IMPROVING THE MODEL CHECKING ACTIVITY USING

H-UPPAAL

A NEW INTEGRATED DEVELOPMENT ENVIRONMENT FOR MODEL CHECKING

BY NIKLAS KIRK MOURITZSEN & RASMUS HOLM JENSEN

AALBORG UNIVERSITET

Contents

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
Software
Selma Lagerlöfsvej 300
9220 Aalborg
`www.cs.aau.dk`

Title
H-UPPAAL

Theme
Model Checking Tools

Project period
P10, spring semester 2017

Project group
DES106F17

Authors
Niklas Kirk Mouritzsen
nmouri12@student.aau.dk

Rasmus Holm Jensen
rhje12@student.aau.dk

Project supervisors
Ulrik Nyman
ulrik@cs.aau.dk

Dimitrios Raptis
raptis@cs.aau.dk

Number of Pages: 74
Number of Appendix Pages: 26
Published: 2017-06-02

Abstract

Model checking is a popular research area; however, most people are concerned with *efficiency* and *speed* regarding verification time and computer resources. In recent years, limited work has been put into optimizing the entire model checking activity. Due to this, we introduced the H-UPPAAL tool, which includes modern concepts such as hierarchies and integrated development features. Throughout the project, we raise and answer the question:

How can the introduction of hierarchies and integrated development features improve the workflow in the model checking activity?

To answer this question, we evaluate the tool using different techniques, such as expert insight and performance evaluation where we compare the tool to the mature model checking tool, UPPAAL. These evaluations indicate that the newly developed tool contains concepts that create value in the model checking activity.

1 Preface

This report is written by project group DES106F17 from the Software Master's program at Aalborg University. This report concludes the semester spanning from January 2017 to June 2017 and expands on the semester project given in Mourtizsen and Jensen, 2016 (same authors). The overall theme of this project is verification, more specifically model checking tools. In this project explorative work have been made on how hierarchies can benefit the world of model checking. During this project, there have been developed a tool that includes such hierarchies while being inspired by modern integrated development environments. The following preface is copied from Mourtizsen and Jensen, 2016 and then expanded on to explain any new concepts.

1.1 Reading Guide

Throughout the report, personal pronouns refer to the authors of the report. The content of this report is written chronologically and should be read as such. The report uses the Chicago style method of citations for instance [Gerd Behrmann, 2006]. A lexicographically sorted list of references can be found in the Bibliography on Page 78.

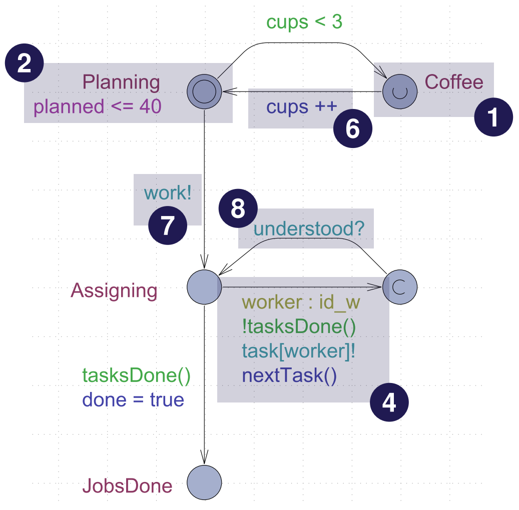
1.2 Terminology

This section will cover the terminology used throughout the report. We will describe the terms used in the UPPAAL tool, and introduce some terms used to describe the newly developed H-UPPAAL tool.

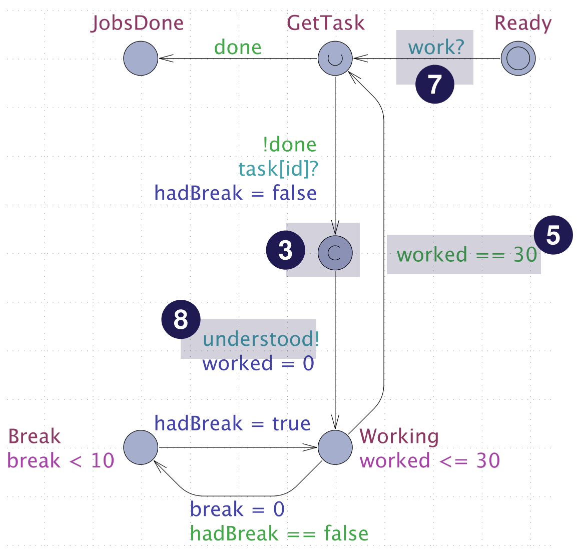
1.2.1 The UPPAAL Tool

The terminology of UPPAAL will be explained through the use of Example 1. Figure 1.1 shows the models for the implemented system, while Figure 1.2 shows the declarations and parameters used for the different templates.

Example 1 – Boss/Worker relation. We want to model a workspace consisting of 1 boss, and an arbitrary amount of workers. The boss has an arbitrary amount of tasks that he uses his workers to complete. The tasks are unordered, and must each take no longer than 30 minutes to complete. Workers may take a break when working on a task, but the break must not exceed 10 minutes. When the workday starts, the boss is allowed a period of 40 minutes, where he can plan the day. In this period he may drink up to 3 cups of coffee. After planning the day, the boss must begin assigning work to individual workers. The workday concludes when all tasks are finished.



(a) Boss template.



(b) Worker template.

Figure 1.1: UPPAAL template of a boss assigning tasks to multiple workers.

```
const int N = 120; // # tasks
const int W = 3; // # workers

typedef int[0,N-1] id_t;
typedef int[0,W-1] id_w;

bool done;

broadcast chan work;
chan task[W], understood;
```

Listing (1.1) Global declarations.

```
clock break, worked;
bool hadBreak;
```

Listing (1.2) Worker declarations.

```
const id_w id
```

Listing (1.3) Worker parameters.

```
clock planned;
int[0,3] cups = 0;
id_t nextUp = 0;

// Function for finding the next task
void nextTask() {
    if(nextUp < N - 1) {
        nextUp = nextUp + 1;
    }
}

// Function telling if the job is done
// Prevents out of bounds error
bool tasksDone() {
    return nextUp == N - 1;
}
```

Listing (1.4) Boss declarations.

```
system Boss, Worker;
```

Listing (1.5) System declarations.

Figure 1.2: UPPAAL declarations and parameters for the boss-worker example.

Template
A structure used to define an abstract timed automaton which later can be instantiated. Works similar to a class in OOP. A timed automaton is simply an instance of a template, meaning that templates are not singleton where the automaton is.

Process
In UPPAAL an instantiated automaton of a **template** is called a process. When writing **queries**, it is the name of the process that is used which sometimes differ from the name of its **template**.

Model (System)
A model, also known as **System**, is the description of the entire network of timed automata. The system declarations in UPPAAL is where we declare which and how templates are instantiated.

In the model for Example 1, there are two templates. These templates are put into the model in parallel by the system declaration (Code Snippet 1.5). Note that we only have one worker template, but the model consists of many worker automata. The worker has the parameter `const id_w id`, which creates workers according to the size of the type `id_w`. This is defined using the constant `W` which is set to 3 as seen in Line 2 in Code Snippet 1.1.

Global State
Describes the state of all **locations** of the timed automata, alongside variables and **clocks**. We can talk about **symbolic states**, where values of clocks can be bounds and not actual values. For instance, a clock might have the bound $c > 32$. Analyzing this symbolic state space is what allows UPPAAL to work with an infinite state space.

Transition
Describes how the entirety of the model goes from one state to the next. This could be by automata in the network taking edges or by delaying time on the clocks.

Query
A query is an inquiry of properties of a given model. The key purpose of verification is that we can query the model of a system resulting in a boolean answer. Often used to verify if the model can reach a specific **state**, or check when specific **edges** are available. Some queries are able to return **traces**, which can be used as proofs for the property. Queries are written in syntax based on TCTL (timed computation tree logic).

Trace
A series of transitions leading up to a state where a given property holds, or an example of a state where the property does not hold.

One example of a query could be `E<> Boss.cups == 3 && Boss.JobsDone` which is the same as asking *"Is it possible for the boss to drink three cups of coffee and still get the job done?"*. If UPPAAL returns `false`, no trace is returned, since it is impossible to reach this particular state. However, if the property holds, a trace will be returned, showing an example of how this is possible. A similar example could be `A<> Boss.JobsDone` which translates to *"Is it guaranteed that the boss eventually gets the job done?"*. In this case, if UPPAAL returns `false`, a trace could be returned, showing a single example of when this property does not hold, e.g. an example of a deadlock before the boss reaches `JobsDone`.

Clock
A special variable that keeps track of time. All clocks progress linearly. Can be reset by the **update**-statement on an **edge**.

In the boss-worker example, there are three clocks defined (`break`, `worked`, `planned`) as seen in Line 1 in Code Snippet 1.2 and Line 1 in Code Snippet 1.4. These clocks track time of how long the worker has been working, the durations of his break, and how long the boss has been planning.

Location
Is the state in which an automaton is at any given time. A location can be annotated with an **invariant**.

> **Invariant**
> A boolean expression that must evaluate to `true` to be in, or enter, this **location**.

Furthermore, a location can be either **urgent** or **committed**:

> **Urgent**
> Denotes that time is not allowed to pass when the system is in this **location**.

> **Committed**
> If the automaton is in this **location**, time cannot pass, and the system is forced to take a transition that includes at least one edge from a committed location.

The visual representation of the `Coffee` location can be seen at 1 in Figure 1.1. This location is **urgent** but has no **invariant**. The location `Planning` at 2 does have an **invariant** restricting the boss from planning for more than 40 minutes. An example of a **committed** location can be seen at 3 .

Edge
An edge is a passage from one **location** to another and can be annotated with four different properties:

> **Select**
> A shorthand for creating identical edges over a specified range. Used to model a non-deterministic switch case.

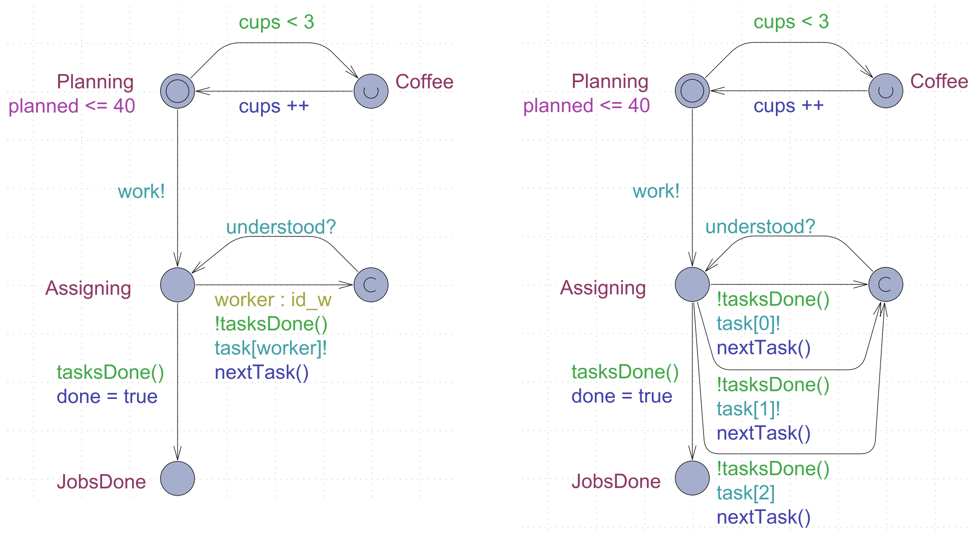
> **Guard**
> A boolean expression, often based on variables and clocks, which must evaluate to true for the **edge** to be active.

> **Synchronization**
> A way for automata in the network to communicate (either handshake or broadcast). A synchronization is mediated using **channels**.

> **Update**
> One or multiple statements that update variables and clocks when the edge is taken.

As described, an edge is a way of moving from one location to another. An example can be seen at 4 . This particular edge has all the possible properties set.

The `worker : id_w` is the **select** statement. As mentioned, this is a shorthand notation for creating multiple edges. Figure 1.3 illustrates how the boss template (Figure 1.3a) could be expanded if we had three workers (Figure 1.3b).



(a) Boss template. (b) Expanded boss template.

Figure 1.3: Functionality of select statements illustrated.

The edge at 4 also has a **guard**, namely `!tasksDone()`. The `taskDone()`-function returns `true` if the boss is done assigning tasks, otherwise `false`. The function is defined in Lines 14-16 in Code Snippet 1.4. This guard will prevent the boss from assigning more tasks than he has available. Another example of a guard can be seen at 5 . This guard ensures, in combination with the invariant of the `Working` location, that each task takes precisely 30 minutes to execute.

The **synchronization** of the edge at 4 indicates that when the edge is taken, synchronization on the **channel** `task[worker]`, should occur. We use the `worker` variable from the **select** statement to indicate which of the workers should accept the synchronization. For instance, when communicating with worker #2, we synchronize on the **channel** `task[2]`.

The last property on the edge is **update**, which in this case will call the function `nextTask()`, declared in Lines 6-10 in Code Snippet 1.4. This function will increment the `nextUp` variable. Another example of an update can be seen at 6 , where we simply increment the amount of cups the boss has consumed by 1, using `cups++`.

Channel
A media for synchronization, which is of the type **handshake** or **broadcast**. A channel can also be **urgent**.

Handshake
If a channel is of the handshake type, only two automata can synchronize over this channel at the time. The sender is marked with `!`, while the receiver is marked with `?`.

Broadcast
If a channel is of the broadcast type, one sender (`!`) on this channel may synchronize with zero to many receivers (`?`).

Urgent
Denotes that no time may pass if a synchronization is possible on this channel.

In the boss-worker example we have five ($2 + W$) channels: `understood`, `work`, and `task[W]`) as seen in Lines 9-10 in Code Snippet 1.1. The boss broadcasts on the `work` channel to tell the workers to get going, seen at 7. The `task[W]` is, as described previously, an array of channels, one for each worker in the system. Furthermore, `understood` is a channel used by the workers to acknowledge the task they have been assigned. This synchronization can be seen at 8.

1.2.2 The H-UPPAAL Tool

The terminology used in H-UPPAAL is similar to the one already covered; however, it does introduce the following terms.

Component
A component is a collection of locations, edges, and subprocedures. A component always has exactly one initial and one final location.

Subcomponent
An instance of a **component** declared inside another **component** with a specific instance name.

Subprocedure
A collection of **subcomponents** that runs in parallel, started by a **fork** and concluded by a **join**.

1.3 Notation

The following notation is from Mourtizsen and Jensen, 2016 and is reproduced here for the readers' convenience since it will be used in the report.

Notation 1.1 $c_1 : C$ denotes that c_1 is a subcomponent of component C. Likewise, we have that $\{c_1, c_2, c_3\} : C$ denotes that all subcomponents in the set are instances of component C.

We might have two components A and B. In B we have a single instance of A, named a_1. We know that a_1 is an instance of A, so we write $a_1 : A$.

Notation 1.2 Sc is a function that, given a component, C, will return the set of all subcomponents instantiated in that component.

We might have a component, A, with subcomponents s_1, s_2, s_3, and, s_4. In this example we have that $Sc(A) = \{s_1, s_2, s_3, s_4\}$. If we have a component B with no subcomponents, we have that $Sc(B) = \emptyset$.

Notation 1.3 Ic is a function that, given a component, C, will return all instances of this component. We now have that $\forall c_i \in Ic(C) \mid c_i : C$.

If we have three components, A, B and, C, where $Sc(A) = \{b_1 : B, c_1 : C\}$, $Sc(B) = \{a_1 : A, c_2 : C\}$, and, $Sc(C) = \{a_2 : A, b_2 : B\}$, we then have that $Ic(A) = \{a_1, a_2\}$.

Notation 1.4 Loc is a function that, given a component, C, will return all locations in that component.

We might have a component, C, with an initial location, l_0, and a final location l_1. In this case, $Loc(C) = \{l_0, l_1\}$.

2 Summary

Model checking is a technique that allows for designing information systems while having a guarantee that this design has certain properties. Tools like UPPAAL have an expressive modeling language that allows for modeling of rather complex systems while having an efficient verification engine that can be utilized to ensure said properties. Detailed models have a tendency to become so complex that they become hard to comprehend.

- Mourtizsen and Jensen, 2016

This project focuses on the continued work on the formal model checking and verification tool H-UPPAAL, which is based on the UPPAAL tool developed primarily at Aalborg University. This project is focused on evaluating and improving the tool, designed, developed and implemented mostly in Mourtizsen and Jensen, 2016, where effort has been made in improving the model checking activity by introducing hierarchies and integrated development environment features. This is done to have a greater modularity in models, better encapsulation, and, in general, a higher abstraction level.

Furthermore, this project also focuses on utilizing the UPPAAL backend in new ways to give modelers new tools and techniques to use while implementing models. One of the concepts introduced is the *automatic reachability analysis*, which allows modelers to visually see reachable parts of the implemented models in real time during the development. The idea is here to notify users whenever potential bugs, or unwanted behavior, is introduced. This feature can be seen as a form of automation of techniques modelers would usually use.

Throughout the project, the H-UPPAAL tool has gone through different evaluations. These evaluations are done to compare the H-UPPAAL tool, and in extension, the new concepts introduced with the tool, to the mature UPPAAL tool. To do this, an *expert user evaluation* were conducted. This evaluation investigated whether expert users of the UPPAAL can understand and use the new concepts introduced. After this, a more in-dept *performance test* was conducted, where the tools were directly compared on participants ability to solve exercises. Considering the amount of work that has been put into H-UPPAAL, the results from both evaluations are promising, indicating that further work within the area should be done might be very interesting.

The report concludes with insight into how the H-UPPAAL tool could be improved in different areas to better compete with mature model checking tools such as UPPAAL. In general, we believe that the tool shows enormous potential and we believe that given more work, the tool would be able to compete with well established model checking and verification tools.

I The Model Checking Activity and H-UPPAAL

3 Problem

Model checking is mostly concerned with validation or verification of concurrent programs, data protocols, and reactive systems. This technique allows developers and researchers to design software while having a guarantee that the design has certain qualities.

- Mourtizsen and Jensen, 2016

There have been done plenty of research in the domain of automated verification and model checking. However, most research focuses on optimizing results from verification queries given a model. Based on experience and conversation with various experts in the realm of model checking we have found that the workload of performing the model checking activity lies in modeling the system rather than verifying its properties. Figure 3.1 illustrates the workflow when using a model checking tool. Research that is concerned with model checking tends to focus around the backend of such tools, where effort is laid into representing and exploring state space efficiently. However, what value does an efficient backend provide if the user spends the majority of the time on generating the model that describes the behavior?

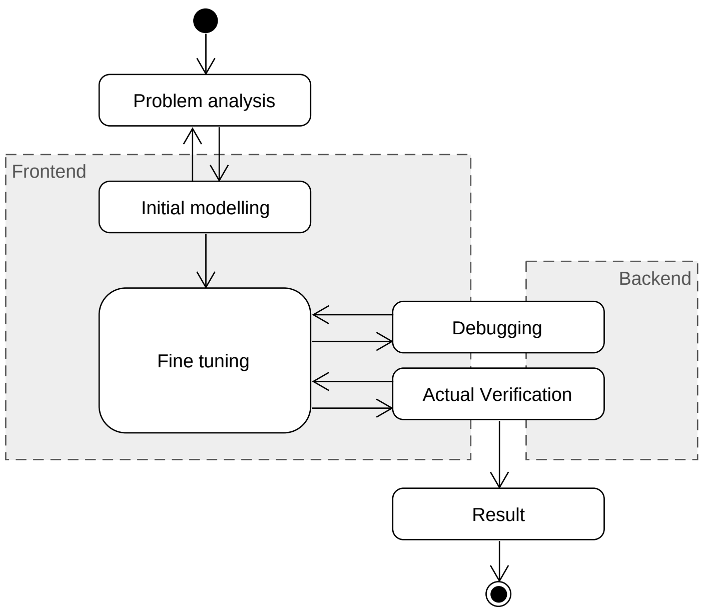


Figure 3.1: Flow of the model checking activity.

This question has motivated us to look at how we can enhance the bridge from user to an underlying verification engine. We would like to explore how we can use hierarchies and features that are inspired by modern programming IDEs. We have been implementing a tool called H-UPPAAL that attempts to do exactly this [Mourtizsen and Jensen, 2016]. This tool extends the verification engine of UPPAAL, which is a model checking tool that models systems using networks of timed automata [Larsen et al., 1997]. H-UPPAAL includes hierarchies and comes with features inspired by integrated development environments. Via this new tool, we would like to see if these concepts can provide value (increasing performance) during the model checking activity. In general, we would like to investigate:

> How can the introduction of hierarchies and integrated development features improve the workflow in the model checking activity?

