INTRODUCING HIERARCHIES TO NETWORKS OF TIMED AUTOMATA

H-UPPAAL

A NEW INTEGRATED DEVELOPMENT ENVIRONMENT FOR MODEL CHECKING

BY NIKLAS KIRK MOURITZSEN & RASMUS HOLM JENSEN

Contents

1 Preface **7**
1.1 Reading Guide 7
1.2 Terminology 7

I Introduction

2 Model Checking in UPPAAL **15**
2.1 Introducing the UPPAAL Tool 15
2.2 Tool Overview 16

3 UPPAAL Issues **19**

4 Scope of Project **23**
4.1 Why Hierarchies? 23
4.2 Integrated Development Environment 24
4.3 Delimitation 25

II H-UPPAAL

5 The H-UPPAAL Manifesto **29**

6 Formalism **31**
6.1 Notation 31
6.2 Initial Formalism 32

7 Design **35**
7.1 Location 36
7.2 Edges 38
7.3 Components 40
7.4 Subprocedures 41
7.5 Setting the Layout for a New IDE 42

8 Showcasing the H-UPPAAL Tool **45**
8.1 Context Menus 46
8.2 Query Pane 47
8.3 Displaying Errors 48
8.4 Declarations 48
8.5 Making Models 50
8.6 Storing Models 52

9 Flattening **55**
9.1 From Components to Templates 55
9.2 Execution of Subprocedures 57

10 Queries **61**
10.1 Single Subcomponent Queries 62
10.2 Multiple Subcomponent Queries 62
10.3 Long Queries 63

Final Thoughts

11 Evaluation **67**
11.1 Working with H-UPPAAL 67
11.2 Adhering to the Manifesto 73
11.3 Thoughts on the Development 74
11.4 Involving Users 75

12 Conclusion **77**

13 Further Work **79**
13.1 Error and Warnings 79
13.2 Periodic Queries 80
13.3 Version Control of JSON-files 80
13.4 H-UPPAAL Specific Queries 80
13.5 Printability 81
13.6 Refactoring 82
13.7 Utilize the UPPAAL Engine 83

References and Appendices

14 Bibliography **87**

15 Full JSON Example **89**

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
P9, fall semester 2016

Project group
DES906E16

Authors
Niklas Kirk Mouritzsen
nmouri12@student.aau.dk

Rasmus Holm Jensen
rhje12@student.aau.dk

Project supervisor
Ulrik Nyman
ulrik@cs.aau.dk

Number of Pages: 94
Number of Appendix Pages: 6
Published: 13th of January, 2017

Abstract

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. This project is focused on developing a tool that introduces mechanisms found in hierarchical modeling in order to achieve a greater modularity, better encapsulation, and, in general, a higher abstraction in a model. Furthermore, this project also considers how such a tool could be inspired by features found in modern integrated development environments.

This report presents the model checking tool H-UPPAAL which realizes some of these concepts by utilizing the UPPAAL verification engine in a new front end, inspired by IDEs. H-UPPAAL creates a foundation for investigating if hierarchies and IDE-inspired features can add value to users in the domain of model checking.

The substance of the report may only be published (with references) in agreement with the authors.

1 Preface

This report is written by project group DES906E16 from the Software Master's programme at Aalborg University. This the report concludes the semester spanning from September 2016 to January 2017. 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.

1.1 Reading Guide

Throughout the report, personal pronouns refer to the authors of the report. The contents 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 88.

