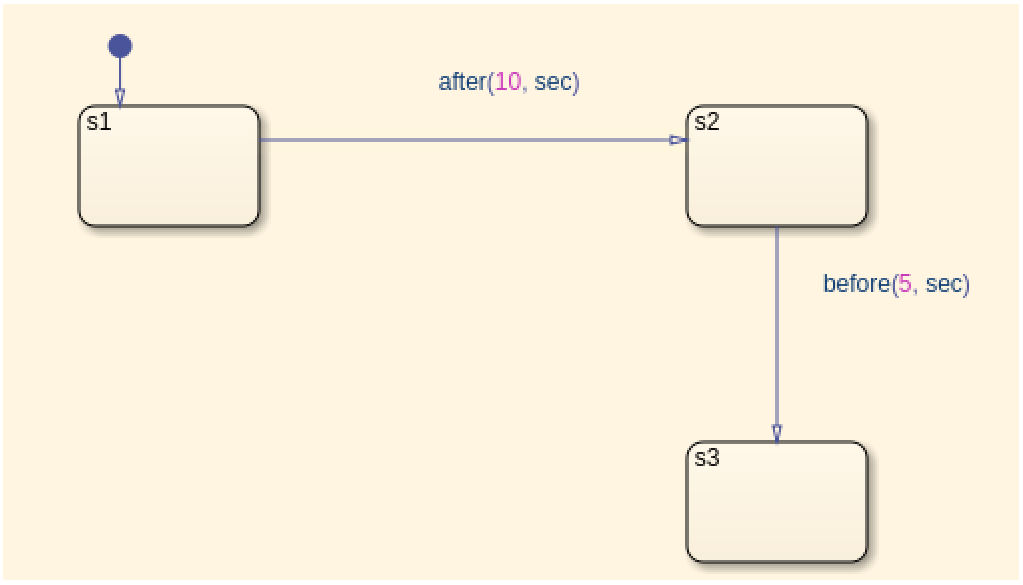


Figure 31. Stateflow example for temporal logic with absolute time

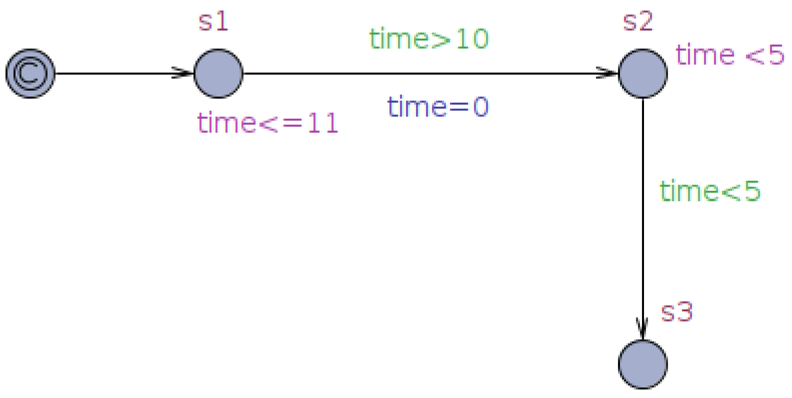


Figure 32. Uppaal example for temporal logic with absolute time

4.3.5.3 Temporal logic with implicit event

In Uppaal, there is no direct representation of temporal logic with an implicit event. We use one of the most popular implicit event tick and present the transformation approach.

4.3.5.3.1 After (n, tick)

To translate After (n, tick) following steps should be done:

1. Create clock variable
2. Transform After (n, tick) as a guard condition with clock variable is greater than n
3. To remove time non-determinism, add clock variable less or equal to n+1 as invariant to the location
4. Update clock value to 0 to maintain the feature of incrementing absolute time towards the state.

4.3.5.3.2 Before (n, tick)

To translate the Before (n, tick) following steps should be done:

1. Create clock variable
2. Transform Before (n, tick) as the guard condition with clock variable less than n
3. To remove time non-determinism, add clock variable less then n as invariant to the location
4. Update clock value to 0 to maintain the feature of incrementing absolute time towards the state.

4.3.5.3.3 At (n, tick)

To translate the At (n, tick) following steps should be done:

1. Create clock variable
2. In the place of At (n, tick) place the guard condition with clock variable equal to n
3. Add invariants to the location to eliminate time non-determinism
4. Update clock value to 0.