We deem that by exposing users with varying knowledge of formal verification to H-UPPAAL, we can find evidence for and against the concepts it introduces. The goal of this semester is to finalize the tool by polishing the hierarchical model alongside adding a bit more IDE flavor. After this, the investigation commences, where hierarchies and IDE features are put to the test.

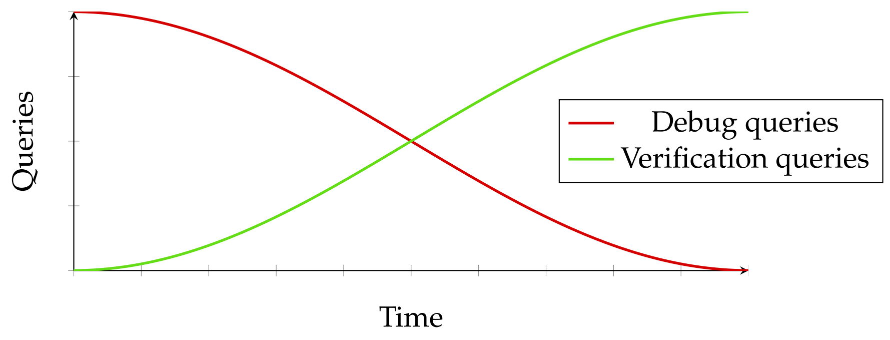


Figure 3.2: Development of different query-types over time.

Looking at Figure 3.1 the state users tend to stay in the longest is *Fine tuning*. Based on our own experience and via conversations with users of model checking tools, users initially want to ensure that the model in being developed represents the behavior of the given system in question. For this reason, users tend to manually utilize the verification engine to debug the model to see if the model represents the intended behavior. As time progresses the users become more and more confident in the model and start to write actual verification queries. Figure 3.2 illustrates how users tend to use these two types of queries over time. They start out by writing queries purely to ensure that the behavior of the model is as intended, and ends up with the actual verification queries that ensure that the designed model have certain qualities. Note that from the backend's point of view we do not distinguish between these two types of queries. For this reason, we will look at how one could introduce features that facilitate debugging queries.

In the fall of 2016 [Mourtizsen and Jensen, 2016] it was stated that formal syntax and semantics for the tool was important to explain and prove that the new hierarchical language of H-UPPAAL can be translated to the language of UPPAAL. However, we deem that a complete formalism for the tool is secondary to investigating the impact of the concepts. If hierarchies and IDE features show to add no value to the model checking activity, work on such a formalism is drawn useless, considering that such drafts and suggestions to this type of formalisms have already been published, e.g. [David and Möller, 2001].

4 Related Work

Some work has already been made in designing and formulating new languages and tools in the domain of model checking. This chapter strives to credit some related work that has been made both regarding formulating a modeling language and how the translation of these languages and test of such tools can be done.

4.1 UPPAAL PORT

There exists many version of the UPPAAL tool out there, which all make use of the power of the verification backend for some specialized purpose. One of such tools is UPPAAL PORT [Håkansson et al., 2008]. This branch of the UPPAAL family is concerned with incorporating *Partial Order Reduction Techniques* (PORT), which exploits the commutativity property of a subset of the state space where the ordering of parallel transitions results in the same state which is used to reduce the state space exploration. In respect to the developed H-UPPAAL tool, this tool includes structures from the SaveCCM language. This language, as H-UPPAAL, has a hierarchical nature, consisting of components, both simple ones, and composite ones. Besides this, details for a collection of components can be abstracted over using assemblies, switches, and ports. In Figure 4.1 we see a model of an adaptive cruise controller (ACC) [Åkerholm et al., 2007]. We see that some of the details of how this cruise controller behaves are encapsulated in components and assemblies, like how the break works (*Brake Controller*) or how the system recognizes objects on the road (*Object Recognition*).

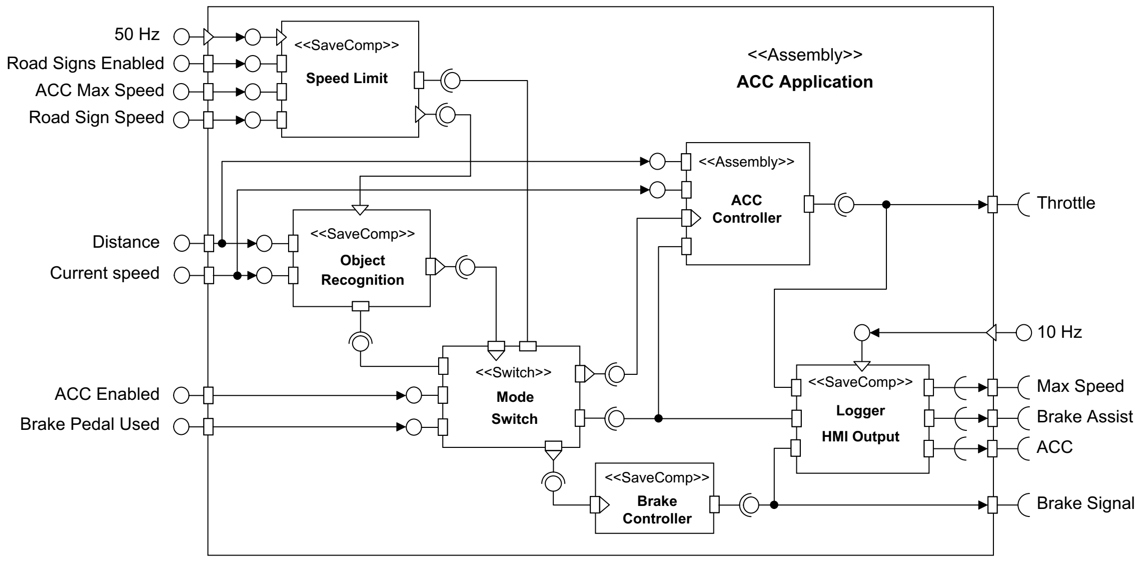


Figure 4.1: Adaptive cruise controller modeled in SaveCCM.

UPPAAL PORT is in particular interesting since it already uses UPPAAL in combination with a hierarchical structure. However, it is more of a merge between two languages rather than a hierarchical version of UPPAAL. UPPAAL PORT combines the compositional structure from the SaveCCM language and the behavior of timed automata from UPPAAL to model a system. You describe the communication between components and, in general, the composition of the system using SaveCCM components, while using the timed automata to describe the behavior of leafs in the tree also known as simple components. In comparison, H-UPPAAL uses a tree of timed automata and use them to describe behavior in all levels of components in the tree. Meaning that we can use constructs from timed automata in the top most level as well as the lowest.

4.2 Colored Petri Nets

In the domain of formal verification, networks of timed automata are not the only formalism used to describe concurrent and reactive systems. In this field of research, another popular formalism is Petri nets. These nets consist of *places*, *transitions*, and *arcs* between the two. *Tokens* can be inside places while a transition describes how tokens relay from one place to the next. An extension of this formalism is *timed* and *colored* Petri nets, which enables the "coloring" of tokens, essentially allowing for the annotation of additional information on a token besides where it is placed in the net. These colors can be used to easier describe a more complex state space by having integers, strings or even time as properties on a token. Furthermore, formalisms including hierarchies in colored Petri nets have been developed. In Figure 4.2 a model of an assembly line is modeled using such a formalism [Huber et al., 1989].

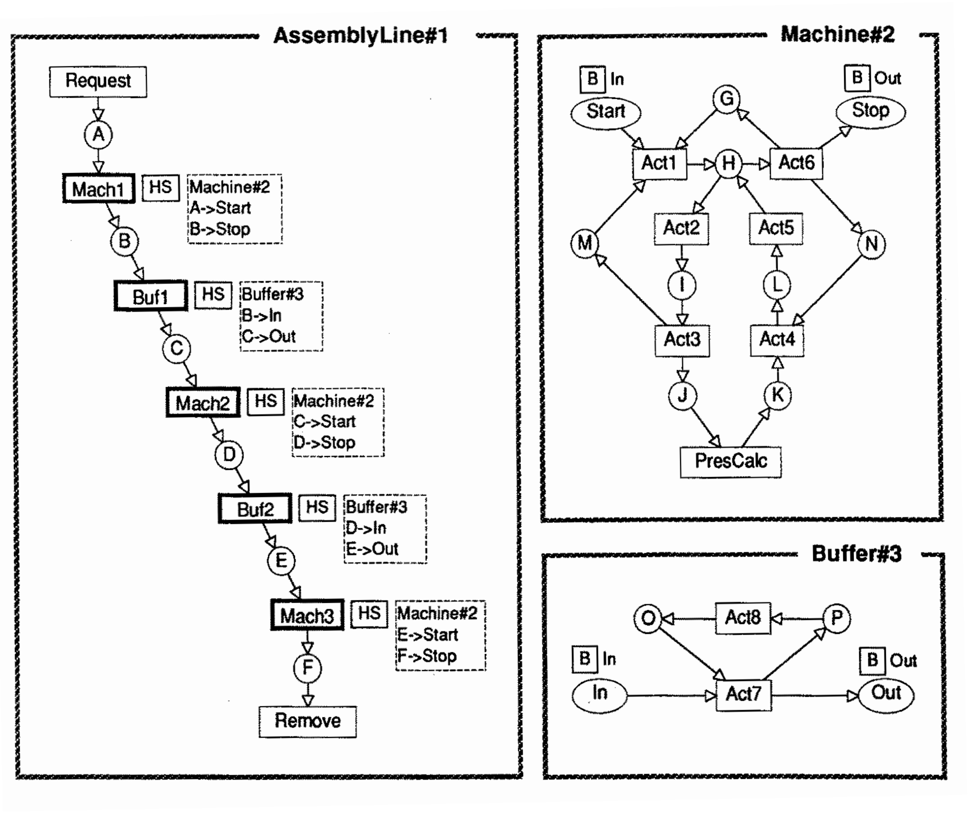


Figure 4.2: Assembly line modeled using hierarchical colored Petri net [Huber et al., 1989].

The figure shows that we simply encapsulate some of the behavior as a transition with a more complex behavior called a *subpage*. Note that multiple instances of the same subpage is allowed, as *Mach1*, *Mach2* and *Mach3* are all instances of the *Machine#2* subpage. This is rather similar to the way components and subcomponents works in H-UPPAAL, where we can instantiate multiple subcomponents of the same component.

4.2.1 CPN Tools

A tool that implements both a hierarchical and colored Petri net is CPN Tools[1]. This tool allows for exploration of state space and verification of properties of said state space similar to tools of the UPPAAL family. CPN Tools is broadly used for scientific purposes. The paper that introduces this tool has more than a thousand citations [Jensen et al., 2007]. A significant amount of these papers is case studies, where the tool have been used in an industrial setting, which indicates the tool has its impact in the industry as well.

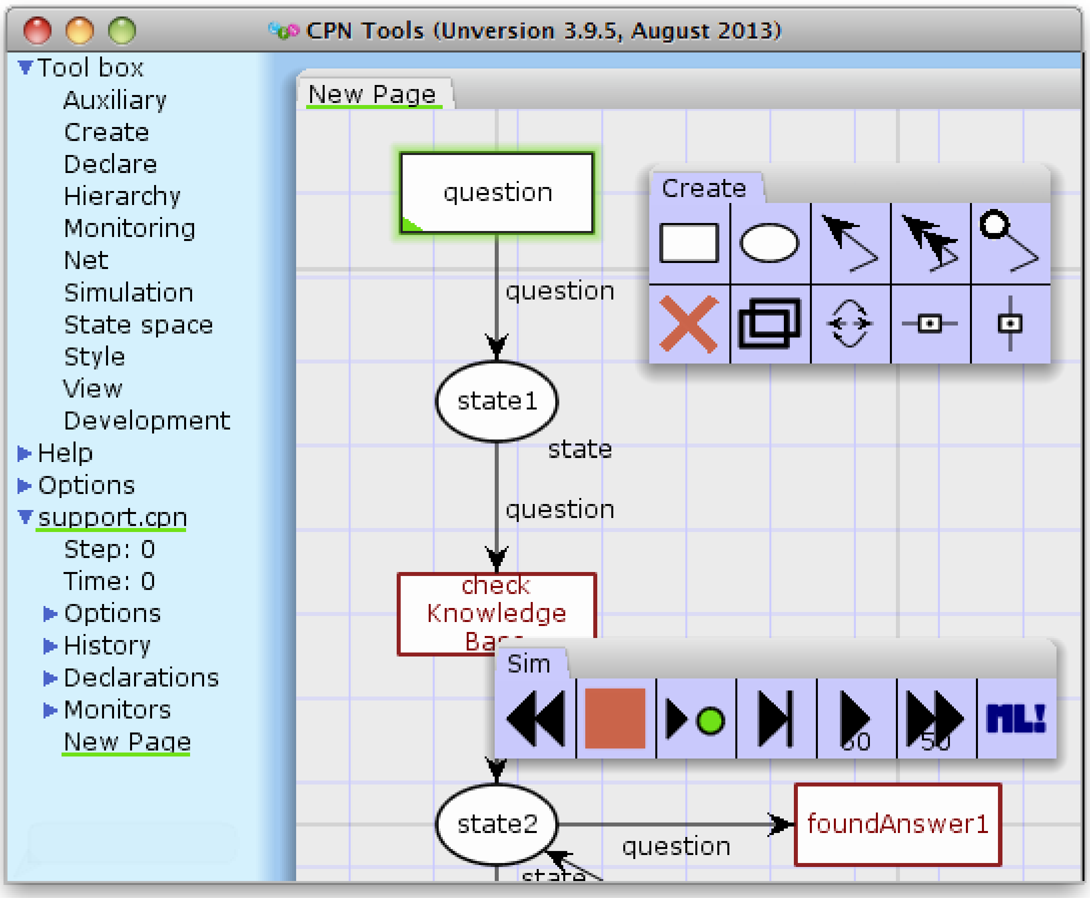


Figure 4.3: Screenshot of CPN Tools.

Figure 4.3 shows CPN Tools and illustrates that the tool is composed of a side panel (left-hand side) and the canvas (right-hand side). The layout of the tool is to some degree similar to the one of H-UPPAAL where we have one part of the model open at the time, and all interaction with this model is accessible from the same view. We have not used CPN Tools for inspiration, but we agree that having all features that interact with a model should be present when modeling opposed to the more divided user interface of UPPAAL.

[1]`http://cpntools.org/`

4.3 Previous work on H-UPPAAL

This report is the continued work on the tool H-UPPAAL which can be seen in Figure 4.4. This work was centered around designing and implementing a tool that introduces hierarchies into networks of timed automata [Mourtizsen and Jensen, 2016]. The tool was manifested around issues with the current version of UPPAAL, where the complexity of performing model checking is attempted reduced by adding a formalism that allows for abstraction and a tool that is inspired by modern programming IDEs. The formalism that ended up in H-UPPAAL is strongly inspired by *From* HUPPAAL *to* UPPAAL*: A Translation from Hierarchical Timed Automata to Flat Timed Automata* [David and Möller, 2001], where the authors presents a hierarchical formalism that can be translated to UPPAAL. Note that H-UPPAAL do not implement this formalism but a simpler prototyped version.

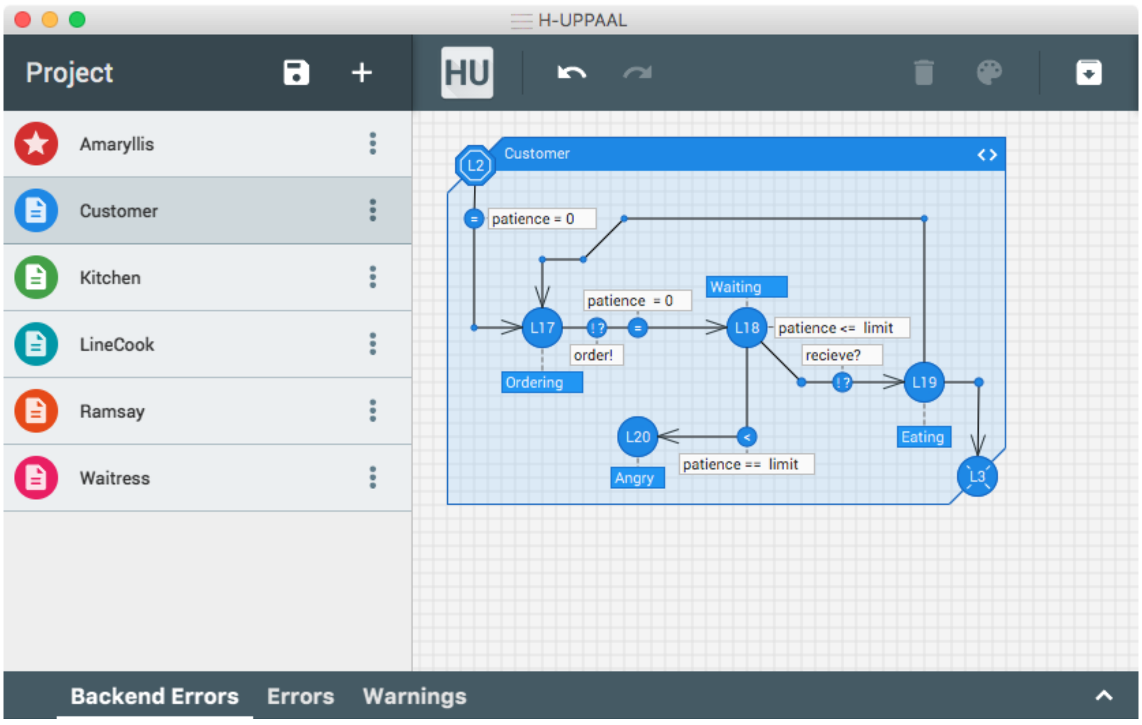


Figure 4.4: The H-UPPAAL Tool.

4.4 Usability Evaluation and Analysis

The theme of the project is centered around improving the model checking activity rather than improving the underlying verification. For this reason, we need to be able to reason about how new ideas and concepts will affect this activity. To do so, there have been conducted two different usability evaluations, a rather qualitative study, to investigate preferences of expert users (Chapter 6) and a quantitative study to measure performance of our newly developed tool H-UPPAAL compared to an existing mature model checking tool UPPAAL (Chapter 9). This performance test is greatly inspired by "Does Size Matter?: Investigating the Impact of Mobile Phone Screen Size on Users' Perceived Usability, Effectiveness and Efficiency." [Raptis et al., 2013]. This study shows a significant effect of screen size on *efficiency*, i.e a larger mobile screen decrease the time it takes for a user to complete a task. However, our inspiration is not found in mobile devices or screen size, but have strong relations to how the study was conducted and the methodology of the statistical analysis.

In general the idea of evaluating and designing systems for experts is obviously not a novice idea. As more and more people started using IT systems, designing and evaluation usability for such systems have become more and more important. In articles like "A methodology for quantifying expert system usability" [Mitta, 1991], the usability and how to quantify it is discussed. And there have in general been done a lot of evaluations on expert systems from all kinds of areas, like systems designed for medical experts "Evaluating Medical Expert Systems: What To Test, And How ?" [Wyatt and Spiegelhalter, 1991], where they discuss how we evaluate systems for expert in the medical domain.

When browsing for scientific papers regarding such evaluations in the domain of model checking, one is often left with no results, indicating that there have been done limited to no research in this particular area. Papers regarding new tools and techniques for model checking claim that they desire a high usability of their system, but not much effort have been put into actually evaluating them. In other words, evaluation of expert systems is for sure not a novice idea, but such evaluation focused on model checking might be.

5 The H-UPPAAL Manifesto

To better help to steer the project and evaluate ideas, a manifesto consisting of 6 principles were put together in Mourtizsen and Jensen, 2016. The principles of this manifesto are reproduced here for the readers' convenience.

Principle 1 – Backward Compatible. Even though H-UPPAAL is a new tool, it should still be clear that it is based on UPPAAL, and should therefore use the same concepts.

Principle 2 – Integrated Development Environment. H-UPPAAL should facilitate quick and easy development of models and verification of these.

Principle 3 – Information Hiding. Unnecessary information should be collapsible to increase overview.