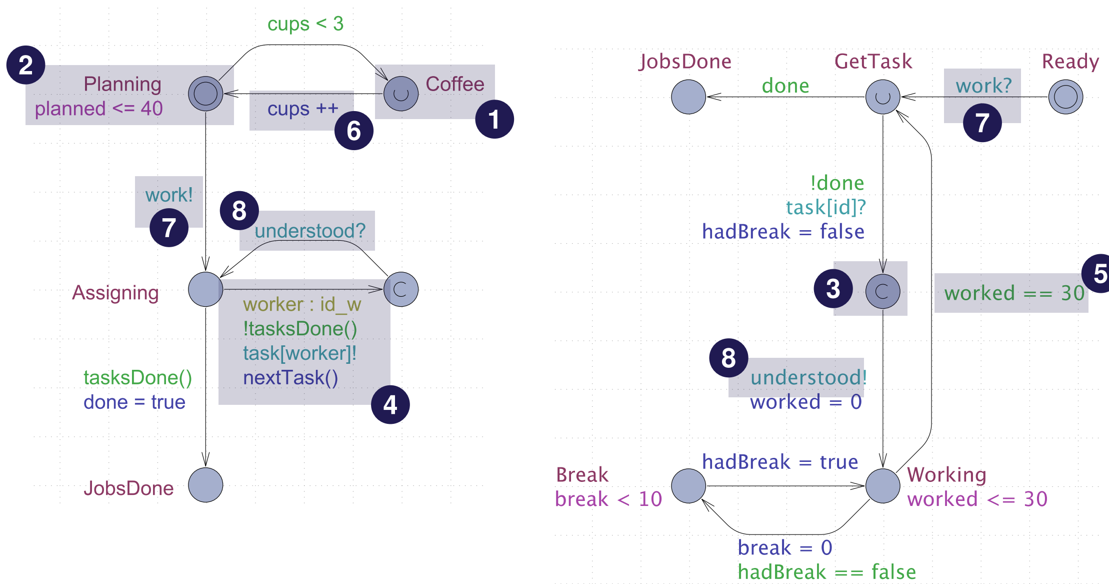
1.2 Terminology

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.

```
1  const int N = 120; // # tasks
2  const int W = 3; // # workers
3
4  typedef int[0,N-1] id_t;
5  typedef int[0,W-1] id_w;
6
7  bool done;
8
9  broadcast chan work;
10 chan task[W], understood;
```

Listing (1.1) Global declarations.

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

Listing (1.2) Worker declarations.

```
1  const id_w id
```

Listing (1.3) Worker parameters.

```
1  clock planned;
2  int[0,3] cups = 0;
3  id_t nextUp = 0;
4
5  // Function for finding the next task
6  void nextTask() {
7      if(nextUp < N - 1) {
8          nextUp = nextUp + 1;
9      }
10 }
11
12 // Function telling if the job is done
13 // Prevents out of bounds error
14 bool tasksDone() {
15     return nextUp == N - 1;
16 }
```