Example: Temporal_logic2

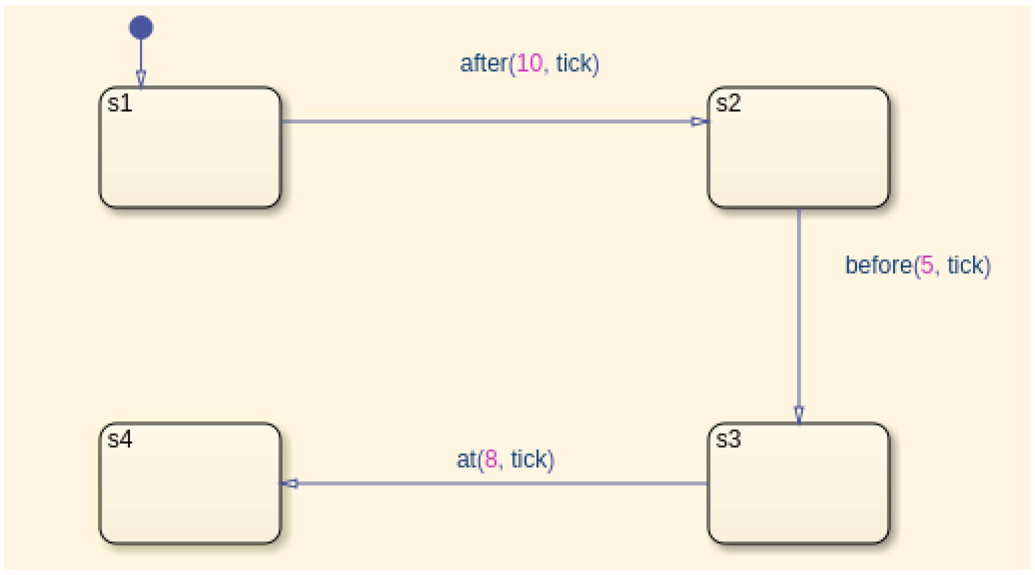


Figure 33. Stateflow example for temporal logic with an implicit event

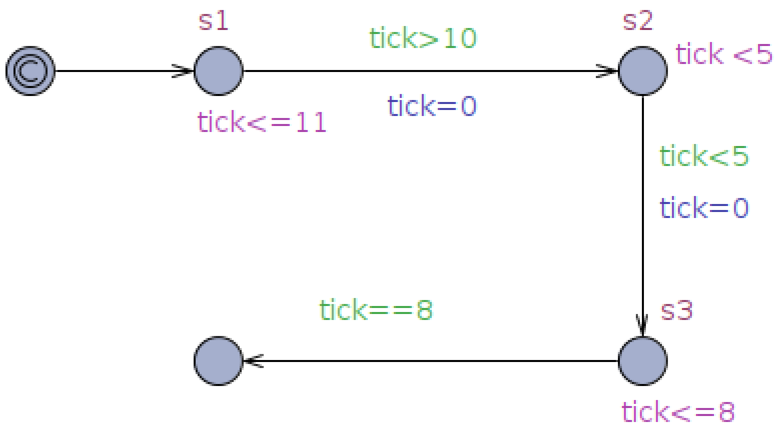


Figure 34. Uppaal example for temporal logic with an implicit event

4.3.5.4 Temporal logic with explicit event

In Uppaal, there is no direct representation of temporal logic with an explicit event and it's not possible to represent it using clock, because event occurrence isn't based on real time value.

4.3.5.4.1 After (n, event_name)

To translate the After (n, time_unit) following steps should be done:

1. Create new template for event and update variable on self-loop.

2. In the place of Before (n, event_name) place the above variable as guard with condition variable greater than n
3. To eliminate flow non - determinism, make location committed with self-loop of negated all outgoing transition
4. Update clock value to 0

4.3.5.4.2 Before (n, event_name)

To translate the Before (n, time_unit) following steps should be done:

1. Create new template for event and update variable on self-loop.
2. In the place of Before (n, event_name) place the above variable as guard with condition variable less than n
3. To eliminate flow non - determinism, make location committed with self-loop of negated all outgoing transition
4. Update clock value to 0

4.3.5.4.3 At (n, event_name)

To translate the At (n, event_name) following steps should be done:

1. Create new template for event and update variable on self-loop.
2. In the place of Before (n, event_name) place the above variable as guard with condition variable equal to n
3. To eliminate flow non - determinism, make location committed with self-loop of negated all outgoing transition
4. Update variable value to 0