Principle 4 – Identity and Relation. Different visual elements, such as locations and components, should have identity through name and color to increase familiarity and allow for easy communication and collaboration. Furthermore, it should be clear which properties are in relation to which elements.

Principle 5 – Printable. You should be able to export a model so that it can be printed without loosing any information.

Principle 6 – Objects Require Space. Objects take up space on the screen, and cannot overlap with other objects. An exception for this is edges, that may overlap, since this restriction would otherwise make it too hard, or even impossible, to model certain systems.

6 Expert User Evaluation

We would like to investigate how users of UPPAAL can transition into our newly developed H-UPPAAL tool, i.e. adhering to Principle 1 (Backward Compatible). To do this, a usability evaluation has been conducted. This test should indicate where H-UPPAAL needs polishing to accommodate familiarity and preferences of UPPAAL experts. Besides a smoother transition to our tool, we would like to get useful feedback from these experts on some of the design decision we have considered and implemented. In general, the purpose this test is to figure out which parts of the tool need to be improved and gain insight on expert users' preferences, especially regarding the visual representation of models in H-UPPAAL.

This test is based on the think-aloud protocol [Nielsen, 1993], where participants are given tasks and asked to think out loud while they are performing them. The think-aloud protocol serves a medium to the mind of the participants, meaning that you can discover what users really think about the system while they are using it. The verbalization of thoughts during a test will assist in catching misunderstanding and easier point out a single part of a system which causes confusion or misconception. In general, this allows us to understand the users' thought process better when working with H-UPPAAL and in extend model checking.

This evaluation was conducted using six participants who all have prior experience with the UPPAAL tool. We label these participants with letters **A** - **F**. For this study we handpicked students that we saw as good representatives of users that should be confident in tools like UPPAAL and that have knowledge regarding model checking. Before starting the actual evaluation, we asked the participants a few questions regarding their experience with UPPAAL and their confidence in this.

6.1 Tasks

The test is divided into four tasks: *Draw a Model*, *Expand a Model*, *Explain a Model*, and *Rank the Different Proposals*. To ensure that the order of the tasks does not affect the outcome of the test (the carry-over effect), the order is randomly permuted between participants. However the *Draw a Model* is always the first task that is used to introduce the participants to our project and the H-UPPAAL tool. The following sections will outline these tasks and explain their purpose. The task sheets can be found in Appendix 14.

6.1.1 Draw a Model

The participant is asked to reproduce the model of a vending machine using H-UPPAAL. The goal here is to see if the participant has a smooth transition from drawing models in UPPAAL to drawing models using our tool. Ideally, each participant should feel no difference, since the tools are similar. However, H-UPPAAL has some changes with respect to the modeling language. For this reason, it is desired to investigate if this change introduces too many misconceptions.

6.1.2 Expand a Model

We would like to see if the participant can understand the hierarchies introduced in H-UPPAAL. In this task, we provide a "Computer Scientist", and a "University"-component, which have a hierarchical structure. After a brief introduction of theses components, the participant is tasked with expanding the model they created in the *Draw a Model*-task.

6.1.3 Explain a Model

A typical use case for tools like H-UPPAAL is scientific research. For this reason, it is preferable that communication of these models is as smooth as possible, e.g. the users' ability to read and understand models. We will in this task present the participant with a printed, hierarchical model developed in H-UPPAAL. The participant is then asked to explain the model in as great detail as possible. This will also, to some extend, test how well we adhere to Principle 5 (Printable). If the participant does not touch upon certain selected areas, we will ask the participant to explain them further.

6.1.4 Rank the different proposals

During the development of the tool, we made some choices on the design of the user interface. Up until now, we have used our intuition and the one of our supervisor, since all of us are to some degree expert user of UPPAAL. However, we would like to investigate if some of the alternative solutions for model representation is more or less preferable to other expert users. For this reason, in this task, we present the participants with design ideas in three different areas. Each area consists of a few ideas. The participant is then asked to rank the ideas on a scale from *Strongly Dislike* the idea to *Strongly Like* the idea. Theses ideas can be found in Appendix 15.

6.2 Results

Prior to the actual results, we will briefly discuss how the participants answered in respect to their own perception of expertise using the UPPAAL. This was done using a short questionnaire. Table 6.1 shows what and how the different participants answered. Participant **A** and **B** had not used the tool for some time but still deemed themselves as *Experienced* users. The last four participants had all used the tool within the last month and deemed that they were either *Experienced* or *Expert* users. This questionnaire simply supports our initial assumption regarding their capabilities, namely that they are familiar with the tool and have used it a lot.

When was the last time you used UPPAAL?	A	B	C	D	E	F
Never						
More than 12 months ago						
6 - 12 months ago	✓	✓				
1 - 6 months						
0 - 30 days ago			✓	✓	✓	✓

How frequent did you use UPPAAL the last year?	A	B	C	D	E	F
0 times						
1 - 10 times	✓	✓				
11 - 50 times				✓		
More than 50 times			✓		✓	✓

How would you describe your experience with UPPAAL?	A	B	C	D	E	F
Not experienced						
Novice						
Experienced	✓	✓			✓	
Expert			✓	✓		✓

Table 6.1: Questionnaire answers.

The purpose of this evaluation is to gather insight from users that are familiar with the UPPAAL tool. It is desired to find the most severe problems that our tool contains according to these experienced users. For this purpose, we have analyzed how the participants executed the *Draw a Model*, *Expand a Model* and *Explain a Model* tasks. This was done by identifying usability problems (14 in total), and for each participant classified them as either *Cosmetic*, *Serious* or *Critical* [Molich and Dumas, 2008]. Table 6.2 shows which participants encountered which problem and the level of severity. A problem is labeled 🟦 *Cosmetic* if they hesitated briefly. 🟨 *Serious* problems delayed the participant significantly and needed a few hints to proceed. The severest problems are 🟥 *Critical*, where participants either needed instructions or rather extensive hints to proceed with the task at hand. The following sections will go through the usability problems found and propose solutions to these, and describe the participants' preferences on the provided design ideas.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	🟥	🟥	🟨		🟨	🟦	🟦	🟦	🟦	🟦				🟦
B	🟥		🟥	🟥	🟦	🟦	🟦	🟦	🟦		🟦	🟦		🟦
C		🟨	🟦		🟥				🟦	🟦		🟦	🟦	🟦
D	🟦	🟨	🟥			🟦	🟦		🟦			🟨		
E			🟨	🟥	🟦				🟨			🟦	🟨	🟦
F	🟦		🟥							🟦		🟦		

Table 6.2: Categorization of usability problems found in the evaluation. Full description of the problems can be found in Section 6.3.

Note that a blank cell in Table 6.2 indicates that the participant did not get exposed to a situation where the problem might occur, or that the participant simply had no problem. For instance, in Usability Problem 4 (Drawing nails), only participant **B** and **E** had severe problems with doing exactly this. Some of the other participants did succeed in creating nails where others did not even try.

In general, the participants seemed rather pleased with the H-UPPAAL tool. Even though they encountered some problems during the test, most participants stated that they found the tool intuitive. Though this test was not intended to get the users perception of the tool, but rather to find usability problems on transitioning between the tools, we also get the intuition that expert users of UPPAAL seem interested in tools like H-UPPAAL.

6.3 Usability Problems

Ideally, we would like to address all the usability problems that we found during the evaluation. However, due to limited time, this remains as wishful thinking. Instead, we would like to try to solve the most serious problems. This section will go through the different problems identified, explaining how to solve them potentially and how extensive this would be.

> **Usability Problem 1 – Initial location.** The participants experienced trouble identifying the initial location in a component. Some participants thought that a new initial location had to be created, while others realized that the location was already created as a part of a component.

The initial location has undergone some changes from UPPAAL to H-UPPAAL. In the newly developed tool, this location is pinned to the top left corner of a component. Though many participants did immediately understand this change, some people required a subtle hint regarding the location shape to realize the change. It seems that people either *ignore* the locations in the components or that they are confused about the looks of them. This effectively means that we do no adhere to Principle 1 (Backward Compatible) since users are unable to map their knowledge about an initial location from UPPAAL to H-UPPAAL.

A solution for this problem could be to emphasize the initial location. This could be done by animating or briefly annotating them as seen in Figure 6.1. The idea is to briefly catch the users' attention to make them realize that initial location is built into the structure of a component, similar to how we gave them hints during the evaluation.

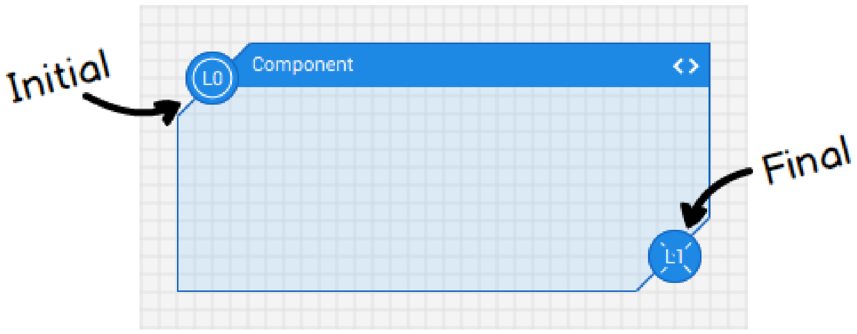


Figure 6.1: Sketch of hints on initial and final locations.

Usability Problem 2 – Parallel subcomponents. Participants could not start subcomponents in parallel since they were unfamiliar with the fork/join concepts.

Parallelization is not a new concept that we have introduced with H-UPPAAL. But with the new hierarchical structure, we have introduced constructs that explicitly models parallel running subcomponents. Since the users were not told that this new construct existed many of them did not know that they were needed. One way of potentially solving this issue could be to make the join and fork constructs look more like the ones used in UML activity diagram (black rectangles). This would, however, cause forks and join to be indistinguishable, potentially causes some other usability problems. Unless a better idea comes along, we deem this problem could be temporarily fixed by introducing the user to the concepts in a help menu or an initial tutorial of the tool.

Usability Problem 3 – Drawing edges. The participants could not draw edges between locations, subcomponents, joins and forks. Most participants looked for an edge creation mouse, as used in UPPAAL, while others tried different key modifiers or simply dragging the mouse from one element to another.

The only way to draw an edge in H-UPPAAL is by either `ALT`-click or middle-click an object that can start an edge. This problem was a simple realization that this special key combination inherited from UPPAAL was used by close to none of the participants, and was not the first way they tried to draw edges. Many of the participants searched the context menus for the option to draw an edge from various objects without luck. This indicates that the draw edge functionality should be enabled within these menus. This effectively also suggest that Principle 1 (Backward Compatible) is not as important as one might initially think. However, additional research has to be done to say anything conclusive.

Usability Problem 4 – Drawing nails. Participants were unable to create nails. Some participants tried double clicking the edge while others tried to hold down different key combinations.

Drawing nails while creating an edge was not problematic to any of the participants. However, some of the participants had severe trouble adding nails to an existing edge. Like Drawing edges (Usability Issue 3) this option was only enabled by `ALT`- or middle-click using the mouse. The addition of an “Add nail” option in the context menu of an edge would most likely decrease the severity of this problem like seen in the sketch in Figure 6.2.

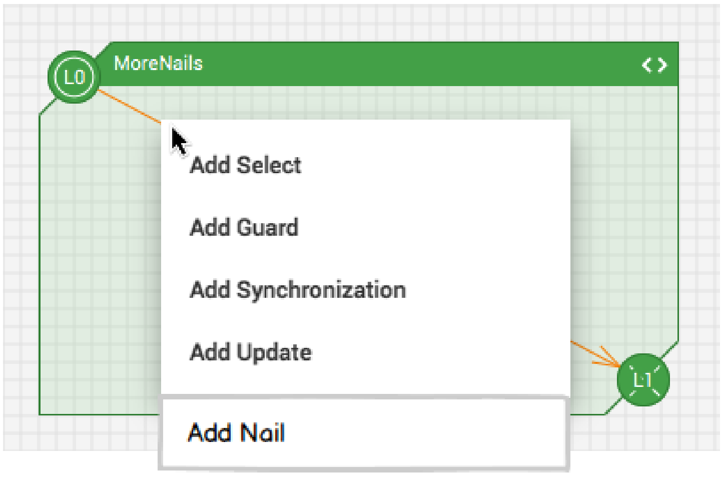


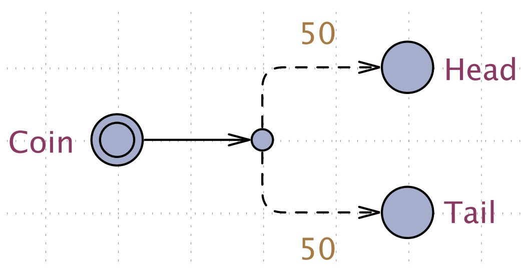
Figure 6.2: Sketch of added entry in the context menu.

Usability Problem 5 – Drawing locations. Participants experienced trouble with drawing locations. Some participants looked for a location creation mouse, while others tried different key combinations.

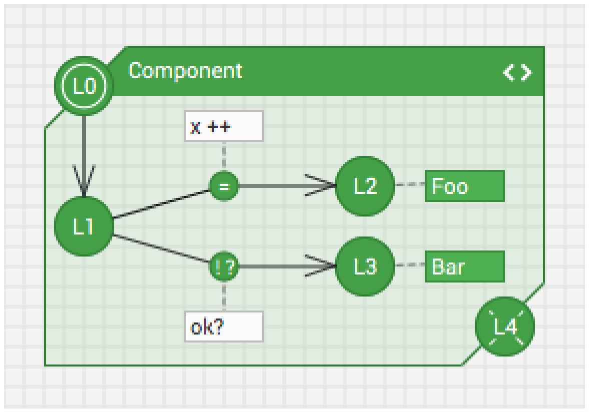
Many of the participants became familiar with the different mice types in UPPAAL (a "Location"-mouse and an "Edge"-mouse). Participants that had trouble figuring out how they could add their first location searched most of the UI to find these mice types without luck. However, many of them realized that right-clicking the canvas opened a context menu with an option for drawing one. Currently, we do not have any good ideas for how to solve this issue, but we believe that if the users got some initial help on how to add locations (either `ALT`-, middle-, right-clicking), this would temporarily fix the issue.

Usability Problem 6 – Reading edge properties. The participants had problems reading the different edge properties. Some participants thought that the properties were nails due to their behavior (when moving the edge) while others had trouble identifying whether they were working with a guard, synchronization, update or select.

In the design of H-UPPAAL, we wanted properties on edges to be closely bound to the edge that they belong to and for this reason, we introduced special nails that acted as anchors. The participants were briefly confused about this new way of annotating an edge since it looked like "mini-states" or probabilistic constructs of UPPAAL SMC (illustrated in Figure 6.3). Most participants agreed that this new way of adding properties was superior to the free floating colored labels used in UPPAAL. A solution that could reduce this confusion is to try to enhance the difference between nails and edge properties by always having nails be some distinct color e.g. black.



(a) Probabilistic state in UPPAAL SMC



(b) Properties on edges in H-UPPAAL

Figure 6.3: Confusion between edge properties and probabilistic states.

Usability Problem 7 – Adding edge properties. Participants were experiencing trouble adding properties to an edge. Most participants tried double-clicking the edge to open a dialog where they could input the different edge properties.

Annotation of a model with properties in UPPAAL is done in a dialog that is opened by double clicking an element. However, in H-UPPAAL these properties are added using context menus. All of the participants figured this rather quickly, but some of them continuously tried to double click an edge to annotate it. This effectively means that we are not adhering to Principle 1 (Backward Compatible). A solution to this problem would be to introduce the same behavior when double-clicking an edge.

Usability Problem 8 – Drawing subcomponents. Participants had problems when drawing a subcomponent. Some participants tried to drag components out from the file panel while others tried clicking elements in the file panel.

Some participants tried to drag in components from the file panel to add them to the canvas without luck. A solution to this problem would simply be to allow dragging of components into the canvas to add subcomponents. One might need some visual aid to see that dragging the components does something, e.g. a ghost-subcomponent following the mouse.

Usability Problem 9 – Differentiating clocks and data variables. The participant could not differentiate clocks and data variables when reading the model.

During the *Explain a Model*-task, participants had a hard time identifying the type of variables without having the declarations available (which variables are clocks and which are representing data). All participants eventually figured out the type of the different variables through deduction with thought processes like "`y` is incremented in this update, so `y` must be an integer". We believe that a solution where clocks are underlined would solve this problem like seen in Figure 6.4, where we have a clock `x` and an integer `y`. If introduced to the idea, it should be clear to see which variables are clocks and which are not.

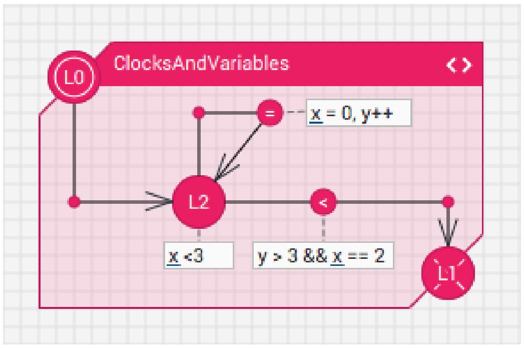


Figure 6.4: Sketch of underlined clocks.

Usability Problem 10 – Location urgency. The participant is experiencing trouble with differentiating the visual indications of location urgency. Some participants ignored the shape of the location, while others pondered what the shape might mean. Most of them did, however, realize that the shape represented urgency, but was still unable to differentiate *urgent* and *committed*.

In the evaluation, participants were exposed to Design Idea 2 (New Urgency with Octagon Shape) (with location identifiers in the center of locations). When doing the *Explain a Model*-task, participants had a hard time immediately differentiating urgent and committed locations. Some participants would look at a location and say that it is urgent, ignoring the bold border. Other participants would do the opposite, saying that the location is committed, even though the border of the locations was not bold. Looking at the results from the table ranking test, we decided to change the location urgency shapes to match the Design Idea 3 (New Urgency with Octagon and Square Shapes). This problem is related to Principle 1 (Backward Compatible), however, as mentioned in Mourtizsen and Jensen, 2016, we decided to change the way location urgency is displayed to utilize the central space for location identifiers.

Usability Problem 11 – Size. Participants had a hard time reading the models due to a small size.

Some participants felt that some UI elements were too small. This problem could be fixed by upping the scale of the smallest element. However, this might cause some users to feel like the size is too big. Instead, one could implement a zoom functionality, similar to the one found in UPPAAL. This would allow users to view the model at their desired size and the only thing we should consider is the relative size of the UI elements to each other, e.g. locations do not feel too big compared to edges, etc.

Usability Problem 12 – Describing fork and joins. The participant experienced trouble when describing the semantics of fork and joins, causing them to use made up words and strange sentences. However, most participants understood the concepts correctly but were just unable to explain them.

When asked to explain a model, participants needed a little time before realizing how the fork and join constructs worked. Some participants saw them as a special location and said that outgoing edges were identical to the normal *location to location* edges, i.e. when multiple edges are available, a non-deterministic choice has to be made (full exploration of the state-space). However, all participants eventually figured out the semantics of the constructs. Furthermore, some participants struggled a bit with describing the constructs since they did not know what to call them, causing them to refer to the fork/join constructs with words like "parallelization-thingies". We deem that it will challenging to make these construct self explanatory, and for this reason we believe that a short introduction view could decrease the severity. This view should explain what the constructs are called and briefly account for the semantics through a simple example.

> **Usability Problem 13 – Removing properties from an edge.** Participants were experiencing trouble when removing properties from an edge since a text field had focus, causing it to consume the keyboard event. This means that, visually, the edge properties looked selected, but since the text field had focus, the participants actually pressed delete inside the text field rather than deleting the property.

This problem is based on the fact that the tool currently has two focus points, meaning that the user can focus a text field while selecting elements in the model. However, when trying to delete the selected elements by pressing the `DELETE`-key, the text fields would react to the key-event. The event is then consumed and not propagated on to the selected-element-delete functionality. To solve this, one should not be able to have two focus points in the application. Instead, text fields should be unfocused when selecting elements, and vice versa. Effectively causing the tool only to have a single focus point.

> **Usability Problem 14 – Keeping context menus open.** The participant is experiencing trouble with keeping a context menu open, since they moved the cursor slightly outside the menu, effectively closing it right after opening it.

This problem occurs when participants open a context menu, for instance by right-clicking a component, and move the cursor slightly outside the context menu. This could be fixed by keeping the context menu open even after the cursor moves outside its boundaries. The context menu could then be closed by clicking some other element in the UI, or by pressing the `ESC`-key.