Listing (1.4) Boss declarations.

```
1  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 defined from 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 on 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 UPPAAL returns `true`, a trace 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**-statements 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 transitions that include 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.

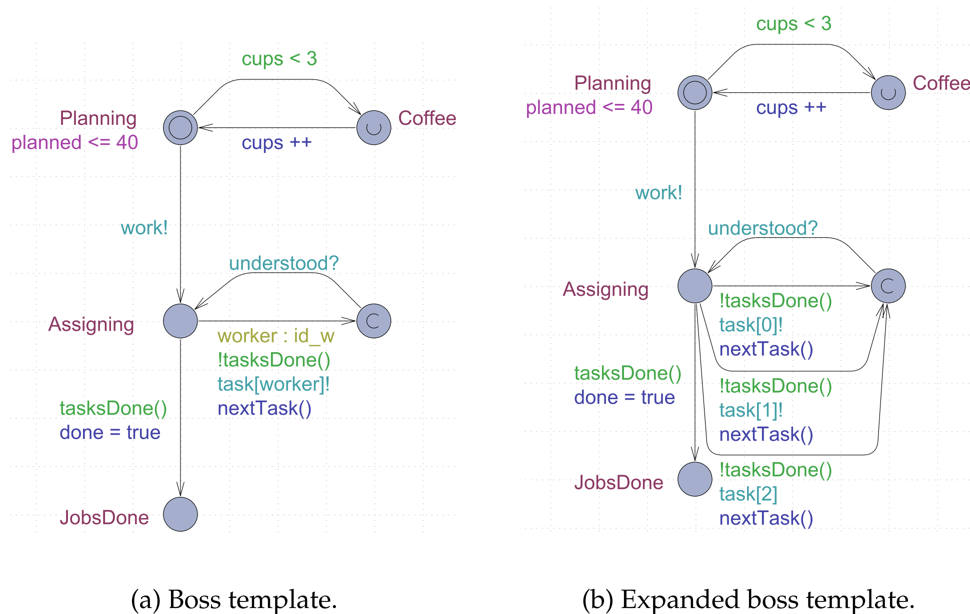
Guard
A boolean expression, often based on variables and clocks, which must evaluate to true for the **edge** to be active.

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 to. 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**.

Introduction

2 Model Checking in UPPAAL **15**
2.1 Introducing the UPPAAL Tool
2.2 Tool Overview

3 UPPAAL Issues **19**

4 Scope of Project **23**
4.1 Why Hierarchies?
4.2 Integrated Development Environment
4.3 Delimitation

2 Model Checking in UPPAAL

Modeling has for decades been a key activity in software development, which is concerned with representing a complex system in a condensed manner. This is, in particular, useful as a communicative technique when designing and implementing information systems. Models can in other words describe how a system behaves, and which states a system can be in. In more recent years, modeling is not only used as a media for communication, but is now also used to perform model checking [Clarke, 2008]. 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. A problem in this domain, is that the state space, represented by the models, becomes so big or even infinite that it is infeasible to verify certain properties. To capture more complex systems, attributes such as time are often desired, which increases the state space further. This require modern model checking tools to handle these issues, so that one can model a complex system while being able to verify properties of said system.

2.1 Introducing the UPPAAL Tool

There exists various tools for model checking, one of the more mature ones is UPPAAL, which is being developed in both the *Department of Computer Science* at Aalborg University in Denmark, and the *Department of Information Technology* at Uppsala University in Sweden. UPPAAL is a tool that allows modeling of systems using networks of timed automata. The tool has a very efficient verification engine that handles timely properties quite well, and is able to handle an infinite state-space by using a symbolic state representation. The tool accelerates when modeling systems consisting of a collection of non-deterministic processes using a finite control structure. It uses real-valued clocks and introduces data types such as bounded integers, arrays, etc. Furthermore it has support for traces and simulation of various verification queries, assisting users in designing models that adhere to certain properties. The train gate example from the UPPAAL tutorial [Behrmann et al., 2004] seen in Figure 2.1 shows how small and simple models, that express a rather complex system, are easily created using the formalism of UPPAAL. More information about the tool is available at:

`http://www.it.uu.se/research/group/darts/uppaal/about.shtml`