Example: Temporal_logic3

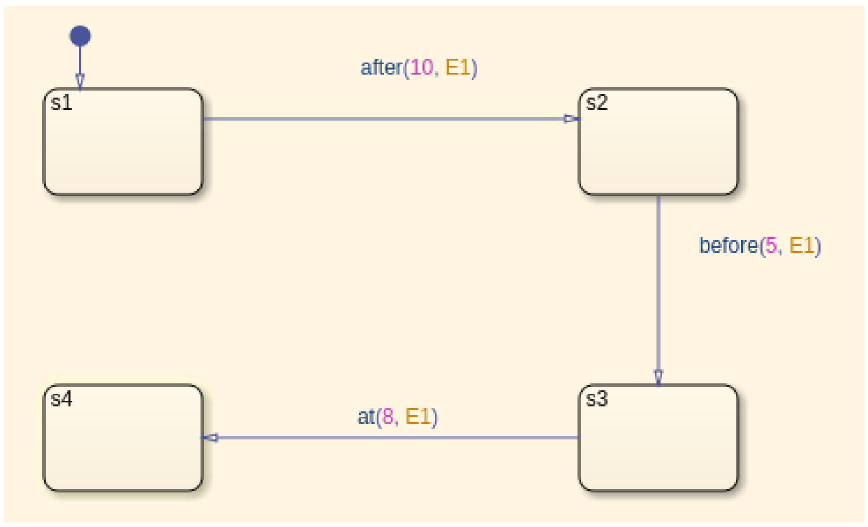


Figure 35. Stateflow example for temporal logic with an explicit event

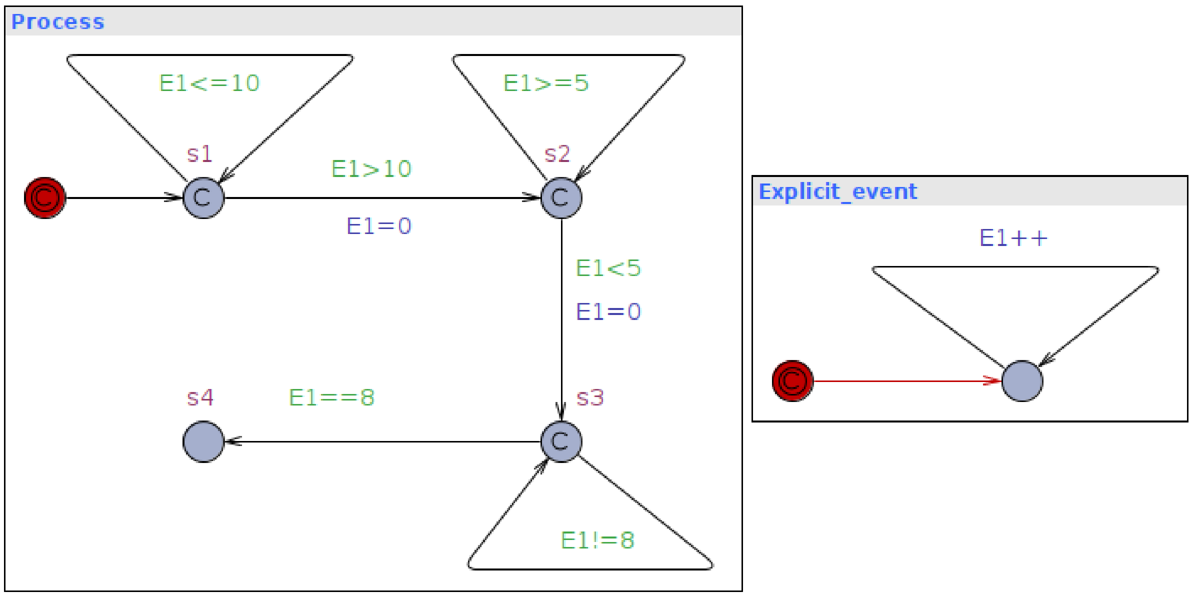


Figure 36. Uppaal example for temporal logic with an explicit event

4.3.6 And composition

Stateflow supports the modelling of different kinds of compositions. Two primary types of compositions are *or* and *and* compositions. In all the previous examples, only *or* composition has been used. However, in this section, we investigate how *and* composition can be represented in Uppaal model checking tool.

The difference between these two compositions is in state execution manner. In "or composition", states are executed sequentially, where only one state inside a chart could

be active at a time, while in *and* composition, activation of states happens in parallel and the several states could be active simultaneously.

After starting the simulation of Stateflow model s1, s11, s2, and s21, states getting active simultaneously (see Figure 37). To present the same behaviour in Uppaal, the following steps should be done:

1. Translate *and* and *or* compositions as templates

2. Create an additional template for synchronisation

3. Add broadcast channel synchronisation to maintain the correct execution order of created elements

When the simulation of the Uppaal model starts "activateAndComposition" template sends the synchronisation message to other templates, and the s1, s11, s2, and s21 states get activated simultaneously (see Figure 38).