6.4 Design Idea Rankings

Another task the participants were asked to perform is the *Rank the Different Proposals*-task. The following sections will cover the results of this task. Each section visualizes the ranking of the design ideas found in Appendix 15, where the circle indicates the average ranking of an idea (identified with a number in the circle) and the lines represents the minimum- and maximum ranking. The left side of the scale represents *Strongly Dislike*, while the right side represents *Strongly Like*.

6.4.1 Location Urgency

When ranking the different ideas in regards to location urgency, participants tend to disagree, as seen in Figure 6.5, especially in regards to Design Idea 1 (Traditional UPPAAL Urgency). In general, participants think that the traditional representation is easy to read, i.e. *U* means *Urgent*, and *C* means *Committed*. However, they find it hard to remember which is the more strict one. Few participants think that both Design Idea 2 (New Urgency with Octagon Shape) and Design Idea 3 (New Urgency with Octagon and Square

Shapes) could help them better remember the difference, due to the more strict shape of the different locations. In general, participants said that the difference between urgent and committed in Design Idea 2 (New Urgency with Octagon Shape) is too subtle and can be hard to notice.

An important thing to remember here is that the reasoning behind changing the shape of the location is to free up space inside of the location so that this space can be utilized for displaying a unique identifier for the location [Mourtizsen and Jensen, 2016]. The participants did not know this when being tasked with ranking the different ideas. The reasoning behind this is that we wanted an unbiased opinion on how to best represent urgency. Had we told them, some participants might say that the introduction of these identifiers would be better than not having them, forgetting about the change in the location shape.

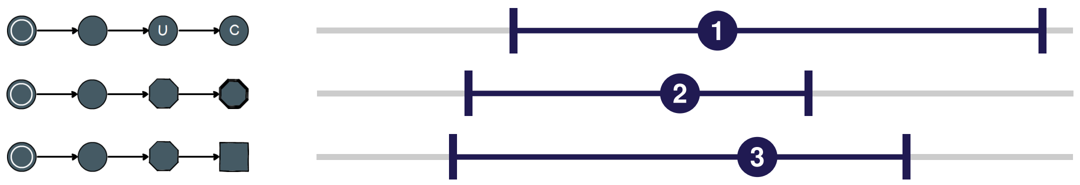


Figure 6.5: Ranking results for location urgency ideas.

6.4.2 Subcomponent Functionality Representation

As seen in Figure 6.6, participants mostly agree that they like Design Idea 4 (Textual Representation). Most participants said that this idea would allow them to get a good overview on what functionality the subcomponent represents. Furthermore, most participants agree that Design Idea 5 (Miniature Model Representation) would quickly become unusable when the encapsulated component is large or complex. Based on this, some participants dislike the idea where others would like functionality that supports switching between the two ideas, depending on the component. The idea here is to show a textual representation for complex or larger models while showing a miniature model for simpler models. The two different views are reflected in the ranking, where the participants disagree on whether they like or dislike the idea.

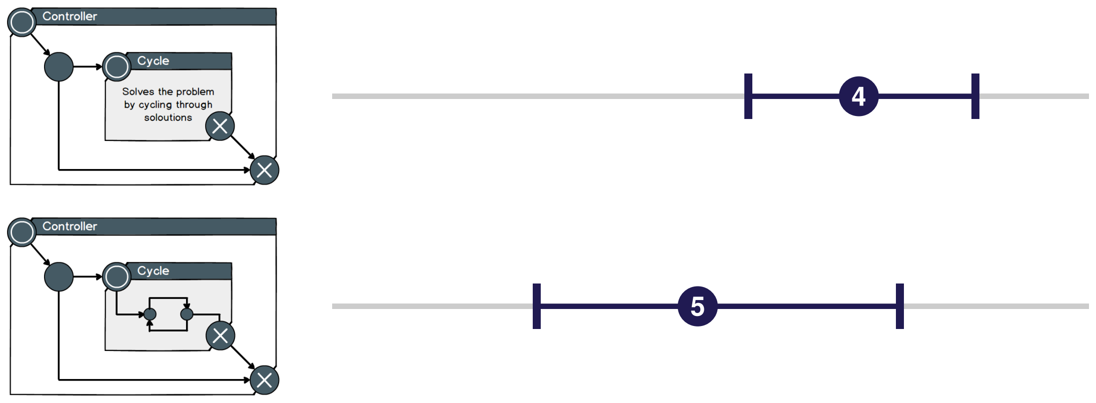


Figure 6.6: Ranking results for subcomponent functionality representation.

6.4.3 Edge Properties

When ranking the ideas concerning edge properties, participants agree that they dislike Design Idea 6 (Traditional UPPAAL Edge with Properties). Participants stated that they find it hard to remember which colors represent what property, especially when not having worked with the tool for some time. When considering Design Idea 7 (Single Box with Properties), people tend to agree that they like this idea more. Here, participants said that it seems nice to have the properties grouped, with an icon representing each property. A few participants said that it might be useful to use the current UPPAAL colors (e.g. yellow for select) for the background of the icons. The participants disagreed on whether they like or dislike Design Idea 8 (Multiple Boxes with Single Property). Here, people say that it might be confusing that the properties are split, but also that it could be nice to have freedom of placement, while still having the properties boxed.

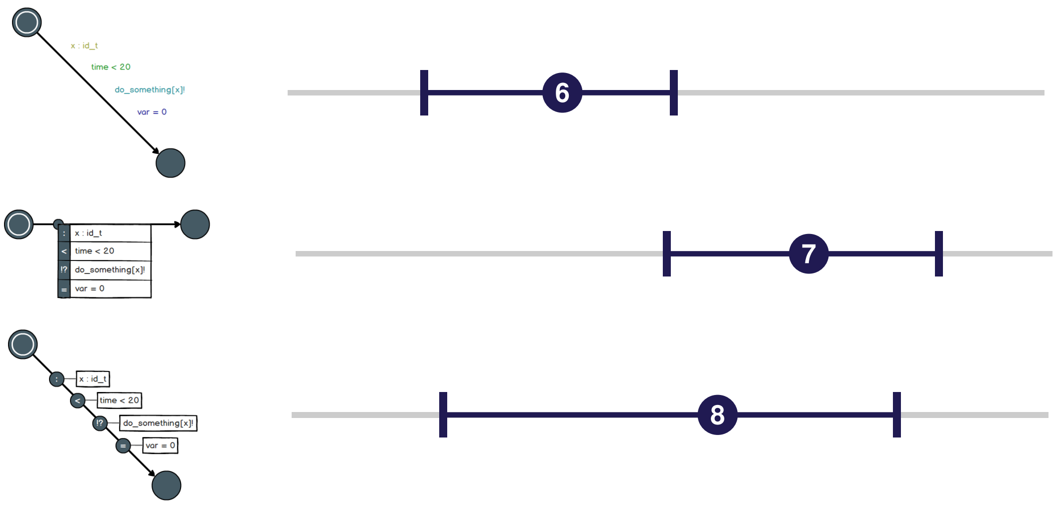


Figure 6.7: Ranking results for edge properties representation.

6.5 Discussion

While performing the evaluation, we noticed some potential problems with the way the evaluation constructed regarding the tasks *Rank the Different Proposals* and *Explain a Model*. This section will discuss these problems.

6.5.1 Concerns With Ranking

In respect to *Rank the Different Proposals*-task, it is important to note that the design ideas only show a fragment of a model, e.g. only an edge. These changes of visual representation were not merely aesthetic changes but were aimed to solve other interaction problems.

In Design Idea 7 (Single Box with Properties) and Design Idea 8 (Multiple Boxes with Single Property) the edge is presented in isolation, meaning that the participants did not see how scattered the nail-property idea could be or how much space the box-property idea would require. Participants are most likely influenced by the context of the design idea, and seeing one of the ideas in use, might change their preferences.

Another thing to have in mind with respect to the ranking task is that we have tried to be as unbiased towards our new ideas as possible. For instance, the center of a location were cleared by changing the shape to use it for identifiers, but Design Idea 2 (New Urgency with Octagon Shape) and Design Idea 3 (New Urgency with Octagon and Square Shapes) did not include these identifiers. We avoided this because we did not want to put any bias towards any of the ideas beforehand.

In general, the ranking test is to some degree an easy way of performing an A/B test for design ideas but without the expense to implement all of the different ideas. In effect, we believe that the results of the ranking task should be considered with this in mind. Participants were only exposed to bits of a model in isolation, which would probably influence their opinion. Despite this, we do believe that the feedback participants delivered, both directly, but also through describing their thought process when ranking the ideas, is useful to us.

6.5.2 Impact of Colors as Identity

The *Explain a Model*-task showed great results. However, this task can be seen as a test of how well the participants understand model checking. The model we provided for them to explain was designed to be easily understandable and made great use of colors. This caused some participants to understand the component/subcomponent idea almost instantly because they could relate the subcomponents to their corresponding component simply by using their colors as identifiers. This is what the colors are intended to do, so we are happy to see that it made sense to the participants.

However, seeing that colors are rarely used in UPPAAL models, we would like to explore solutions that force users to use the colors that are built into the tool. Our initial idea is to use a new, random, not already used color whenever the user creates a new component. This would give each component a unique color and, in extend, a unique identity with no effort required. Having such a feature would effectively mean that the tool would help the readers of the model in the same way as we did when designing the *Explain a Model*-task.

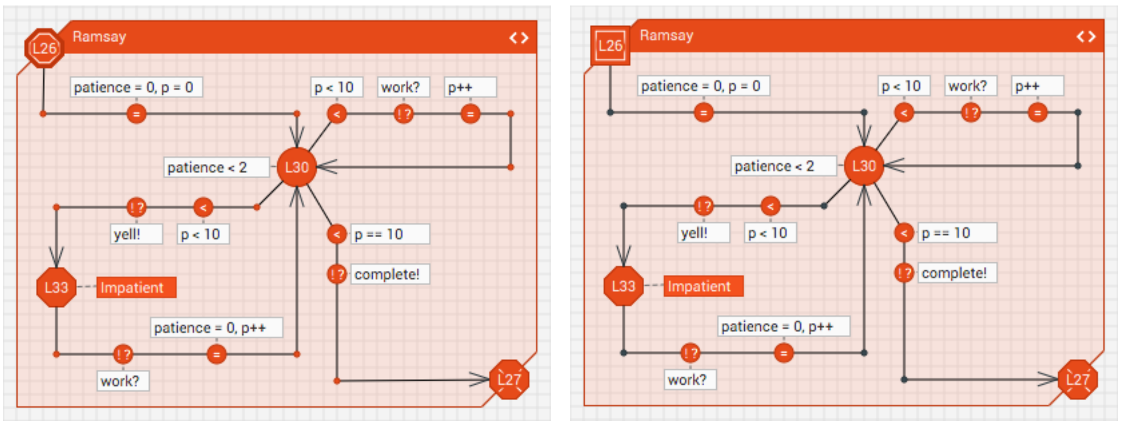
II Changelog

7 Visual and Minor Changes

This chapter briefly sums up the visual and minor changes that have been made to the H-UPPAAL as an response to the Expert User Evaluation (Chapter 6).

7.1 Visual changes

Based on the Expert User Evaluation (Chapter 6), there were found some concerns with respect to the visual representation of a model in H-UPPAAL. This section sums up what have changed. Taking a look at Figure 7.1, we notice two differences. Firstly by examining the initial location `L26`, we see that the way a committed location have changed from being an octagon with a bold border to being a square. And secondly, we see that nails are no longer colored to clarify what changes behavior of the model and what does not change behavior.



(a) Before changes. (b) After changes.

Figure 7.1: Location urgency and nail colors changed.

Some expert users had issues recognizing the initial location and had no concept of a final location when they opened the tool for the first time. For this reason, we implemented indicators that will show and animate briefly when a component is created like seen in Figure 7.2.

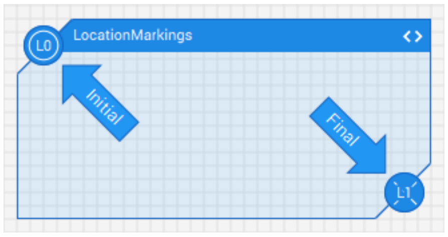
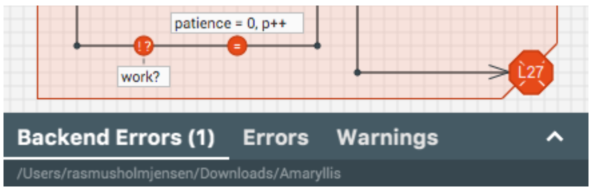


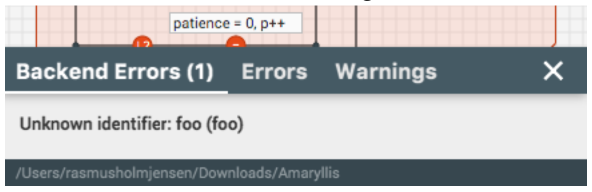
Figure 7.2: Location markings.

Lastly, we have made the message panel more responsive in the manner that whenever any message enters the panel, it will unfold itself. This is done to accommodate syntax errors and warnings as soon as they are introduced as visualized in Figure 7.3.



/Users/rasmusholmjensen/Downloads/Amaryllis

(a) Before changes.



Unknown identifier: foo (foo)

/Users/rasmusholmjensen/Downloads/Amaryllis

(b) After changes.

Figure 7.3: A responsive message panel.

7.2 Minor Changes

There have been added some small but still noteworthy features which should make the tool feel more like an IDE. This section will sum up these changes.

7.2.1 Replacing Subcomponents

There have been added an option that allows the user to replace one subcomponent for another from the context menu of a subcomponent as seen in Figure 7.4. This will allow users to try out different strategies in various components. This feature adheres to Principle 2 (Integrated Development Environment).

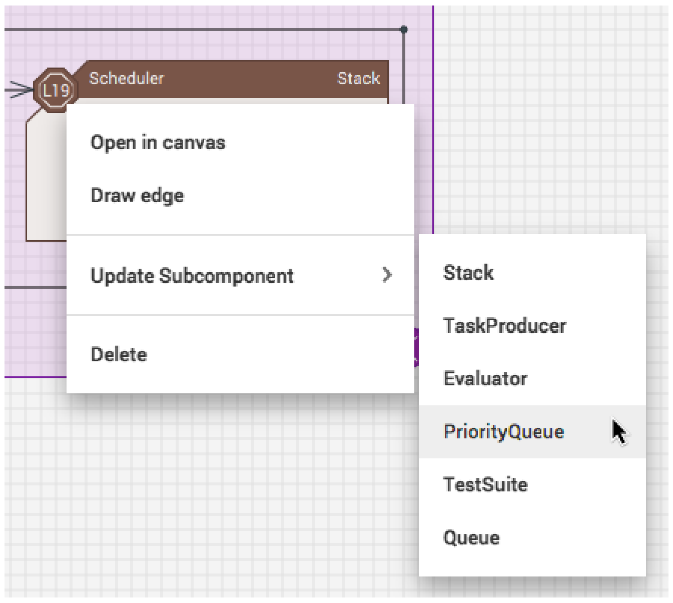
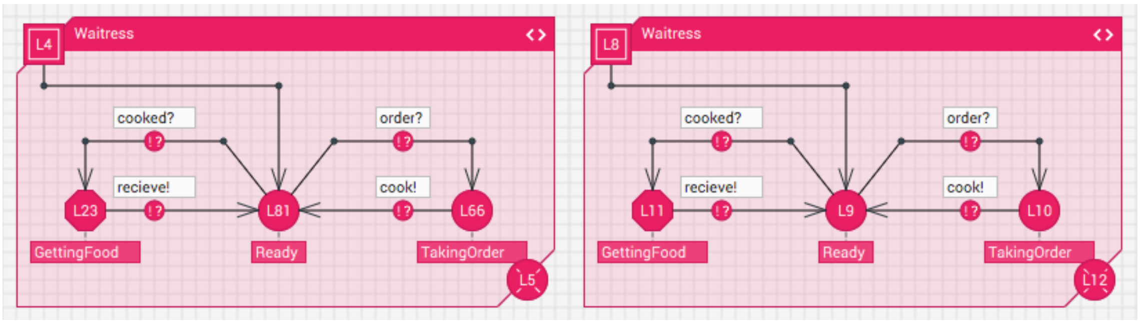


Figure 7.4: Replacing a subcomponent menu.

7.2.2 Location Identifier Rebalancing

The models in H-Uppaal have unique identifiers for locations across a project. In combination with the way we simply increment a counter for locations, the user can end up with models where the unique location identifiers can seem rather random as seen in Figure 7.5a. For this reason, we have introduced a rebalancing feature of these identifiers by performing a breadth first search in respect to subcomponents, which ensures that identifiers in a component are within the same span of numbers as seen in Figure 7.5b. This feature adheres to Principle 2 (Integrated Development Environment), however, allowing users to completely change all identifiers at once might have a negative influence on the overall readability of the model. Imagine that a user is used to working with specific location identifier, e.g. `L18` is the error location, while `L37` is the success location. These might be changed completely, forcing the user to use the new location identifiers.



(a) Unbalanced model. (b) Rebalanced model.

Figure 7.5: Location Identifier Rebalancing.

7.2.3 Multiple Projects

A feature that one might expect to be present in a modern tool would be the ability to open a project in a specific directory. In the previous version of H-Uppaal, this was not the case, and the tool simply opened a fixed folder relative to the executable. To better adhere to Principle 2 (Integrated Development Environment), we have now added this missing functionality, allowing users to use H-Uppaal for more than one project at the time. However, similar features like *save project as*, *import components from existing project* are still not present.

7.2.4 Updated Keyboard Shortcuts

Ideally, we would have liked to use the same keyboard shortcuts as Uppaal. However, we realized that the `ALT`-key (used in Uppaal to create locations, edges and more) is also used by Linux to manage the position of the application on the screen. Since comparability with Linux is desired, and these keyboard shortcuts are a core part of H-Uppaal, we decided to change all `ALT` keyboard shortcuts to use the `SHIFT`-key instead. This might seem like an issue in regards to Principle 1 (Backward Compatible), however, as mentioned in Usability Problems (Section 6.3) and Usability Problems (Section 6.3), people were not familiar with the `ALT` keyboard shortcuts in the first place. This might indicate that introducing `SHIFT` keyboard shortcuts is a elegant alternative.

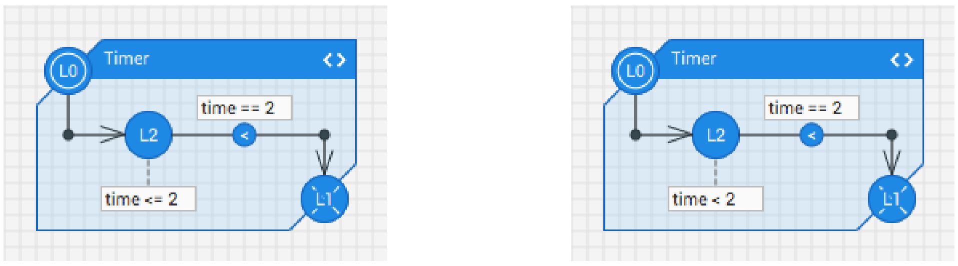
8 Utilizing the UPPAAL Backend

As described in Mourtizsen and Jensen, 2016, the H-UPPAAL tool is very dependent on the utilization of the powerful UPPAAL verification engine. Previously, this engine was only utilized to run relatively simple queries. This chapter will go through some changes that introduce some more advanced and interesting behavior to the H-UPPAAL tool, inspired by integrated development environments and our experiences with the model checking activity.

8.1 Automatic Reachability Analysis

As previously mentioned, modelers tend to utilize two types of queries during the model checking activity. One type of queries is utilized to debug the model. The other type of query a modeler might write is one that verifies the behavior of the model. We would like to facilitate the debugging queries better, making us adhere to Principle 2 (Integrated Development Environment). A feature that could assist the user passively during the development of a model is something we call *automatic reachability analysis*. The idea is that the tool continuously checks if all locations in a model are reachable. This would allow users to get notified when a location is not reachable. Based on our experience and expert user conversations, it seems that models often have some locations which should never be reachable, e.g. an error or deadlock location. If models contain locations which are not desired to be reachable, they are often annotated as error states with a name or sometimes color.