Example: And_composition1

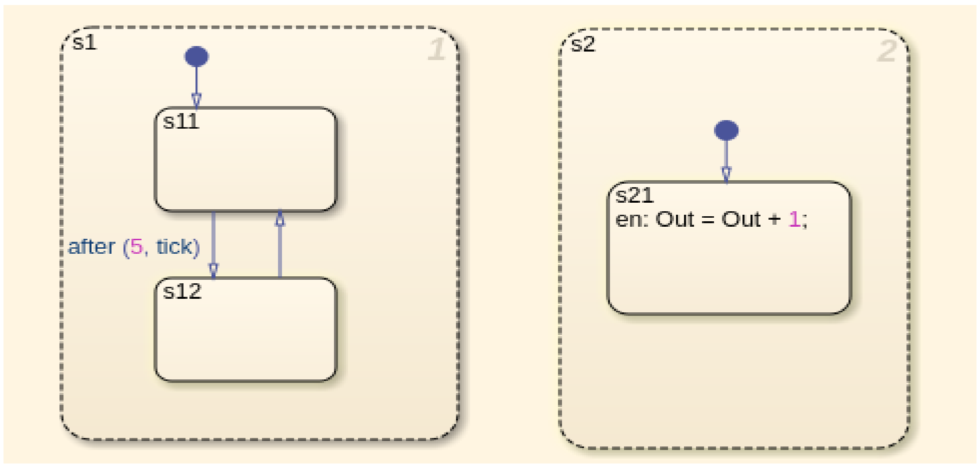


Figure 37. Stateflow example model for *And composition*

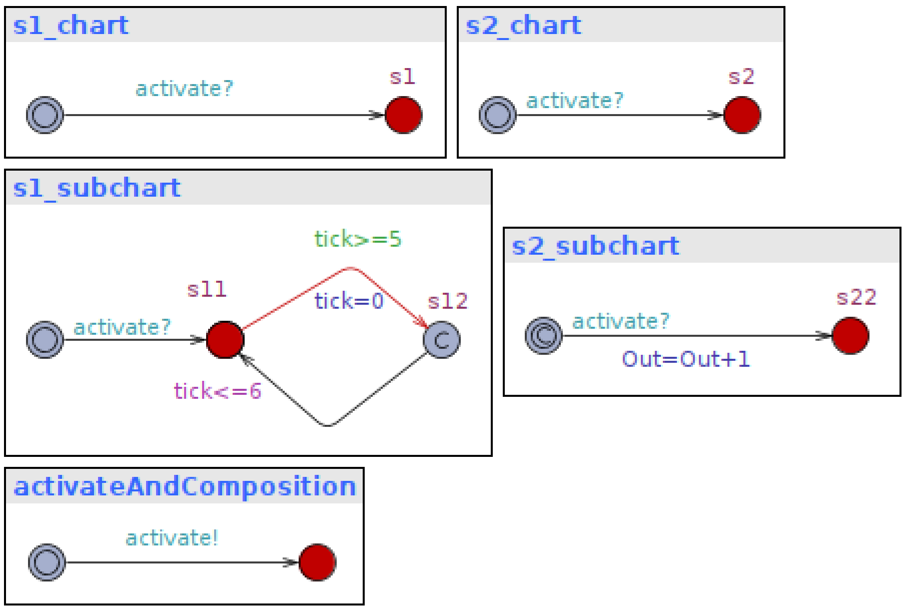


Figure 38. Uppaal example model for *And composition*

5 Verification of the translation

5.1 Case study

The proposed mapping pattern for translating from Stateflow to Uppaal models is tested on different case study models. The details of transformation of models is given by two specific case study models described below.

5.1.1 Model 1:

The elements of Stateflow model (See Figure 39):

- Or composition
- State with actions
- State with more than 1 outgoing transitions with prioritization
- Junction

The state change of the given model is triggered by the value of Y1 and condition statement. Model contains three types of state actions and transition prioritization. For example, when the system is in the s1 state the condition on transition of first priority is checked and if it's not satisfied then the system checks the condition of the second transition.

After applying the mapping (section 4.3) we get the Uppaal model (See Figure 40)

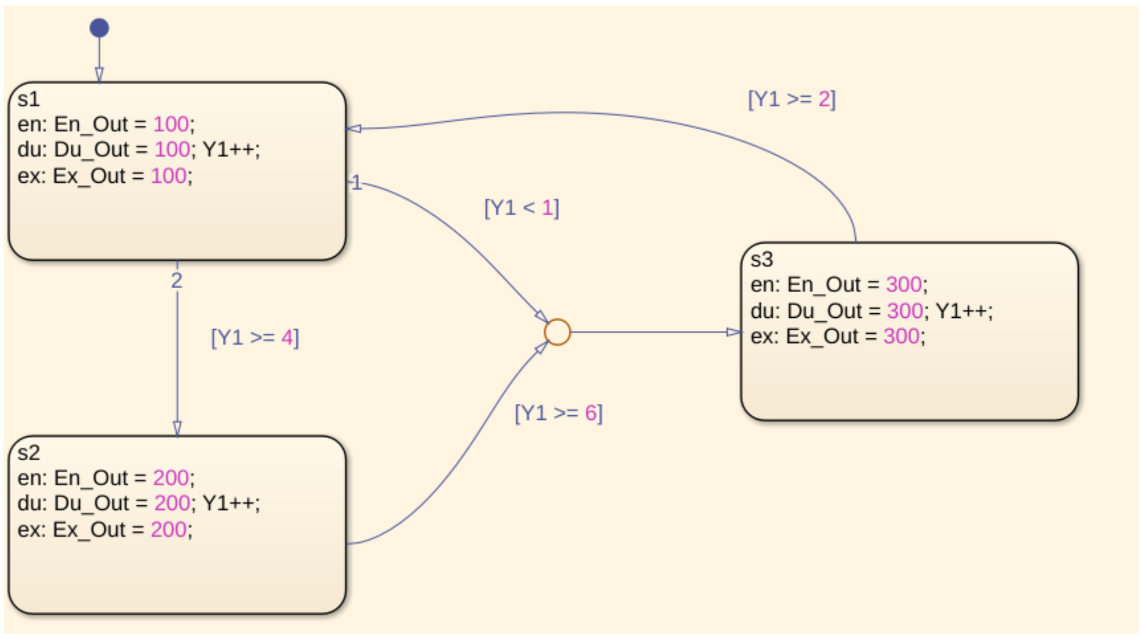


Figure 39. Case study model 1 in Stateflow

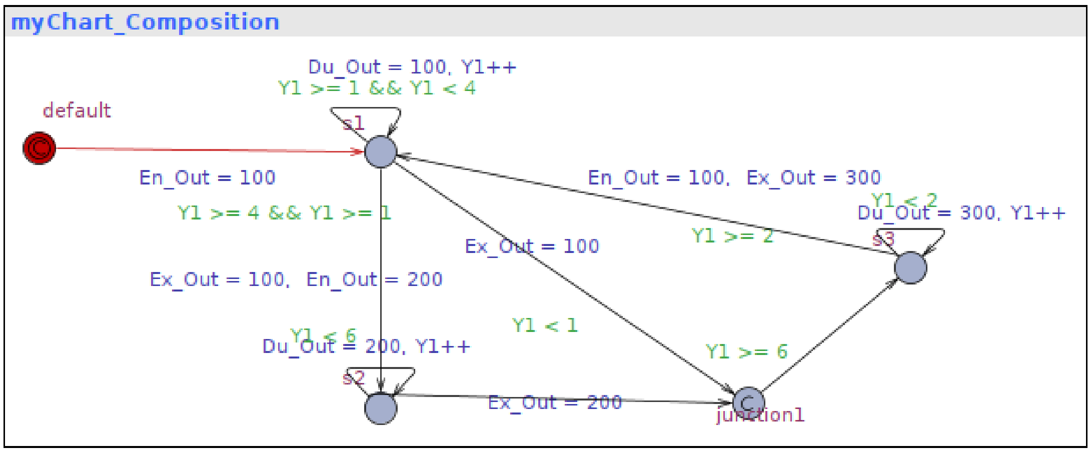


Figure 40. Case study model 1 in Uppaal

5.1.2 Model 2

The elements of Stateflow model (See Figure 41):

- Or composition
- End composition
- State actions with temporal logic
- Implicit event
- Supercharts
- Junction
- Integer array

From the start of the model simulation, s1, s11, s2, and s21 states are getting active simultaneously, and the state actions are executed based on temporal logic represented with an explicit event. Besides state actions, temporal logic is also presented as a condition on transition. For example, the system from s21 to s22 state moves when the guard condition on transition is satisfied, which happens exactly the "In1+1" tick.