Automatic reachability analysis in H-UPPAAL should act like static code analysis of IDEs, which can detect if some part of the code is unreachable, for instance when the condition inside an if statement always evaluates to false, the code block inside is not reachable. Similarly, users should be notified of which parts of the model is not reachable. Looking at Figure 8.1 we see two versions of a `Timer` component, where a subtle change to the invariant of `L2` makes `L1` unreachable since the edge `L2 → L1` is never available.



(a) `L1` Reachable. (b) `L1` Unreachable.

Figure 8.1: Example of subtle change in model that changes reachability.

It is desired that the effect of changes like this one is instantly noticeable for the user. The most straightforward solution to do this is to run a simple reachability query for all locations in the model on every model change. The result of these queries then has to be reflected in the UI. We have decided only to mark the locations that are not reachable in a model with the same philosophy as in static code analysis where no answer is a good answer. Marking all locations that are reachable is similar to marking all lines of code in an programming editor that are syntactically correct, which most certain would add more noise than value. An illustration of how we visualize that a location is not reachable is applied to the previous example as seen in Figure 8.2. Here, `L0` and `L2` are reachable, while `L1` is not.

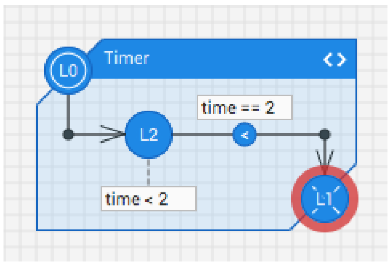


Figure 8.2: Location L1 marked as unreachable.

Since the `Timer` component only consists of 3 locations, we only have to run 3 queries to check if all locations are reachable, i.e. the following queries:

```
E<> Timer.L0     E<> Timer.L1     E<> Timer.L2
```

8.1.1 Running on change

Ideally, you would want the information this feature provides as fast as possible. However, in some cases, it seems better to wait for a small period before actually starting the background analysis. Imagine that you are writing a guard. Every single character you type or remove, have to be considered as a change, since changing `<=` $\rightarrow$ `<` could have a huge influence on the model. This does, however, also mean that when you are typing the guard `a > 2`, you will end up with a change, where the guard is incomplete, and in some cases syntactically incorrect, e.g. `a >`. In these cases, the background reachability analysis would not work, and therefore it would be better to wait for the user to introduce all of the changes he wants. In this case, this would be to finish writing the guard. We are not able to detect when the user is done typing a guard, but we can detect when the users have not been typing for some time.

8.1.2 Exhaustive Queries

However, as mentioned, for some locations it might be very time consuming to check if they are reachable. This can be problematic especially when a model is centered around avoiding an error location, effectively forcing us to explore the entire state space. For this reason, we need to ensure that the tool does not hog the CPU when doing the automatic reachability analysis. This could easily be solved by forcing a query to stop when a time limit is exceeded. A problem with this is then, how do we notify the user? If we do not mark the location in question, the user will think that the location is reachable. On the other hand, if we are not sure that the location is, in fact ,unreachable, the red marking might confuse the user. For this reason, we introduce an additional marking: yellow. Red now means that a location is guaranteed to be unreachable while yellow demonstrates uncertainty, i.e. we do not know for sure if a location is reachable or not. This will allow modelers to have locations that might be exhaustive to check for reachability while having the automatic reachability analysis running, like in the example seen in Figure 8.3. This illustrates a location Error that is only reachable when some other more complex part of the model synchronizes on the `error` channel.

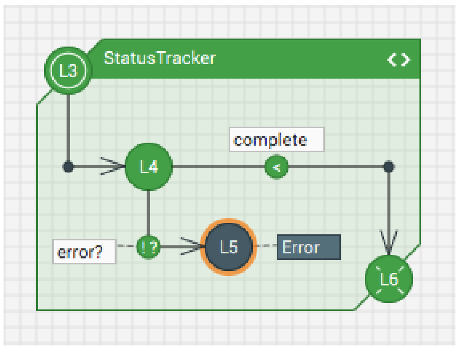


Figure 8.3: Example of uncertain reachability.

8.1.3 Optimizations

One could imagine that there are some optimizations one could introduce to optimize this reachability analysis to achieve faster results and reduce the impact on CPU and RAM usage. This section will go through some optimizations one could introduce to achieve this.

Start by Analyzing the Opened Component. It is not uncommon for a change in one component to influence the behavior of other components. This is the reason why we need to re-do the reachability analysis on the whole model every time the user introduces a new change. However, since we are working with visual changes (a colored aurora around the locations), it would make sense to run the reachability queries for the visible locations before the rest of the model. One might even go as far as to *only* run the reachability queries for the locations visually presented to the user. Let us, for instance, consider the component presented in Figure 8.4. When browsing this component, you only really care about locations L8 - L12 *until* you look at the other parts of the model. This would, however, require H-UPPAAL to run new reachability queries every time the user looks at different components. However, this might still be better than always running all queries. Further investigation is required on the matter before anything conclusive can be said.

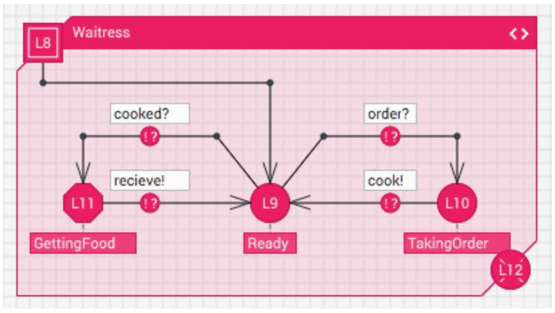


Figure 8.4: Waitress component.

Only Analyze Incomplete Components. Our experience tells us that not *all* areas of a model are finished at the same time. This means that some components are *completed* before other components. This would also mean that doing analysis on these components would be a "waste". To facilitate this, we have introduced a feature, where users can include or exclude components in this analysis. Figure 8.5 shows how a component can be either included or excluded from the analysis. The ability to toggle this functionality on or off, depending on the users need adheres to Principle 3 (Information Hiding).

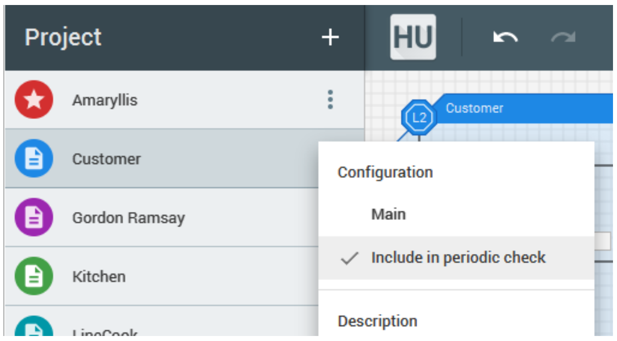


Figure 8.5: "Include in periodic check" option in component context menu.

Utilizing Metavariables. It might be interesting to investigate whether using metavariables [David and Larsen, 2011] could result in a performance increase by making some of the reachability queries redundant. Imagine a model with 20 locations, `L1`, `L2`, ..., `L20`. This would result in the reachability analysis running 20 queries. However, whenever running any one of these queries, we would potentially gain additional knowledge about other reachable locations. For instance, we might have edges `L5 → L6`, `L6 → L7`, and, `L7 → L8`. Executing the reachability query `E<> L8` *could* result in us gaining knowledge about `L6` and `L7` (storing this information in metavariables), causing their corresponding queries to become redundant. This scenario is visualized in Table 8.1. It might also be interesting to do a full exploration of the state-space while using metavariables to keep track of which locations are reachable. However, this raises additional problematic situations, such as dealing with very large state spaces, which we will not discuss in this report.

Waiting	**Running**	**Finished**	**Redundant**
`E<> L13`	`E<> L11`	`E<> L8`	`E<> L6`
`E<> L16`	`E<> L14`	`E<> L20`	`E<> L7`
`E<> L15`	`E<> L17`	`E<> L4`	
`E<> L19`	`E<> L2`	`E<> L12`	
`E<> L5`	`E<> L1`	`E<> L10`	
`E<> L3`		`E<> L18`	
`E<> L9`			

Table 8.1: Redundant queries scenario visualized.

Using Under- and Over-Approximation Techniques. It might be worth investigating whether utilizing under- and over-approximation [David and Larsen, 2011] techniques could yield meaningful results while optimizing the response time for the reachability queries. Remember that an indication of whether a location is reachable or not is "good enough". We do not necessarily need a full-blown guarantee that a particular location is reachable or not. The idea of automatic reachability analysis is based around assisting the user during the modeling process, rather than giving him deep insight into the model that is being developed. This effectively means that implementing these techniques, could provide the user with answers such as "This location might not be reachable" faster than "This location is definitely not reachable", and still allow him to utilize the automatic reachability analysis during model development.

8.2 Periodic Queries

After introducing the automatic reachability analysis into the tool, we thought to ourselves that users might have some model-specific queries that they keep executing many times modeling. To facilitate a tool that allows users to automate this, we have introduced something we like to call *periodic queries*. The concept is very similar to the approach in our automatic reachability analysis, but instead of reachability queries, users are able to run custom queries, and get the corresponding UPPAAL response, as they would with normal queries.

To create a periodic query, the user must firstly create a query as they would normally do. After this, the newly created query can be marked as periodic by clicking the three vertical dots to the right and then clicking "*Run periodically*". This is illustrated in Figure 8.6, where we have several queries for finding different Fibonacci numbers.

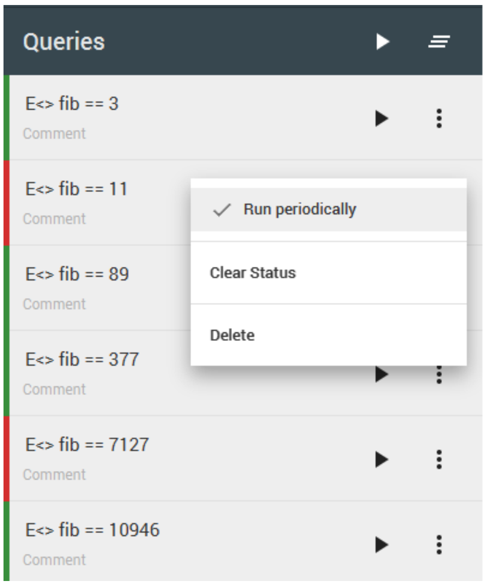


Figure 8.6: Periodic queries checking for fibonacci numbers.

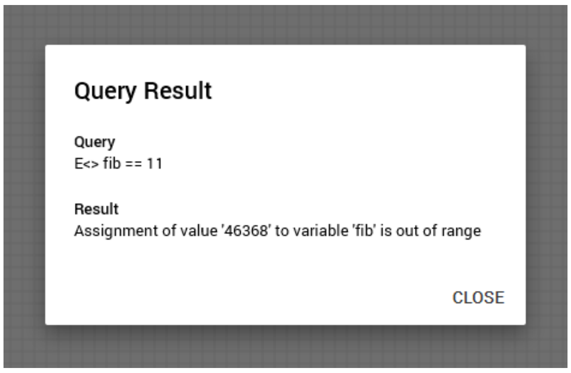
8.3 Query Feedback

In UPPAAL, running a query will return a status describing if the model adheres to the property specified in that query, i.e. if the query is successful. However, some queries will also return some additional information. One example of such could be the bounded liveness queries: supremum and infimum [David and Larsen, 2011]. These queries will find the supremum or infimum for a given expression, exp, for all reachable states that satisfy a particular predicate, Pr. The queries are formulated using the following syntax:

$$\texttt{sup}\{Pr\} : exp \qquad \texttt{inf}\{Pr\} : exp$$

Running this type of query will cause UPPAAL to respond with a dialog describing the result. This type of query is not the only one which returns some textual representation of the query result. Another example could be when doing a regular verification query which encounters a runtime error, e.g. a variable overflowing. In this case, the query will be marked as unsuccessful, and provide a meaningful error message.

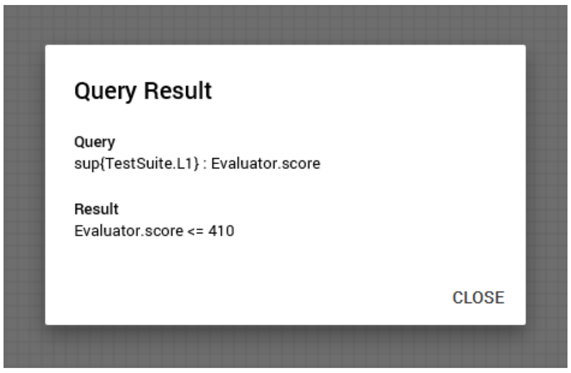
To accommodate the desire to return a textual representation of additional query details, we re-used a dialog we had previously implemented. This dialog can be seen in Figure 8.7. Implementing this feature will allow us to be more independent of the UPPAAL tool when testing H-UPPAAL (we do not have to open UPPAAL every time a query is unsuccessful, since we now have the same error messages).


Query Result

Query
E<> fib == 11

Result
Assignment of value '46368' to variable 'fib' is out of range

(a) Out of range error message.


Query Result

Query
sup{TestSuite.L1} : Evaluator.score

Result
Evaluator.score <= 410

(b) Supremuim query result.

Figure 8.7: Query feedback dialogs.

8.4 Multiple Engines

In old versions of H-UPPAAL, only a single instance of the UPPAAL engine was utilized. This effectively meant that *all* queries had to be run sequentially. Considering the feature where the user can run all queries in the query pane, this might not seem like a big deal. However, consider the features discussed in Section 8.1 and Section 8.2. These features are dependent on *fast results*, which would be virtually impossible if we kept this sequential behavior.

To solve this issue, multiple engines are instantiated and are allocated as a special resource that can be acquired when running queries. If no more resources are available, additional engines will be instantiated and allocated, until a maximum threshold is reached. This threshold is implemented to avoid hogging too much RAM and CPU. The point here is that we run out of computational power and memory at some point. This would indicate that, at some point, it is probably not worth instantiating additional engines, but instead wait for existing engines to be available. This points towards allowing users to specify this upper threshold themselves. We have not yet done this, but we imagine that when a settings panel is introduced, this threshold would be an ideal candidate to put in such a panel.

Currently, when engine resources are released, they are made available to be used again, but never de-allocated or destroyed. This means that if the system reaches its maximum threshold of engine resources, it will keep all of these engines in memory, waiting for them to be used. In most cases, it is very nice to keep these engines in memory since we do not have to re-instantiate them when doing new queries. However, this also means that H-Uppaal can become very memory-intensive, even in "idle"-mode. One could imagine a feature that would de-allocate these additional engines, freeing up some memory. However, one would have to consider how much time it takes to re-instantiate the engines, compared to how much memory they require.

III Final Evaluation and Thoughts

9 Performance Test

In order to verify whether hierarchies and integrated development environment features make a difference in formal verification tools, a performance test [Nielsen, 1993] comparing the performance of H-UPPAAL to UPPAAL have been conducted. The goal of this performance test is to formally test if the concepts in H-UPPAAL will add value for users. More specifically this performance test compares *efficiency* and *effectiveness* between the two tools during three modeling tasks. It is important to note that this test is not designed to test the performance of the verification engine, but rather to be able to compare the performance of users when using H-UPPAAL and UPPAAL. This chapter will cover how this test have been structured and shows an analysis of the results.

9.1 Participants

To compare the performance of UPPAAL and H-UPPAAL, we would like the expertise of the participants to be as similar as possible. For this reason, we have chosen to conduct the performance test using novel users. We imagine that experts in this field know all different kind of tools and concepts that could impact the comparison. However, the participants must still have an understanding of verification and model checking, and for that reason, we have chosen to use computer scientist students at Aalborg University. These students are ideal since all of them have been exposed to model checking and UPPAAL at some point. We were lucky to be allowed to provide a guest lecture in the course *Test and Verification*, and in extend, use the students in this course to conduct this performance test. Ideally, we would do a power analysis [Field, 2009, p. 58] to determine how many of these students we need to include in our study. Since this course have roughly 20 attendees, we deemed that it would be a relative fine sample size to deduce some effect. However, only 14 students ended up participating in the study.

9.2 Tasks

During the development of H-UPPAAL, we have given a lot of attention to the idea that users should be able to *query*, *expand*, and *create* models with ease. Based on this, we decided to evaluate the tools on these three parameters.

9.2.1 Task A: Querying models

An important step in developing models is the aspect of making inquiries. The idea here is to see if users can generate correct queries based on the new hierarchical structure, and extract meaningful information from the responses to these queries, i.e. can the users translate the results of a query into information about the model behavior.

9.2.2 Task B: Expanding models

The idea here is to evaluate if users understand a model that is not yet finished, and based on this, capable of expanding the model such that it is complete and models correct behavior. We have this tasks since models are often built over longer periods of time, repeatedly returning to the model like other work documents. Furthermore, you could also imagine that several people are working on the model at the same time.

9.2.3 Task C: Creating models from scratch

One could argue that without being able to create a model from scratch, a tool such as H-UPPAAL would be rather useless. Based on this, we believe that evaluating the users' capabilities to create brand new models is important.

9.3 Hypothesis

As described in Chapter 3, we would like to investigate whether the concepts introduced in H-UPPAAL increases *efficiency* and *effectiveness*. To do this, the following hypotheses have been defined. The goal of the performance test is to either prove or disprove these.

H_1: The tool used to complete an exercise will have an influence on the *efficiency* of a user.

H_2: The tool used to complete an exercise will have an influence on the *effectiveness* of a user.

These hypotheses will be tested in four different settings. One for each task, and one where these tasks are combined. Meaning that we will attempt to verify these hypotheses in a setting where we isolate the users' ability to *query*, *expand* and *create* models, and a more general setting, comparing the performance across these activities.

9.4 Test Structure and Setting

This performance test is a crossover study using a repeated-measures design [Field, 2009, pp. 457-505], meaning that we have a within-subject design. To do this, we permute the order of the tasks the participants are assigned to cancel out the carry over effect, i.e. we want to cancel the fact that knowledge from one task has an influence on the next. We do this by dividing the test into two sessions, one for each tool. Each session includes the three task in permuted order similar to the structure of the Expert User Evaluation in Chapter 6. Furthermore, we want the two sessions for a participant to be comparable, meaning that the order of tasks that a participant is assigned during the first session is identical in the second session. The carry-over effect also applies in respect to the order of tools the participants uses, and for this reason, half of the participants will start the first session using H-UPPAAL and the other half starting with UPPAAL.

The test will be conducted with all participants in parallel, meaning that we will have to be able to track each of the participants' individual performance with respect to time. For this reason, we have time-framed each task to 15 minutes for an easier track of time, but also to avoid dragging the participants into frustration or boredom depending on their speed. Figure 9.1 illustrates how the test will unfold for one particular participant. This participant will have to perform the tasks in the order A, C, B, using UPPAAL in the first session and H-UPPAAL in the second. The task sheets this participant was assigned to can be found in Appendix 16.

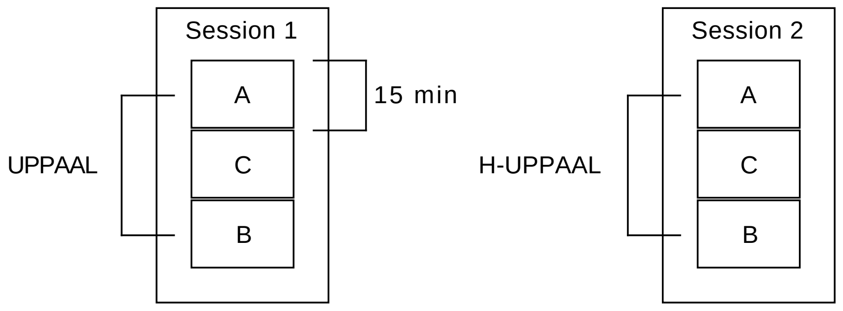


Figure 9.1: Layout of the test for one participant.

Lastly, the participants are allowed to solve the exercises however they see fit, as long as they do not use the simulator available in UPPAAL. We disallow this to make a more fair comparison between the two tools in their current state. We do however believe that the performance test should be re-run when a simulator is introduced into H-UPPAAL.

9.4.1 Laboratory and Field Testing

In respect to the setting of a usability test, one can consider conducting it in a laboratory environment where everything is controlled. Another option is to conduct the test in the field, where the test happens where the users already exist, where one does not have to simulate an office environment because the test is conducted at the users' office in the first place. Essentially a field study trades control for a realistic setting [Kaikkonen et al., 2005].

The setting of this test is a combination of a field and lab study. We let the users be in the "field" doing their exercises as they would otherwise during the *Test and Verification* course. This means that they are allowed to use their personal laptops. However, we enforce "lab"-rules to fit the test structure, essentially trying to control the setting, like you do in a lab study. Besides the participants being able to work in a more relaxed environment, this setting also allows us to easier handle many participants at once.

9.5 Results (Efficiency)

The time a participant spent on each exercise were tracked during this performance test, to measure the *efficiency* of the users. The completion time for each participant can be seen in Table 9.1 for H-UPPAAL and Table 9.2 for UPPAAL. Entries are formatted using the `minutes.seconds` format, describing the amount of time the participant used to finish a particular exercise. The cells are colored in such a way that white indicate that the participant used up all allocated time for an exercise, where more orange cells indicate that the participant used less time.

	1	2	3	4	5	6	7	9	10	11	12	13	14	15
A	12.43	15.00	15.00	06.35	15.00	05.17	10.38	15.00	09.29	15.00	15.00	15.00	15.00	15.00
B	11.49	15.00	15.00	10.19	11.40	15.00	13.55	15.00	15.00	15.00	12.22	09.05	15.00	15.00
C	05.41	15.00	15.00	09.30	15.00	12.26	12.42	15.00	15.00	15.00	14.51	08.40	14.35	15.00

Table 9.1: Time used in H-Uppaal.

	1	2	3	4	5	6	7	9	10	11	12	13	14	15
A	07.10	10.34	15.00	10.29	06.15	10.28	04.53	15.00	04.21	06.36	10.39	08.09	15.00	15.00
B	08.45	14.25	15.00	15.00	14.35	13.16	09.55	15.00	15.00	07.55	15.00	08.30	15.00	13.22
C	06.00	15.00	15.00	15.00	10.50	15.00	05.54	15.00	13.00	08.24	15.00	05.15	15.00	15.00

Table 9.2: Time used in session Uppaal.

Taking a look at the means and standard deviations for H-Uppaal and Uppaal (Table 9.3 and Figure 9.2), we see an indication that the tool used had some effect on how much time a participant spends on completing a task in favor of Uppaal. However, we would like to have some statistical foundation that the *efficiency* is affected by the tool. For this reason, we have analyzed our results using a *t*-test which is used to compare two groups of people and test if there are significant evidence that the tool has an effect on the measured time. A *t*-test is particularly useful when you have a categorical, independent variable (the tools) and a continuous, dependent variable (time measurements). We use the paired samples *t*-test [Field, 2009, pp. 324-341], since the first group, using H-Uppaal, and the second group, using Uppaal, are dependent i.e. they are of the same participants. We have conducted such a test for each exercise, A, B, and C, and one test where we considered all exercises combined.

Exercise	**Tool**	**Mean**	**SD**
A	**H-Uppaal**	12.49	03.28
	Uppaal	10.00	03.52
B	**H-Uppaal**	13.31	02.04
	Uppaal	12.55	02.48
C	**H-Uppaal**	13.06	03.01
	Uppaal	12.06	03.59
All Combined	**H-Uppaal**	13.09	02.51
	Uppaal	11.40	03.43

Table 9.3: Means and standard derivation (SD) for H-Uppaal and Uppaal in `minutes.seconds`

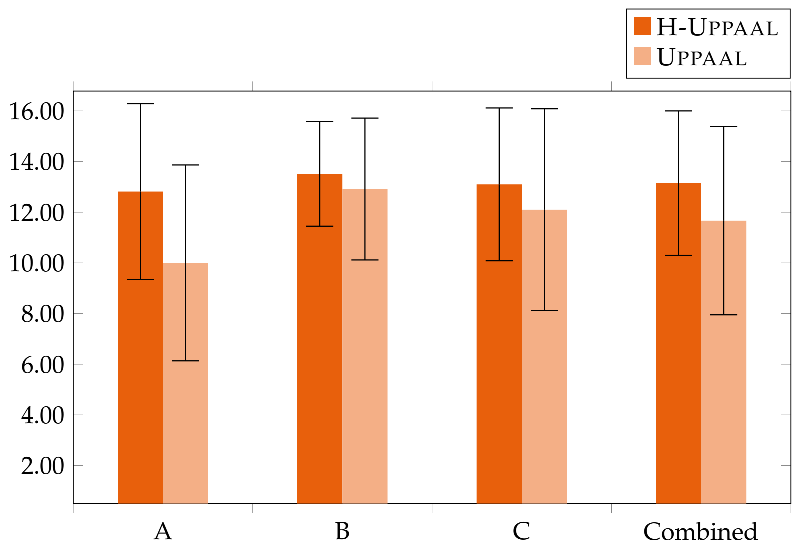


Figure 9.2: Means and standard deviations for H-UPPAAL and UPPAAL visualized.

Exercise A
There is **a significant** difference in the task completion time and tool with:
$t(26) = 2.065$, $p = 0.049$

Exercise B
There is **no significant** difference in the task completion time and tool with:
$t(26) = 0.649$, $p = 0.522$

Exercise C
There is **no significant** difference in the task completion time and tool with:
$t(26) = 0.750$, $p = 0.460$

All combined
There is **a significant** difference in the task completion time and tool with:
$t(82) = 2.061$, $p = 0.042$

9.6 Results (Effectiveness)

Each exercise has been ranked on a boolean scale: **Incorrect** or **Correct**. The following evaluation rules were established to more precisely judge the correctness of the exercises. Ideally, we would hand this information over to a third party to do the actual evaluation of the exercises to avoid being biased towards one of the tools. However, since we do not have the resources to do this, we decided to simply do them ourselves. The results can be seen in Table 9.4 and are visualized in Figure 9.3.

Correctness Criterion for Exercise A
If the participant can formulate at least one query that retrieves some supremum (worst case) or infimum (best case) values from the given model and if the values provided on the exercise sheet are also correct, the exercise is marked as **Correct**, otherwise **Incorrect**.

Correctness Criterion for Exercise B
If the participant can extend the provided model by inserting a `Gordon` component/template correctly while modeling the described behavior correctly, the exercise is marked as **Correct**, otherwise **Incorrect**.

Correctness Criterion for Exercise C
If the model created by the participant enables queries that are successful with the first 20 Fibonacci numbers and somehow fail with other numbers, the exercise is marked as **Correct**, otherwise **Incorrect**.

Exercise	Tool	Incorrect	Correct	Total
A	**H-UPPAAL**	9	5	*14*
	UPPAAL	4	10	*14*
	Combined	*13*	*15*	*28*
B	**H-UPPAAL**	12	2	*14*
	UPPAAL	11	3	*14*
	Combined	*23*	*5*	*28*
C	**H-UPPAAL**	2	12	*14*
	UPPAAL	7	7	*14*
	Combined	*9*	*19*	*28*
Combined	**H-UPPAAL**	23	19	*42*
	UPPAAL	22	20	*42*
	Combined	*45*	*39*	*84*

Table 9.4: Task correctness rates.

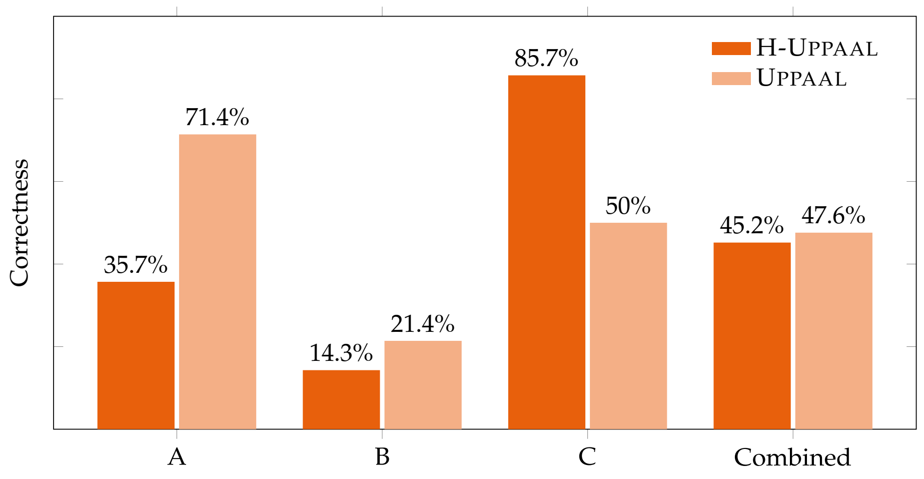


Figure 9.3: Task correctness rates visualized.

The correctness rates for the different tasks indicate that there is in fact some correlation between what tool is used and the *effectiveness* of the participant. To be sure that there is significant evidence for this correlation we used a Pearson Chi-Square test to analyze the results [Field, 2009, pp. 687-692]. We use this method since both our independent variable (the tools) and the dependent variable (correctness) are categorical. Using this test, we will see if there is any significant association between these two variables. We have, as with the time measurements, conducted a Chi-Square test for each exercise, A, B, and C, and one test where we considered all exercises combined, based on the results in the contingency matrices in Table 9.4.

Exercise A
There is **no significant** association between correctness and tool, given:
$\mathcal{X}^2(1, N = 28) = 3.590$ with $p = 0.058$

Exercise B
There is **no significant** association between correctness and tool, given:
$\mathcal{X}^2(1, N = 28) = 0.243$ with $p = 0.622$

Exercise C
There is **a significant** association between correctness and tool, given:
$\mathcal{X}^2(1, N = 28) = 4.094$ with $p = 0.043$

Combined
There is **no significant** association between correctness and tool, given:
$\mathcal{X}^2(1, N = 84) = 0.048$ with $p = 0.827$

9.7 Interpreting the Results

After the statistical analysis of the results, we found that there are in fact some correlations between the performance of the participants during the exercises and the model checking tool used. This section will cover how, when and why this is the case.

9.7.1 Task A: Querying models

When it comes to querying the model, we found significant results that UPPAAL is the more *efficient* tool (H_1 is accepted in this setting). We did not find any significant difference in the *effective* of the two tools (H_2 is rejected in this setting).

When investigating the exercises handed in by the participants, we found that some of them had trouble writing the correct queries in H-UPPAAL, effectively resulting in them spending all their time and not completing the exercise. 5 participants had trouble with the way H-UPPAAL queries must be formulated [Mourtizsen and Jensen, 2016], causing them to write queries similar to the one seen in Figure 9.4a, where the correct query would be as seen in 9.4b. Furthermore, 1 of the participants had swapped the predicate and and expression (see Section 8.3). Others might have gone through the same struggle as these, but ended up with the correct query in the end. This might explain why H-UPPAAL users are significantly less *efficient*, but not significantly less *effective*.

```
inf{Evaluator.L18} : score
```
(a) Example of incorrect query.

```
inf{TestSuite.L1} : Evaluator.score
```
(b) Correct query.

Figure 9.4: Incorrect and correct queries.

One might also want to consider the fact that in H-UPPAAL, we have the unique identifiers, e.g. `L1`, but also descriptive names, such as `Finished`. Currently, when querying the model, you have to use the unique identifiers. In UPPAAL, you only have descriptive names, making it a bit more simple. This difference might cause some confusion when considering that the participants only used the UPPAAL tool prior to this evaluation. However, as described in Mourtizsen and Jensen, 2016, it is desired that both identifiers is supported when querying a model.

9.7.2 Task B: Expanding models

When it comes to reading a model and then expanding upon it, we found no significant difference in neither *efficiency* nor *effectiveness* (H_1 and H_2 are both rejected in this setting). This might be caused by the complexity of the model that were provided to the participants, i.e. if the model is too complex, participants might spend all their time on understanding the model, leaving them with little to no time to actually expand it. Seeing that users spent the most time on this exercise (on average 13.31 for H-UPPAAL and 12.55 for UPPAAL) and that most people (12/14 in H-UPPAAL and 11/14 in UPPAAL) got this exercise marked as *Incorrect*, we believe that this is the case. However, considering that there are no significant difference in neither *efficiency* nor *effectiveness*, this might also indicate that users performed equally in the two tools.

9.7.3 Task C: Creating models from scratch

When considering creating models from scratch, we found no significant difference in the *efficiency* of users (H_1 is rejected in this setting). We did, however, find a significant difference in *effectiveness* in favor of H-UPPAAL (H_2 is accepted in this setting).

Investigating the exercises solved in UPPAAL that the participant handed in, we can not see any clear tendency as to why UPPAAL users are less *effective*. 2 participants made incorrect use of arrays, looking up wrong indices as seen in Figure 9.5. 3 participants updated some intermediate calculation in the wrong order, causing the query always to fail. 2 participants used undeclared variables or otherwise introduced syntax errors.

```
Arr[i] - 1
```

(a) Incorrect array lookup.

```
Arr[i - 1]
```

(b) Correct array lookup.

Figure 9.5: Incorrect and correct array lookups.

We believe that the introduction of integrated development features such as continuous syntax check are helping users to spot issues, such as undeclared variables. Furthermore, the more structured approach of showing edge properties might give users a better insight in the model behavior, leading to more correct models in H-UPPAAL.

9.7.4 Combined

When considering all of the exercises combined, we find a significant evidence that users are more *efficient* in UPPAAL compared to H-UPPAAL (H_1 is accepted in this setting). We see no significant difference in users' *effectiveness* (H_2 is rejected in this setting). This might indicate that the users are more familiar with UPPAAL since they have been working with it during the *Test and Verification* course. Furthermore, since there are no significant difference in *effectiveness*, we believe that H-UPPAAL adheres well to Principle 1 (Backward Compatible). We believe that if we had introduced concepts that did not make sense for the users, this would be reflected in the measured *effectiveness*.

Despite some problematic situations, described in Section 9.8, we believe that this evaluation shows that H-UPPAAL has great potential. Given some more work, especially in regards to the querying of models, we believe that it would be able to compete with or even out perform other model checking tools.

9.7.5 Limitations

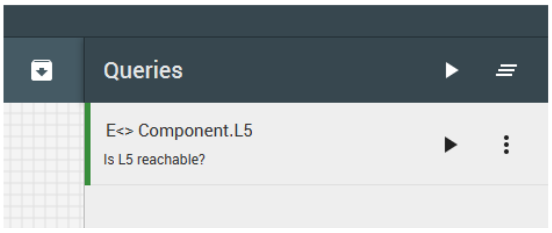
We deem that a larger sample set, i.e. more participants during this performance test, would give us a better statistical foundation. Besides the limited sample size, we also had some limitations during the test, where, because of practical implications, we did not get the fifty-fifty split we had planned. Participant 8 had to leave before even getting started with the tasks, and participant 10 started with the exercises intended for the second session. This resulted in us having 9 participants who started the first session with H-UPPAAL and only 5 started with UPPAAL, which could have some influence on the results. Alongside this issue, we also had a two participants switching machines after the first session to the next, meaning that the used different PC's to run the two tools. Many of these complications are expected when you do have limited control of the environment, i.e. not a true lab setting.

9.8 Problematic Situations

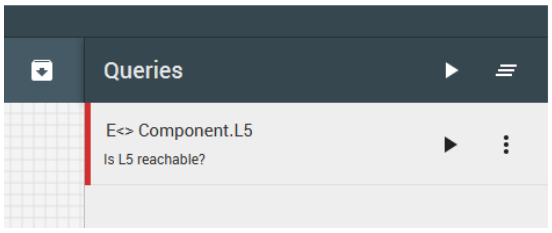
In the period where the test was conducted, we offered to help participants if they were experiencing any trouble with the tools, e.g. when a tool crashed or when the exercise formulation was unclear. We wanted to help with any problems that should not occur in such a test, such as crashes, but not provide help with solving the actual task, since this would influence the test results. When helping the different participants, we came across some problematic situations that might be worth considering.

9.8.1 Color Blindness

We were made aware by two of the participants that they were in fact colorblind. This caused them to be unable to see the color-status on the queries they had run. Figure 9.6 shows the two query status indicators. One of the participants informed us that he had been taking screenshots of the tool, and looking up the RGB-values to determine the color. The other participant stated that he knew that the query would probably be unsuccessful when he got errors such as "*Syntax Error*".



(a) Successful query



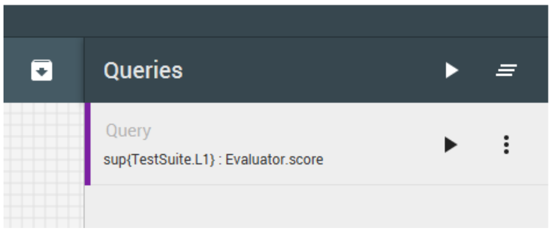
(b) Unsuccessful query

Figure 9.6: Query status indicators.

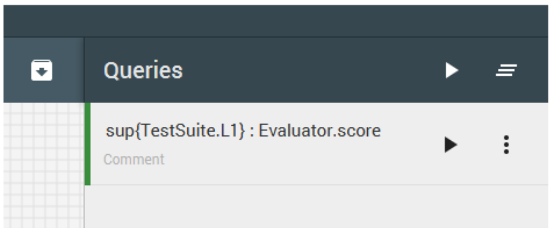
According to the National Eye Institute, as many as 8% of men and 0.5% of women with northern European ancestry have a form of red-green color blindness [*Facts About Color Blindness*]. Based on this, we believe that a solution to this problem will have to be implemented. One of the participants suggested using a textual response in addition to the color indicator currently implemented. We believe that this is a good solution, however, due to limited time, we will not implement it.

9.8.2 Running Queries

Two participants asked for help in regards to a strange response when they tried to run queries. They demonstrated that running the query results in error: "*ServerException: Can only handle one property at a time*". To us, the error was clear. The participants had both entered the query in the comment field. They might have thought that the "Query"-text was simply a headline for the input field below. Figure 9.7a illustrates how the participants wrote the query in the wrong place, where Figure 9.7b shows how the query is successful if placed in the correct field.



(a) Query placed in comment field.



(b) Query placed in query field

Figure 9.7: Query field misconception.

Since the actual query field is empty, it would be relatively easy to inform users that they cannot run an empty query. However, we do not believe that this solves the core of the problem. Instead, we believe that it should be more clear where to write the query, and where to write the comment (if this is even needed). However, due to limited time, we will not be implementing anything that tries to solve this issue.

10 Conclusion

Model checking is a technique used today for various purposes. A lot of work is going into improving this technique, primarily concerning the efficiency and speed of verification of models. However, we believe that it is also important to consider the actual creation of models, since this part is strongly dependent on the users' ability to work with models. Based on this, we raised the question:

> How can the introduction of hierarchies and integrated development features improve the workflow in the model checking activity?

In the fall of 2016, we developed a new tool called H-UPPAAL, focused on improving the model checking activity by trying to speed up the modeling instead of the verification. Here, new concepts like hierarchies and integrated development features were introduced, while utilizing a mature engine to do the underlying verification.

The work we have done this semester was to investigate further if these concepts and ideas would add some value to the activity. An initial evaluation was conducted using a small group of experts and a think-aloud test, where we found some usability problems with the newly developed tool. The test also showed that participants found creating and understanding H-UPPAAL-models rather intuitive.

Furthermore, we have conducted a performance test, comparing the mature model checking tool UPPAAL and our newly developed tool H-UPPAAL. This evaluation consisted of three exercises in each tool measuring *efficiency* and *effectiveness*, using computer scientist students who are novices in the model checking domain. We found significant evidence ($p = 0.042$) that UPPAAL was, in general, a more *efficient* tool, and that it was superior ($p = 0.049$) during the exercise concerned with querying existing models. On the contrary, we found significant evidence ($p = 0.043$) that H-UPPAAL is a more *effective* tool during the exercises regarding creating models from scratch.

Besides evaluating the tool, we have also investigated interesting ways of utilizing the UPPAAL backend. One of the features introduced is called *Automatic Reachability Analysis*. This feature is similar to static code analysis, however, instead of analyzing program logic, analysis of state space is performed on-the-fly. We deem that it is this type of feature, that will assist modelers during their work, e.g. by finding bugs, or wrong parts of a model, faster. More generally, we have utilized this on-the-fly technology to enable periodic queries, introducing unit-test-like queries providing the users with continuous feedback, which will guide them during the model checking activity.

It seems that with limited efforts we have in fact been able to improve the model checking activity. A year ago a hierarchical version of UPPAAL was only an idea. Today we have a functional tool that utilizes many years of research in the model checking domain while being inspired by modern concepts like integrated development environments. Based on this, the comparison of performance, feedback from experts users, our work and experience in the field, we believe that the concepts and ideas in H-UPPAAL show great potential and we encourage further work in the area.