After applying the mapping (section 4.3) we get the Uppaal model (See Figure 42)

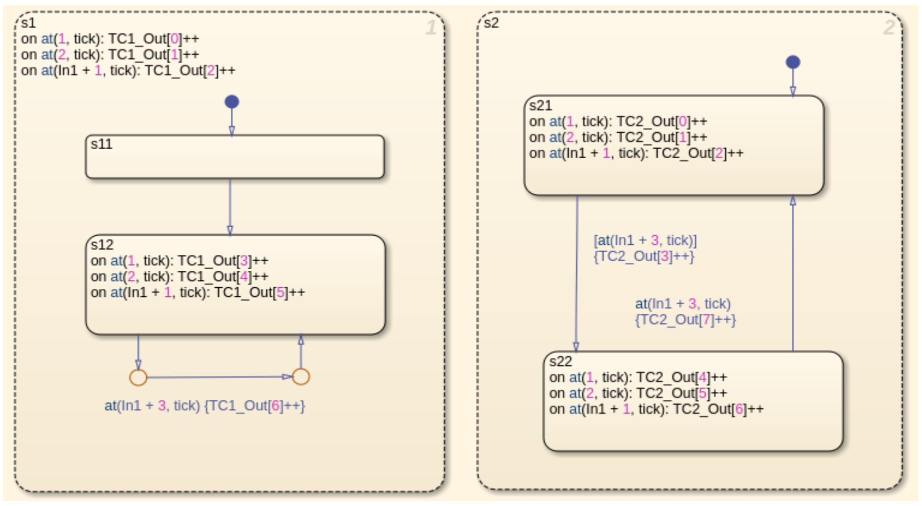

s1
on at(1, tick): TC1_Out[0]++
on at(2, tick): TC1_Out[1]++
on at(In1 + 1, tick): TC1_Out[2]++
1
s11
s12
on at(1, tick): TC1_Out[3]++
on at(2, tick): TC1_Out[4]++
on at(In1 + 1, tick): TC1_Out[5]++
at(In1 + 3, tick) {TC1_Out[6]++}
s2
2
s21
on at(1, tick): TC2_Out[0]++
on at(2, tick): TC2_Out[1]++
on at(In1 + 1, tick): TC2_Out[2]++
[at(In1 + 3, tick)]
{TC2_Out[3]++}
at(In1 + 3, tick)
{TC2_Out[7]++}
s22
on at(1, tick): TC2_Out[4]++
on at(2, tick): TC2_Out[5]++
on at(In1 + 1, tick): TC2_Out[6]++

Figure 41. Case study model 2 in Stateflow

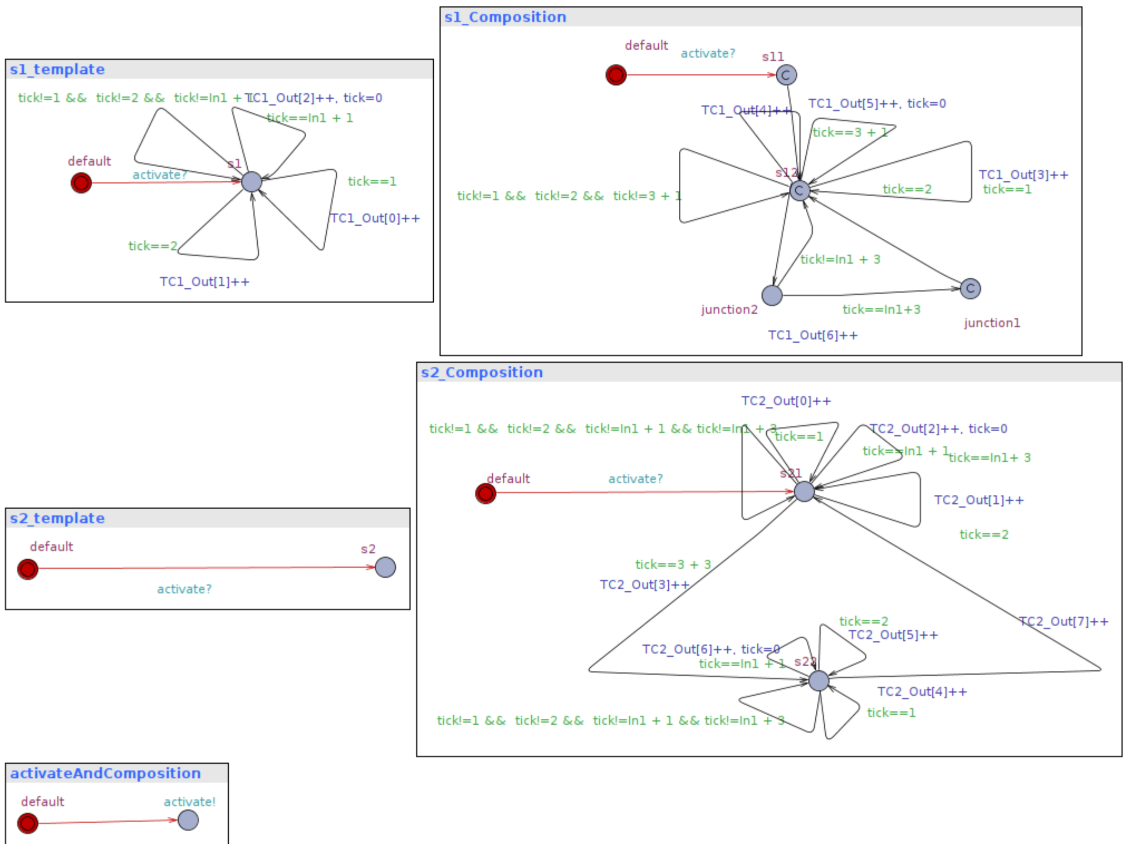


Figure 42 . Case study model 2 in Uppaal

6 Conclusion

6.1 Summary

MDE is widely used in developing real-time safety-critical systems where verification plays a vital role in maintaining system safe and liveness. One of the widely used MDE tools, Stateflow, cannot verify temporal, safety, and liveness properties and creates the need for a better verification tool. For that purpose, in this paper, we present an approach for translating Stateflow models to the Uppaal model checking tool. Uppaal model checking tool uses the theory of time automata and its query language, which represents the sufficient base for the verification.

In the process of developing the thesis a lot of models in both Stateflow and Uppaal were created. Before moving to the translation, the approach thesis presents one of thise experiments: a non-periodic traffic light system modelled in both Stateflow and Uppaal and presents the advantages of verification capabilities of the Uppaal model-checking tool.

The main goal of this work was first to analyse the advantages of the verification language of the Uppaal model checking tool, secondly to create the translation. The process of translation is divided into two main parts. First, an EMF metamodel for Uppaal described in section 4.1, which is used as an intermediate layer for manipulating Uppaal elements. Second, the mapping rules are described in section 4.3. The algorithm automating the translation from Stateflow to Uppaal model was developed based on those two components of translation. The verification of the proposed approach is demonstrated using example models from which two are presented in the chapter 5.

6.2 Future work

While working on this research, many experiments were performed, and many models with different variety of elements were translated. However, because of limited time the verification of the whole mapping stayed out of the scope of this research. Additionally, because the approach is not tested on large scale models, there is no proof about the translated model complexity represented in the Uppaal.

Another important aspect that remained outside the scope of this thesis and has to be developed further is the approach of defining verification conditions in Simulink model. Using a property language that enables to specify the contracts in earlier stage would potentially allow to hide Uppaal form the user for simpler models.

To sum up, this thesis can be considered a starting point of future research on developing verification using formal methods on a large scale Stateflow models and additionally, the paper can be used as a baseline for translating other Simulink blocks into the Uppaal model checking tool.

References

[1] Akdur, Deniz & Garousi, Vahid & Demirors, Onur. (2018). A survey on modeling and model-driven engineering practices in the embedded software industry. Journal of Systems Architecture. 91. 10.1016/j.sysarc.2018.09.007.

[2] Paz, Andrés & El-Boussaidi, Ghizlane. (2020) . Breesse: Bridging EMF, Simulink and Stateflow for Model-Based Design of Safety-Critical Systems. 10.1145/3417990.3421408.

[3] Leitner, F. (2008). Evaluation of the Matlab Simulink Design Verifier versus the model checker SPIN.

[4] Filipovikj P., Mahmud N., Marinescu R., Seceleanu C., Ljungkrantz O., Lönn H. (2016) Simulink to UPPAAL Statistical Model Checker: Analyzing Automotive Industrial Systems. In: Fitzgerald J., Heitmeyer C., Gnesi S., Philippou A. (eds) FM 2016: Formal Methods. FM 2016. Lecture Notes in Computer Science, vol 9995. Springer, Cham.

[5] Yang, Yixiao & Jiang, Yu & Gu, Ming & Sun, Jiaguang. (2016). Verifying simulink stateflow model: timed automata approach. 852-857. 10.1145/2970276.2970293.

[6] The Mathworks: Stateflow and Stateflow Coder, User's Guide. Release 13sp1 edn. (2003).

[7] Hamon, G., & Rushby, J. (2004, March). An operational semantics for Stateflow. In International Conference on Fundamental Approaches to Software Engineering (pp. 229-243). Springer, Berlin, Heidelberg.