11 Future Work

This chapter covers thoughts and ideas we have regarding the H-UPPAAL project and how they could be realized. The overall idea with this chapter is to get an idea of how H-UPPAAL can be shaped into a tool that can compete with mature model checking tools such as UPPAAL by introducing interesting and useful concepts, further improving the model checking activity. This chapter will not cover areas for improvements or further work already covered in Mourtizsen and Jensen, 2016.

11.1 Reading Performance Evaluation

Being able to print, and in general, share a model is something we value important Principle 5 (Printable). This is important in a research setting where we would like to be able to present models in scientific papers, but also in a development environment, where clean documentation of design is often desired. For this reason, we would like to have some evaluation in respect to this. One could consider doing some qualitative study involving experts in the domain of model checking. Besides this, we would like to conduct a performance test in respect to this. This chapter outlines how such a test could be conducted and what hypotheses we would like to prove. However, due to limited time in the project, this test has not yet been realized.

11.1.1 Hypothesis

We have two hypotheses which we would like to confirm. Namely that a hierarchical modeling language increases the performance in understanding the behavior and design of a system or problem. This can be seen as two hypotheses concerned with *efficiency* and *effectiveness*:

H_3: Readers are more *efficient* in understanding a model presented in a hierarchical language compared to a non-hierarchical one.

H_4: Readers get a *effective* in understanding when a system is presented in a hierarchical language compared to a non-hierarchical one.

11.1.2 Setting and Structure

To get measurements that could prove or disprove these hypotheses, we suggest a structure very similar to the one done in Chapter 9. There is a limited amount of model checking experts available at Aalborg University, and for this reason, we would like to have one particular participant perform a task using both the hierarchical language of H-UPPAAL and the non-hierarchical language of UPPAAL. We suggest having at least two scientific models or system documentations that can be produced in both UPPAAL and H-UPPAAL. A way of getting realistic models would be to find existing complex UPPAAL models in work documents from developers or in scientific papers, translate them to H-UPPAAL and then use these to do the comparison. In this test, it is also important that order of the tasks is permuted and that half of the participants starts with one language and the other half starts with the other.

Regarding structure, we think that an exam would be a good source of inspiration. For each model, the participants are required to explain; we would give them some preparation time. Following we would have a digital questionnaire, where the participants will have to answer questions in regards to the model they just investigated. This questionnaire should keep track of time, and then we would have both time measurement and some answers to evaluate and determine how well the participant understood the model.

With the result from such a test, we could see if there are significant evidence that could prove or disprove that a language with hierarchies has an effect on *efficiency* (H_3) and *effectiveness* (H_4) of a model.

11.2 Improvements to Periodic Queries

The introduction of the periodic queries increases the memory footprint and CPU utilization of H-UPPAAL. During our use of the tool, we found that in some cases the tool would end up using quite a lot of resources, which indicates that these periodic queries need to be handled with even more care. A result of this can in some cases lead to the user never finding an answer since the tool tries to run all periodic queries on every change. A solution to this problem would be to exclude the queries that repeatedly are unable to complete or at least run them last. In respect to the automatic reachability analysis where we set a deadline, one could consider running the exhausting location reachability checks last but extend the time window in which it is allowed to run. This issue is, in general, a question of trading resource acquisition for quick feedback to the user, i.e. if a query takes too long, it might not even be "worth it" to check.

Another feature which might be interesting would be to consider which queries often change status, e.g. which locations are switching between unreachable/reachable. The idea is here that if a location has been reachable the last many times we checked it, it probably will still be reachable when we check it again. Because of this, one might want to prioritize queries that are more likely to change status.

11.3 Simulator

During the development of the tool, and during the planning of the two tests, conducted during this semester, we found that one of the features that H-UPPAAL strongly lacks is an option for exploring the state space of a model manually, as simulator of the UPPAAL. We have put lots of work into enabling on-the-fly testing and debugging with the introduction of automatic reachability analysis and periodic queries. However, without a functionality to investigate how and why a model does or does not adhere to a certain property, limits the users' ability to resolve faults and understand the behavior of models. In other words, a feature that can express how the model transitions from one state to the next, like the simulator of UPPAAL. This will enable better debugging of the models, but will also provide the user with traces of properties, allowing them to explain exactly how a model does or does not have a particular quality.

The challenge with this is not retrieving the trace and states from the verification engine, which is already possible through its interface. Instead the challenge is presenting these states and traces to the user. One has to consider how one can keep abstraction and decomposability in models while displaying the necessary data about a state or trace. We suggest having a look at how other model checking tools that have hierarchies, like CPN-Tools [Jensen et al., 2007] or TAPAAL [Byg et al., 2009], displays the states and transitions between them.

11.4 Support for Multiple UPPAAL Versions

To utilize the UPPAAL engine for verification of H-UPPAAL models, we utilize the `model.jar` library distributed with different UPPAAL versions (this works as an interface to the UPPAAL backend). Furthermore, upon using H-UPPAAL for verification, the users would have to copy his UPPAAL binaries (backend) to a specific directory, relative to the H-UPPAAL distribution. When doing this, one has to make sure that the versions of the `model.jar` library and UPPAAL binaries are the same. Currently, H-UPPAAL is compiled with version `4.1.19` of the `model.jar` library, causing it to be incompatible with older versions of the `model.jar` library. This is not optimal, since many people still use version UPPAAL `4.0.14`, and would have to download new binaries to use H-UPPAAL.

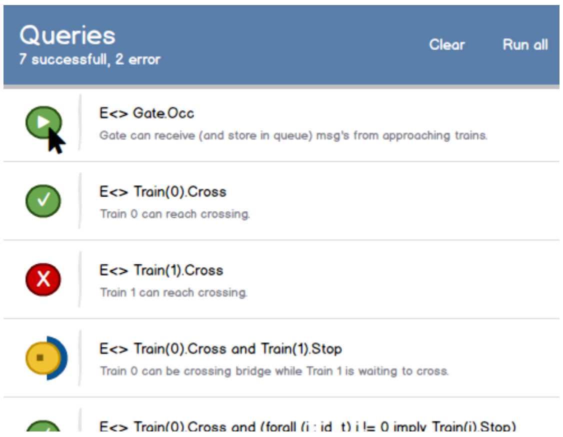
To improve this situation, one could include different versions of the `model.jar` library, and let the user pick which version of UPPAAL engine his wishes to use. Alternatively, it might be possible to detect the version of the UPPAAL binaries and use the corresponding version of the `model.jar` library.

11.5 Addressing Colorblindness and Accessibility

During our performance test, we ran into the issue that two of the participants were colorblind, who had a very hard time determining if a query was successful or not due to the color strips that indicate the result of the query as seen in Figure 11.1b. Because this disability is rather common (8% of men 0.5% of women [*Facts About Color Blindness*]) and the fact that the result of queries is key to a model checking tool, this issue must be solved in some way. We had a few suggestions from the participants with colorblindness, but we found that an early mockup of the tool did not have this issue, where we used both colors and symbols to indicate the result as seen in Figure 11.1a.
This particular issue in itself should be looked into, but more importantly, one should be more considered and have common disabilities, like this one, in mind when designing a GUI. This is especially important when designing key functionality like the query pane is for H-UPPAAL. However, some people might have mobility issues and for that reasons have a hard time using the mouse. Other might be sensitive to flickering light or sound and could trigger seizures. Implementing accessibility, like having a colorblind mode or enabling a keyboard only could address some of the common disabilities.

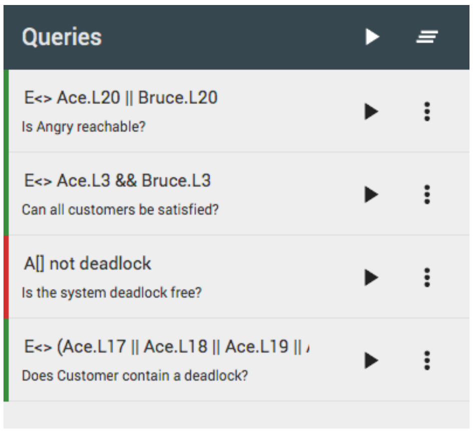
11.6 Reworking the Query Pane

In UPPAAL, there exists many different options and techniques for how a query can be executed by the verification engine. One example of such techniques are *over-* and *under approximation* [Gerd Behrmann, 2006], which are useful techniques when dealing with *safety* and *liveness* properties, respectively. In UPPAAL, you pick which techniques and settings to use through a global settings menu, however, in our experience, these are


Queries
7 successfull, 2 error

E<> Gate.Occ
Gate can receive (and store in queue) msg's from approaching trains.
E<> Train(0).Cross
Train 0 can reach crossing.
E<> Train(1).Cross
Train 1 can reach crossing.
E<> Train(0).Cross and Train(1).Stop
Train 0 can be crossing bridge while Train 1 is waiting to cross.
E<> Train(0).Cross and (forall (i : id_t) i != 0 imply Train(i).Stop)

(a) Initial query pane mockup.


Queries
E<> Ace.L20 || Bruce.L20
Is Angry reachable?
E<> Ace.L3 && Bruce.L3
Can all customers be satisfied?
A[] not deadlock
Is the system deadlock free?
E<> (Ace.L17 || Ace.L18 || Ace.L19 ||
Does Customer contain a deadlock?

(b) Present query pane.

Figure 11.1: Query pane mockup and implementation.

often very dependent on which query that you are running. This causes these settings to become somewhat tedious to change over and over while developing models since you are running different queries dependent on the situation. Because of this, we would like to re-work the query pane in H-UPPAAL to accommodate this, allowing users to set techniques and settings *per query*. Furthermore, it might be interesting to keep track of previous results of queries, especially considering results from infimum and supremum queries [Gerd Behrmann, 2006]. One could even go as far as to keep a history of what the model looked like the last time a specific query was successful, allowing users to roll back changes which might not be wanted.

11.7 Visualizing Communication

In our experience within the model checking activity, it is often difficult to keep track of synchronization between templates (and, in H-UPPAAL, components). We believe that introducing a feature that visualizes communication between components would help to improve the general understanding of where and how different synchronization channels are used. In H-UPPAAL, one might be especially interested in such a feature; this communication can be seen as a part of its interface, e.g. input/output channels. In extension to the notation introduced in Section 1.3, this section will also use the following:

Notation 11.1 Let $A \xrightarrow{com} B$ denote that component A and B contains edges with the synchronization properties `com!` and `com?` respectively.

11.7.1 Subcomponent Communication

Imagine a component, C, with two subcomponents, $a_1 : A$ and $b_1 : B$, running in parallel. If we have $A \xrightarrow{communicate} B$, these subcomponents would be able to communicate over the `communicate` channel. However, to visually *see* this, we would have to open up both component A and B. Even though this works, we believe that it could be improved by describing this communication when viewing the places where the components are used, which, in this example, would be in the C component. This communication is visualized in Figure 11.2, where subcomponents a_1 can synchronize with b_1 over the `communicate` channel. This idea would adhere to Principle 4 (Identity and Relation), since it *links* the subcomponents together through their communication.

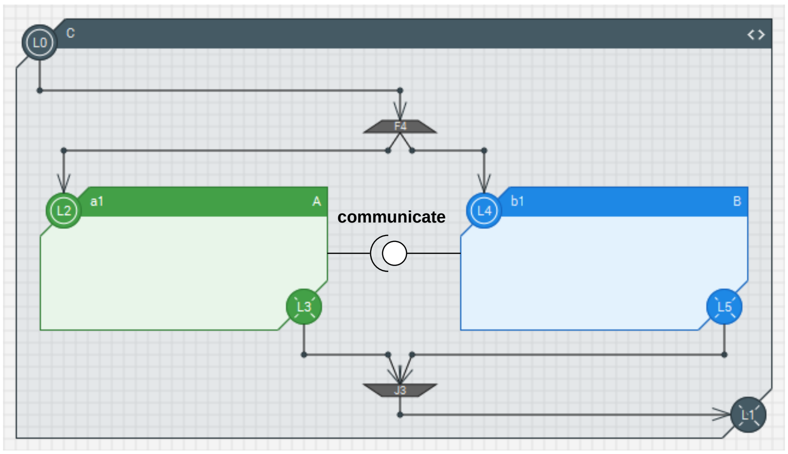


Figure 11.2: Visualizing the $A \xrightarrow{communicate} B$ synchronization.

The idea is to give the modelers an insight in how subcomponents might interact. Based on this, we do not believe that visualizing other channels that component A might synchronize on, would provide any additional information for the modeler, but rather create a more cluttered model. One might even leave it up to the modeler to describe which channels he wants to display, making them function more like an auto-generated comment.

11.7.2 Component Communication

Like how one could introduce this visualization between subcomponents to describe how they interact with each other, one could introduce this feature for a single component, describing how it *might* interact with other components, once used in a subcomponent somewhere in the model. This is visualized in Figure 11.3, where we see that component A can initiate synchronization on the `communicate`-channel, and receive a synchronization on the `status`. We do not know which components that otherwise take part in these synchronizations if any at all. This is very similar to integrated development environments, where we do not know where a particular function is called. One could also indicate if any of these channels are unused in other parts of the model, as a kind of warning, e.g. "*Channel is unused*" (static analysis) or "*Synchronization on channel **complete** is never received*" (dynamic analysis).

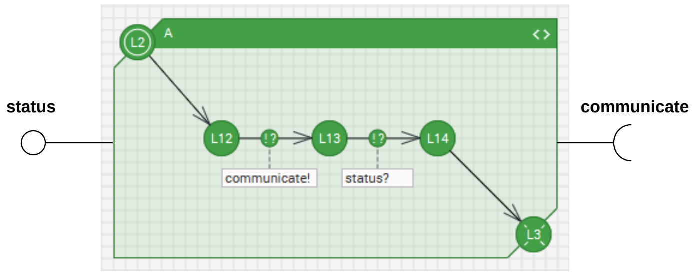


Figure 11.3: Component with synchronizations *communicate!* and *status?*.

11.7.3 Different Channel Types

In UPPAAL, and in effect, also H-UPPAAL, one might declare a channel as either *Urgent*, *Broadcast*, or both. Especially the broadcast trait is important to keep in mind when designing and understanding a model. Based on this, we believe that the visual additions described above should reflect these different types of channels. One might make the channel name italic to describe that the channel is urgent, and underline to describe that it is a broadcast channel. This would adhere to Principle 5 (Printable).

References and Appendices

12 Bibliography

Åkerholm, Mikael, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli (2007). “The SAVE approach to component-based development of vehicular systems”. In: *Journal of Systems and Software* 80.5, pp. 655–667.

Byg, Joakim, Kenneth Yrke Jørgensen, and Jiří Srba (2009). “TAPAAL: Editor, simulator and verifier of timed-arc Petri nets”. In: *International Symposium on Automated Technology for Verification and Analysis*. Springer, pp. 84–89.

David, Alexandre and Kim G Larsen (2011). “More Features in UPPAAL”. In: *Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook*, p. 49.

David, Alexandre and M. Oliver Möller (2001). *From* HUPPAAL *to* UPPAAL*: A Translation from Hierarchical Timed Automata to Flat Timed Automata*. Tech. rep. RS-01-11. BRICS. URL: `http://www.brics.dk/RS/01/11/index.html`.

Facts About Color Blindness. URL: `https://nei.nih.gov/health/color_blindness/facts_about`.

Field, Andy (2009). *Discovering Statistics Using SPSS, 3rd Edition (Introducing Statistical Methods)*. SAGE Publications Ltd. ISBN: 1847879071.

Gerd Behrmann Alexandre David, Kim Guldstrand Larsen (2006). “A Tutorial on Uppaal 4.0”. In:

Håkansson, John, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej (2008). “Component-Based Design and Analysis of Embedded Systems with UPPAAL PORT”. In: *Automated Technology for Verification and Analysis: 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings*. Ed. by Sungdeok (Steve) Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 252–257. ISBN: 978-3-540-88387-6. DOI: `10.1007/978-3-540-88387-6_23`. URL: `http://dx.doi.org/10.1007/978-3-540-88387-6_23`.

Huber, Peter, Kurt Jensen, and Robert M Shapiro (1989). “Hierarchies in coloured Petri nets”. In: *International Conference on Application and Theory of Petri Nets*. Springer, pp. 313–341.

Jensen, Kurt, Lars Michael Kristensen, and Lisa Wells (2007). “Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems”. In: *International Journal on Software Tools for Technology Transfer* 9.3-4, pp. 213–254.

Kaikkonen, Anne, Aki Kekäläinen, Mihael Cankar, Titti Kallio, and Anu Kankainen (2005). “Usability testing of mobile applications: A comparison between laboratory and field testing”. In: *Journal of Usability studies* 1.1, pp. 4–16.

Larsen, Kim G, Paul Pettersson, and Wang Yi (1997). "UPPAAL in a nutshell". In: *International journal on software tools for technology transfer* 1.1-2, pp. 134–152.

Mitta, Deborah A (1991). "A methodology for quantifying expert system usability". In: *Human Factors* 33.2, pp. 233–245.

Molich, Rolf and Joseph S Dumas (2008). "Comparative usability evaluation (CUE-4)". In: *Behaviour & Information Technology* 27.3, pp. 263–281.

Mourtizsen, Niklas Kirk and Rasmus Holm Jensen (2016). *HUPPAAL: Introducing Hierarchies to Networks of Timed Automata - HUPPAAL - a New Integrated Development Environment for Model Checking*.

Nielsen, Jakob (1993). *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 0125184050.

Raptis, Dimitrios, Nikolaos Tselios, Jesper Kjeldskov, and Mikael B. Skov (2013). "Does Size Matter?: Investigating the Impact of Mobile Phone Screen Size on Users' Perceived Usability, Effectiveness and Efficiency." In: *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services*. MobileHCI '13. Munich, Germany: ACM, pp. 127–136. ISBN: 978-1-4503-2273-7. DOI: 10.1145/2493190.2493204. URL: http://doi.acm.org/10.1145/2493190.2493204.

Wyatt, Jeremy and David Spiegelhalter (1991). "Evaluating Medical Expert Systems: What To Test, And How ?" In: *Knowledge Based Systems in Medicine: Methods, Applications and Evaluation: Proceedings of the Workshop "System Engineering in Medicine", Maastricht, March 16–18, 1989*. Ed. by Jan L. Talmon and John Fox. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 274–290. ISBN: 978-3-662-08131-0. DOI: 10.1007/978-3-662-08131-0_22. URL: http://dx.doi.org/10.1007/978-3-662-08131-0_22.

13 Expert User Evaluation Results

This appendix presents results from the expert user evaluation. Here, the letters A-F represents participants.

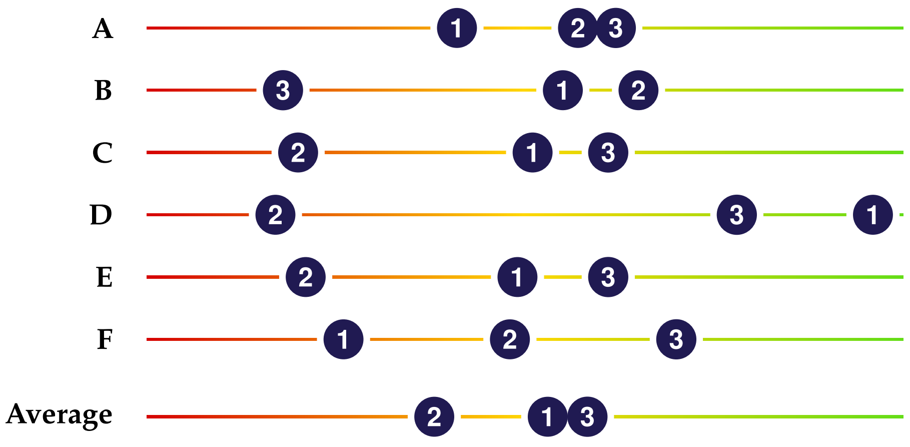


Figure 13.1: Results from table rating task for location urgency representation ideas.

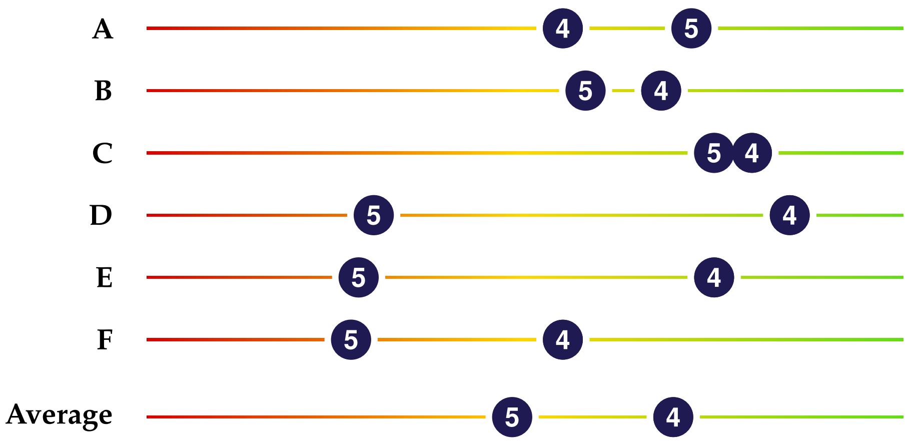


Figure 13.2: Results from table rating task for subcomponent functionality representation ideas.

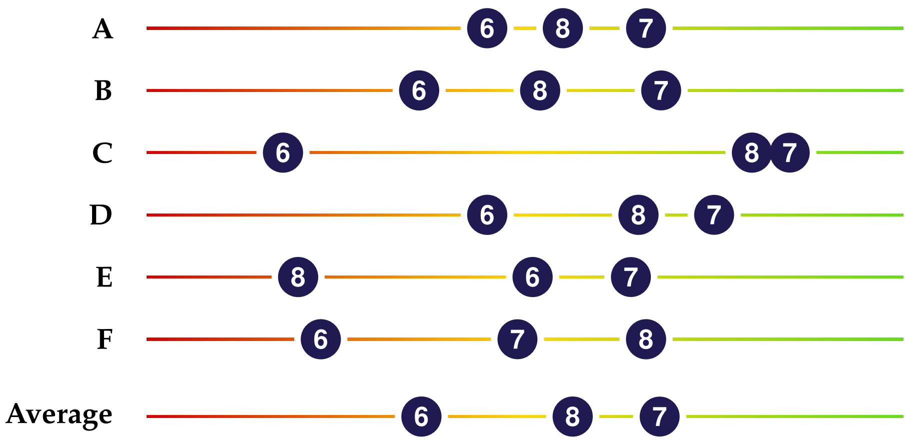


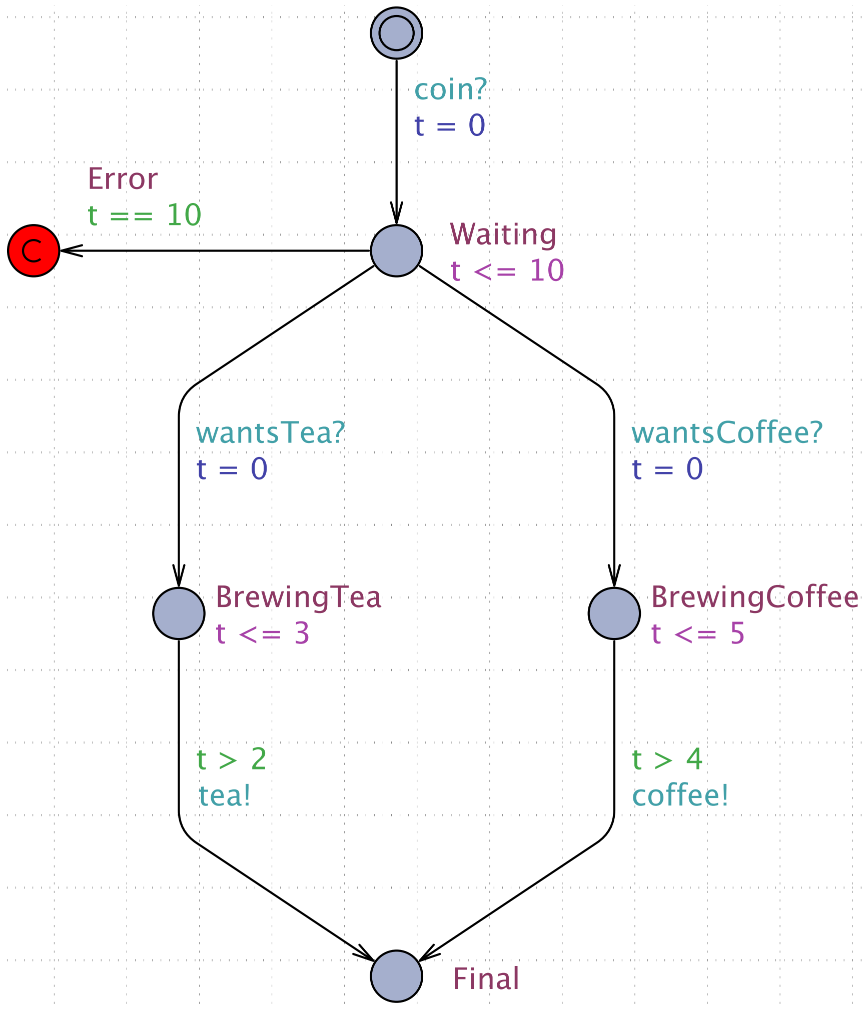
Figure 13.3: Results from table rating task for edge properties representation ideas.

14 Expert User Evaluation Tasks

This appendix lists the tasks the experts of UPPAAL was assigned during the user expert evaulation.

14.1 Task: Draw a model

Draw this model in the H-UPPAAL tool.



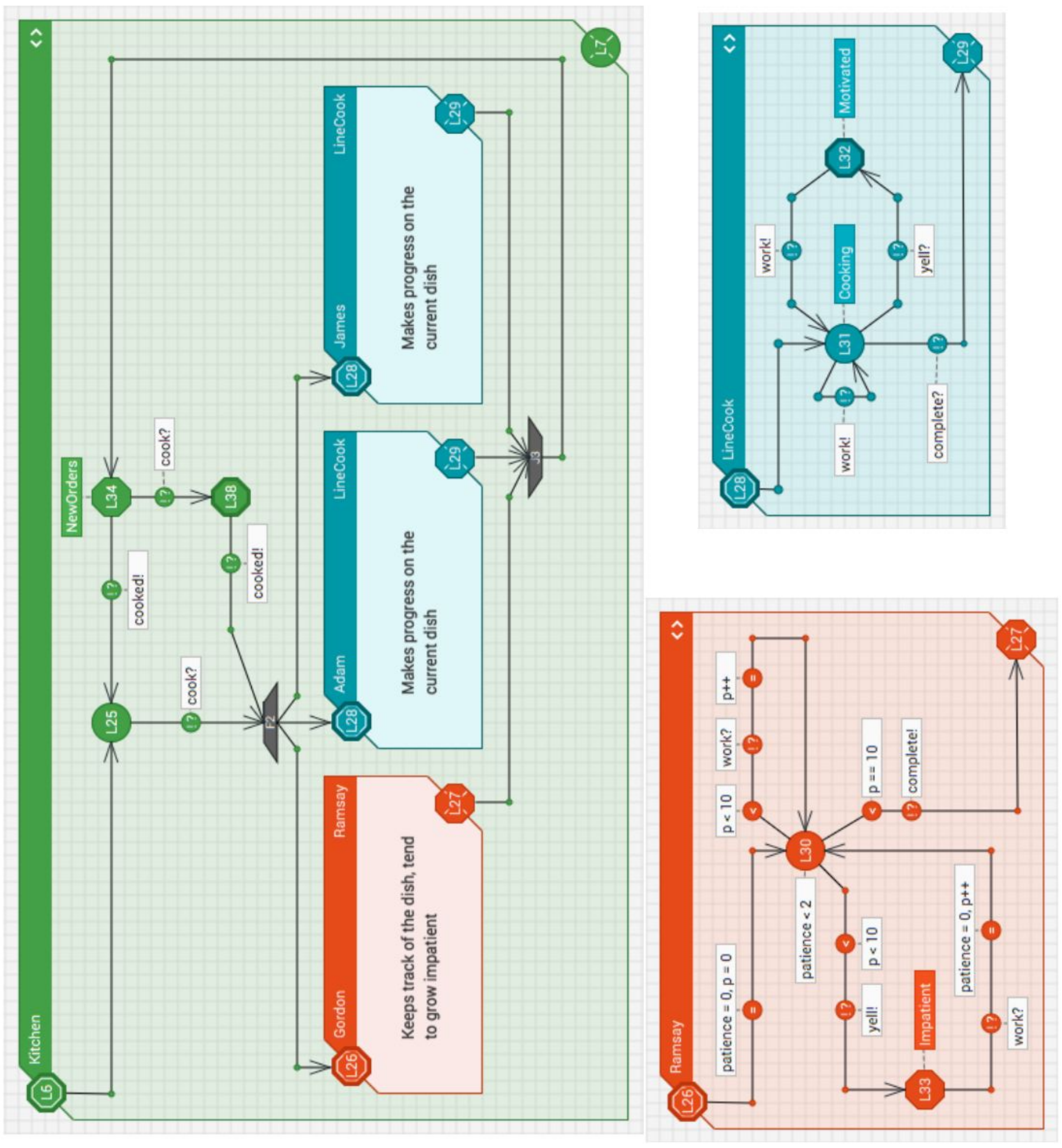
14.2 Task: Expand a model

Expand the university model. Here we want the computer scientist to use the vending machine to retrieve hot beverages in exchange for coins.

The participants were presented with a model created priorly in the tool, and were asked to extend it as described.

14.3 Task: Explain a model

Please describe the following model out loud in as much detail as possible.



14.4 Task: Rank the different proposals

Please rank the following proposals in regards to how much you like them on the special ranking-table. The arrow on the paper indicates the position on the table.

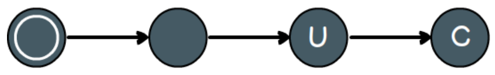
This correspond to the proposals found in Appendix 15

15 Design Ideas

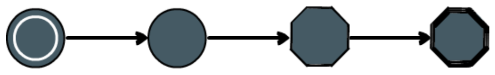
This appendix contains different design ideas that have been produced through this semester and the last semester, documented in Mourtizsen and Jensen, 2016.

15.1 Location Urgency

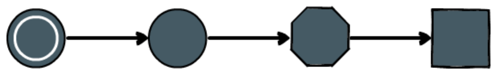
Design Idea 1 – Traditional UPPAAL Urgency.



Design Idea 2 – New Urgency with Octagon Shape.

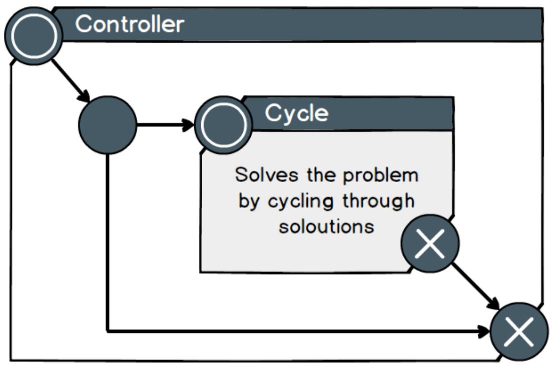


Design Idea 3 – New Urgency with Octagon and Square Shapes.

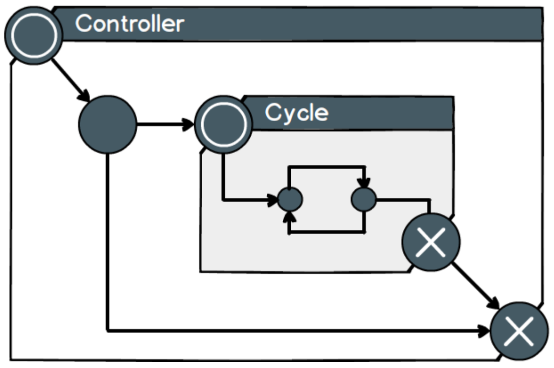


15.2 Subcomponent Functionality Representation

Design Idea 4 – Textual Representation.

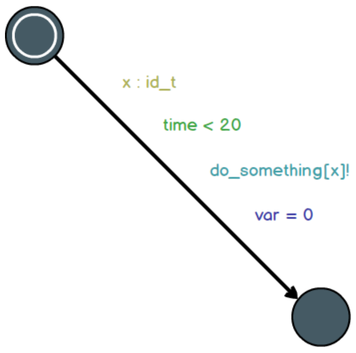


Design Idea 5 – Miniature Model Representation.

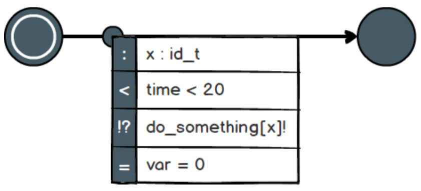


15.3 Edge Properties

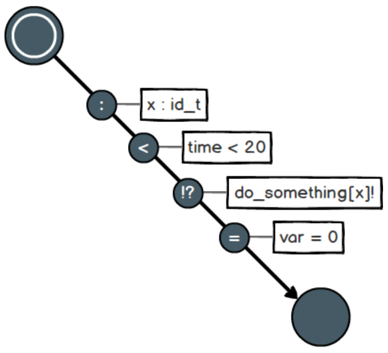
Design Idea 6 – Traditional UPPAAL Edge with Properties.



Design Idea 7 – Single Box with Properties.



Design Idea 8 – Multiple Boxes with Single Property.



16 Performance Task Sheets

The following pages will show the task sheets that were handed to the eighth participant during the performance test execution. This particular participant were assigned with the order A, C, B. Using UPPAAL in session 1 and H-UPPAAL in session 2.

SESSION 1

PARTICIPANT 8

UPPAAL

AALBORG UNIVERSITET

A08

1 Finding the Best Scheduler

Your team of experienced CPU developers have recently been more and more interested in model checking. This has caused your team to develop a model for different schedulers which you plan to use in the next line of CPUs. However, before beginning production, you want to evaluate the performance of these scheduling techniques. To do this, your team implemented a UPPAAL model based on the following description:

Project Description

We want to model a computer, where arbitrarily ordered *tasks* with different priority should be executed by a CPU. This CPU can only execute a single task at a time. However, to determine when to execute the different tasks, the CPU may utilize a scheduling technique. Our goal is to find the best possible technique.

A good scheduling technique is one where few tasks are waiting on being executed and high priority tasks are executed before ones with lower priority.

Your colleague informs you that three different scheduling techniques have been implemented: **Stack**, **Queue**, and, **Priority Queue**. He also informs you that you may evaluate a particular technique using the **score** variable in the **Evaluator** template, where values closer to 0 are considered better.

A) Start **UPPAAL** and open the `/uppaal/cpu/cpu.xml` file.

B) Find the worst case **score** after the **TestSuite** is completed.

Stack: ____________________

Queue: ____________________

Priority Queue: ____________________

C) Find the best case **score** after the **TestSuite** is completed.

Stack: ____________________

Queue: ____________________

Priority Queue: ____________________

D) Save the model, close the program, and inform the test conductors that you finished exercise `A8`.

Tip!

It is possible to find **infimum** (smallest possible value) and **supremum** (largest possible value) of a variable or clock x in location l using the following queries:

```
inf{l} :  x
sup{l} :  x
```

C08

2 Fibonacci

A) Start UPPAAL and open the `/uppaal/fib/fib.xml` file (this project will be empty, but load it anyway).

B) Create a new template where the following query should pass when `x` is found in the first 20 Fibonacci numbers and fail otherwise.

```
E<> fib == x
```

Tip!
The Fibonacci sequence is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

where $F_0 = 0$, and $F_1 = 1$.

C) Indicate with a ✗ how your model reacts to the following values of `x`.

x	Pass	Fail
3		
11		
89		
377		
7127		
10946		

D) Save the model, close the program, and inform the test conductors that you finished exercise C8.

B08

3 Kitchen Modeling

Your team of experienced model checkers has recently been working with the following project description for Gordon Ramsay's new restaurant, Amaryllis. The Amaryllis is a restaurant where Gordon Ramsay is the chef. The concept of his restaurant is simple: you place an order, and Gordon will surprise you with one of his famous dishes. To help him in the restaurant, Gordon has hired the **waitress** Sofia, and two **line cooks**, Adam and James.

Your colleagues have already finished a big part of the project, but are still missing the modeling of Gordon Ramsay himself.

> **Gordon Ramsay**
>
> To prepare the dishes, Gordon has hired two line cooks, Adam and James. They are good workers, but sometimes they require additional motivation in order to progress on a dish. All of the dishes designed by Gordon consists of exactly **10 steps**. Gordon also knows that none of the steps should take longer than **2 minutes** to perform, causing Gordon to grow impatient after **2 minutes**, resulting in him yelling at his cooks.

Model the Chef Gordon Ramsay

A) Start UPPAAL and open the `/uppaal/amaryllis/amaryllis.xml` file.

B) Model the behavior of Gordon Ramsay using a new template and add him to the **kitchen** personnel.

> **Tip!**
>
> Familiarize yourself with the work done. Investigate how the **LineCook** templates are started. Also, consider how the line cooks are instructed to prepare the meals for Gordon.

C) Save the model, close the program, and inform the test conductors that you finished exercise B8.

SESSION 2

PARTICIPANT 8

H-UPPAAL

AALBORG UNIVERSITET

A08

1 Finding the Best Scheduler

Your team of experienced CPU developers have recently been more and more interested in model checking. This has caused your team to develop a model for different schedulers which you plan to use in the next line of CPUs. However, before beginning production, you want to evaluate the performance of these scheduling techniques. To do this, your team implemented a H-UPPAAL model based on the following description:

Project Description

We want to model a computer, where arbitrarily ordered *tasks* with different priority should be executed by a CPU. This CPU can only execute a single task at a time. However, to determine when to execute the different tasks, the CPU may utilize a scheduling technique. Our goal is to find the best possible technique.

A good scheduling technique is one where few tasks are waiting on being executed and high priority tasks are executed before ones with lower priority.

Your colleague informs you that three different scheduling techniques have been implemented: **Stack**, **Queue**, and, **Priority Queue**. He also informs you that you may evaluate a particular technique using the **score** variable in the **Evaluator** component, where values closer to 0 are considered better.

A) Start **H-UPPAAL** and open the `/huppaal/cpu/` folder.

B) Find the worst case **score** after the **TestSuite** is completed.

Stack:

Queue:

Priority Queue:

C) Find the best case **score** after the **TestSuite** is completed.

Stack:

Queue:

Priority Queue:

D) Save the model, close the program, and inform the test conductors that you finished exercise `A8`.

Tip!

It is possible to find **infimum** (smallest possible value) and **supremum** (largest possible value) of a variable or clock x in location l using the following queries:

```
inf{l} :  x
sup{l} :  x
```

C08

2 Fibonacci

A) Start **H-UPPAAL** and open the `/huppaal/fib/` folder (this project will be empty, but load it anyway).

B) Create a new component where the following query should pass when `x` is found in the first 20 Fibonacci numbers and fail otherwise.

```
E<> fib == x
```

Tip!
The Fibonacci sequence is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

where $F_0 = 0$, and $F_1 = 1$.

C) Indicate with a ✗ how your model reacts to the following values of `x`.

x	**Pass**	**Fail**
3	☐	☐
11	☐	☐
89	☐	☐
377	☐	☐
7127	☐	☐
10946	☐	☐

D) Save the model, close the program, and inform the test conductors that you finished exercise `C8`.

B08

3 Kitchen Modeling

Your team of experienced model checkers has recently been working with the following project description for Gordon Ramsay's new restaurant, Amaryllis. The Amaryllis is a restaurant where Gordon Ramsay is the chef. The concept of his restaurant is simple: you place an order, and Gordon will surprise you with one of his famous dishes. To help him in the restaurant, Gordon has hired the **waitress** Sofia, and two **line cooks**, Adam and James.

Your colleagues have already finished a big part of the project, but are still missing the modeling of Gordon Ramsay himself.

Gordon Ramsay

To prepare the dishes, Gordon has hired two line cooks, Adam and James. They are good workers, but sometimes they require additional motivation in order to progress on a dish. All of the dishes designed by Gordon consists of exactly **10 steps**. Gordon also knows that none of the steps should take longer than **2 minutes** to perform, causing Gordon to grow impatient after **2 minutes**, resulting in him yelling at his cooks.

Model the Chef Gordon Ramsay

A) Start **H-UPPAAL** and open the `/huppaal/amaryllis/` folder.

B) Model the behavior of Gordon Ramsay using a new component and add him to the **kitchen** personnel.

Tip!

Familiarize yourself with the work done. Find where the **Gordon Ramsay** subcomponent should be started in parallel. Also, consider how the line cooks are instructed to prepare the meals for Gordon.

C) Save the model, close the program, and inform the test conductors that you finished exercise B8.