[8] E. -. Olderog, "Formal methods in real-time systems," Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No.98EX168),Berlin, Germany, 1998, pp. 254-263, doi: 10.1109/EMWRTS.1998.685130.

[9] Bouyer, P. (2009). Model-checking timed temporal logics. Electronic notes in theoretical computer science, 231, 323-341.

[10] Naveed Ahmed Alizai, Master's degree, 2020, (leader) Leonidas Tsiopoulos; Jüri Vain, Bisimulation verification of UPPAAL Models, Tallinn University of Technology, Faculty of Information Technology, Institute of Software Science.

[11] Behrmann G., David A., Larsen K.G. (2004) A Tutorial on Uppaal. In: Bernardo M., Corradini F. (eds) Formal Methods for the Design of RealTime Systems. SFM-RT 2004. Lecture Notes in Computer Science, vol3185. Springer, Berlin, Heidelberg.

[12] Bošnački, D., Wijs, A. Model checking: recent improvements and applications. Int J Softw Tools Technol Transfer 20, 493–497 (2018). https://doi.org/10.1007/s10009-018-0501-x.

[13] Axelsson, Roland & Hague, Matthew & Kreutzer, Stephan & Lange, Martin & Latte, Markus. (2010). Extended Computation Tree Logic. Lecture Notes in Computer Science. 10.1007/978-3-642-16242-8_6.

[14] "UPPAAL." Semantics of the Requirement Specification Language: UPPAAL Documentation, docs.uppaal.org/language-reference/requirements-specification/semantics/.

[15] University of Pennsylvania (2012). UPPAAL TUTORIAL, Part-4-Uppaal-Input, www.seas.upenn.edu/~lee/10cis541/lecs/part-4-uppaal-input-new-1x2.pdf.

[16] Daw, Z., & Cleaveland, R. (2015). Comparing model checkers for timed UML activity diagrams. Science of Computer Programming, 111, 277-299.

[17] L. Baresi, M. M. Pourhashem Kallehbasti and M. Rossi, "Efficient Scalable Verification of LTL Specifications," 2015 IEEE/ACM 37th IEEE International

Conference on Software Engineering, 2015, pp. 711-721, doi: 10.1109/ICSE.2015.84.

[18] J. Li, J. Li and F. Zhang, "Model Checking UML Activity Diagrams with SPIN," 2009 International Conference on Computational Intelligence and Software Engineering, 2009, pp. 1-4, doi: 10.1109/CISE.2009.5363181.

[19] Filipovikj, Predrag & Mahmud, Nesredin & Marinescu, Raluca & Seceleanu, Cristina & Ljungkrantz, Oscar & Lönn, Henrik. (2016). Simulink to UPPAAL Statistical Model Checker: Analyzing Automotive Industrial Systems. 9995. 748-756. 10.1007/978-3-319-48989-6_46.

[20] Hsiung, Pao-Ann & Chen, Yean-Ru & Lin, Yen-Hung. (2007). Model Checking Safety-Critical Systems Using Safecharts. Computers, IEEE Transactions on. 56. 692-705. 10.1109/TC.2007.1021.

[21] Dammag H., Nissanke N. (2003) A Mathematical Framework for Safecharts. In: Dong J.S., Woodcock J. (eds) Formal Methods and Software Engineering. ICFEM 2003. Lecture Notes in Computer Science, vol 2885. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-39893-6_35

[22] Jiang, Yu & Yang, Yixiao & Liu, Han & Kong, Hui & Gu, Ming & Sun, Jiaguang & Sha, Lui. (2016) . From Stateflow Simulation to Verified Implementation: A Verification Approach and A Real-Time Train Controller Design. 1-11. 10.1109/RTAS.2016.7461337.

[23] Gromov M., El-Fakih K., Shabaldina N., Yevtushenko N. (2009) Distinguing Non-deterministic Timed Finite State Machines. In: Lee D., Lopes A., Poetzsch-Heffter A. (eds) Formal Techniques for Distributed Systems. FMOODS 2009, FORTE 2009. Lecture Notes in Computer Science, vol 5522. Springer, Berlin, Heidelberg.

[24] Hamon, G., & Rushby, J. (2004, March). An operational semantics for Stateflow. In International Conference on Fundamental Approaches to Software Engineering (pp. 229-243). Springer, Berlin, Heidelberg.

[25] The Mathworks: Stateflow and Stateflow Coder, User’s Guide. Release 13sp1 edn. (2003).

[26] UPPAAL, uppaal.org/documentation/.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Mariam Mikava

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis, supervised by Tonu Naks
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

10.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author (s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.