Currently, UPPAAL is mostly used within the research domain, with interesting work being made on tools such as UPPAAL STRATEGO[1], UPPAAL CORA[2], UPPAAL TRON[3], and UPPAAL SMC[4] just to name a few. All of these different versions of UPPAAL introduce new angles to the classical model checking. One example is UPPAAL SMC, which is

[1] `http://people.cs.aau.dk/~marius/stratego/`

[2] `http://people.cs.aau.dk/~adavid/cora/`

[3] `http://people.cs.aau.dk/~marius/tron/`

[4] `http://people.cs.aau.dk/~adavid/smc/`

concerned with modeling probabilities for specific actions in an environment. This is especially useful when modeling, and verifying, areas such as communication protocols.

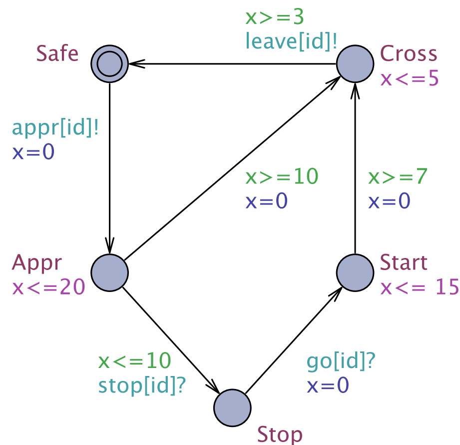


Figure 2.1: The UPPAAL train gate example.

UPPAAL has also seen some use in the industry with companies such as *Bang & Olufsen* and *Philips* taking interest in the tool. For more examples of industrial use-cases and case studies, see:

`http://uppaal.com/index.php?sida=203&rubrik=92`

2.2 Tool Overview

This section gives an overview of the UPPAAL tool and its core capabilities, using Example 1. A detailed overview of terms and concepts used in UPPAAL can be found in Section 1.2.1. For deeper insight and additional examples, we invite the reader to take a look at "A Tutorial on Uppaal 4.0" [Gerd Behrmann, 2006]. Here, the authors define and explain all the different concepts of the tool.

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.

When modeling in UPPAAL, the first part of this tool you will meet is template editor as seen in Figure 2.2. The *Editor* tab, allows the users to design the network of timed automata by creating templates, managing declarations and parameters. Figure 2.2 visualizes how a worker from the Example 1 could be modeled in the tool. Changing how the workers communicate with the boss, or changing how long a break they are permitted, is done here.

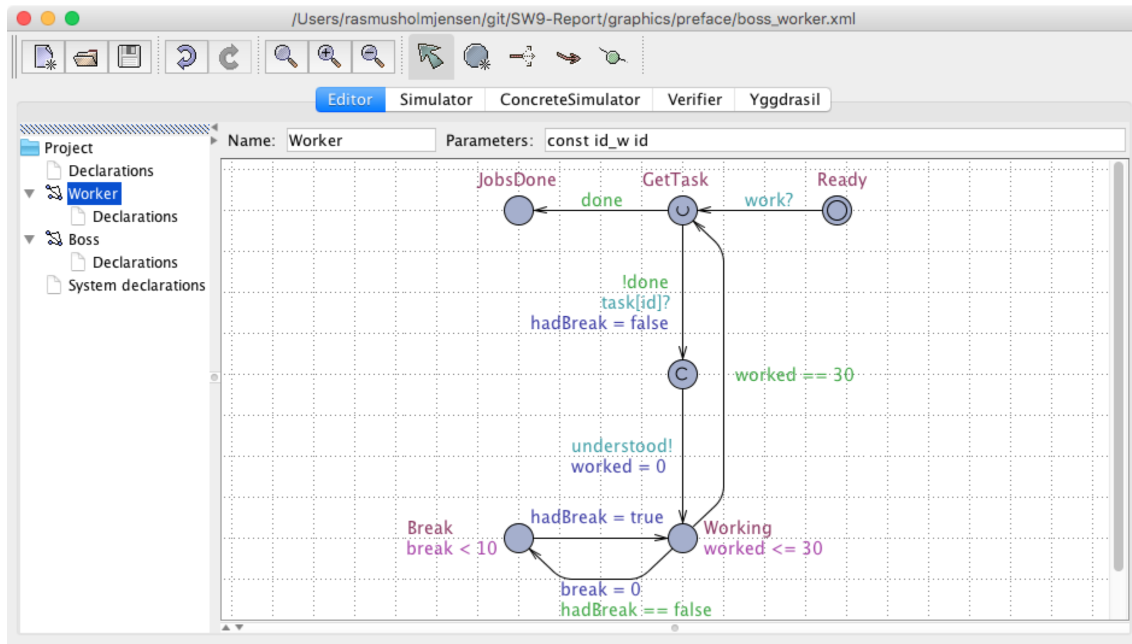


Figure 2.2: The *Editor* tab.

The *Verifier* tab is where the users interact with the verification engine. This tab allows users to add queries and run them against the model they designed in the *Editor* tab. It is this part of the tool, that enables users to verify properties of the system in question, like figuring out if the boss from Example 1 is guaranteed to get the job done. This question could be answered by running the query `A<> Boss.JobsDone`, as seen in Figure 2.3. This figure also show that we have indicators stating which queries passed, alongside a log with results and details regarding what resources went into running the query.

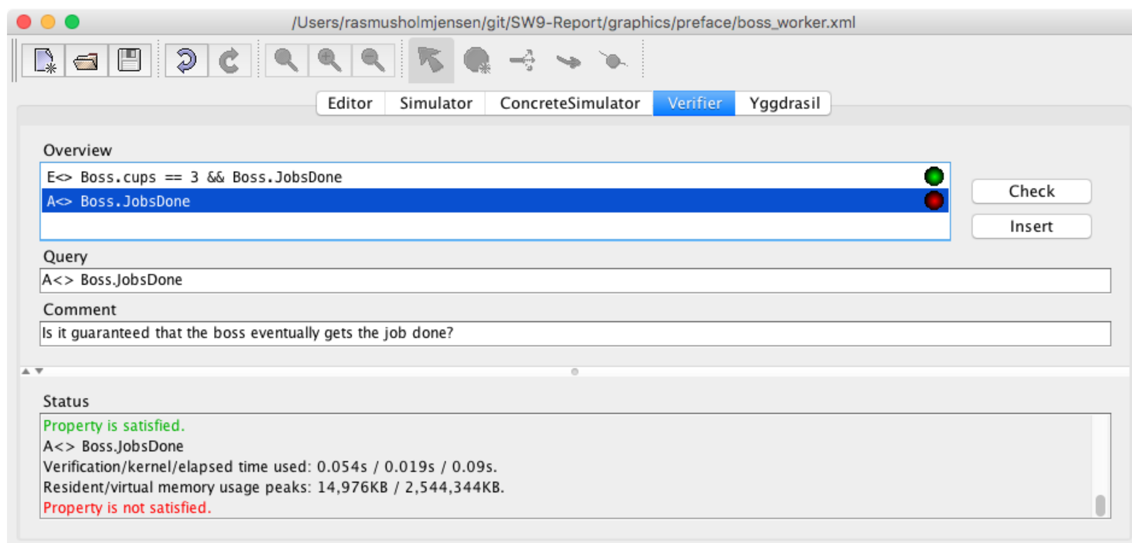


Figure 2.3: The *Verifier* tab.

However, the *Verifier* tab only gives us boolean answers and does not help the users to figuring out why a particular property does not hold. For this purpose we can use the *Simulator* tab. If we run the query `E<> Boss.cups == 3 && Boss.JobsDone`, this tab can

visualize trace of how the system can reach this property as seen in Figure 2.4. This tab can in general be used to explore and understand the symbolic states and behavior of the model, and, in combination with the verifier, is in some cases able to produce proofs or contradictions for various properties.

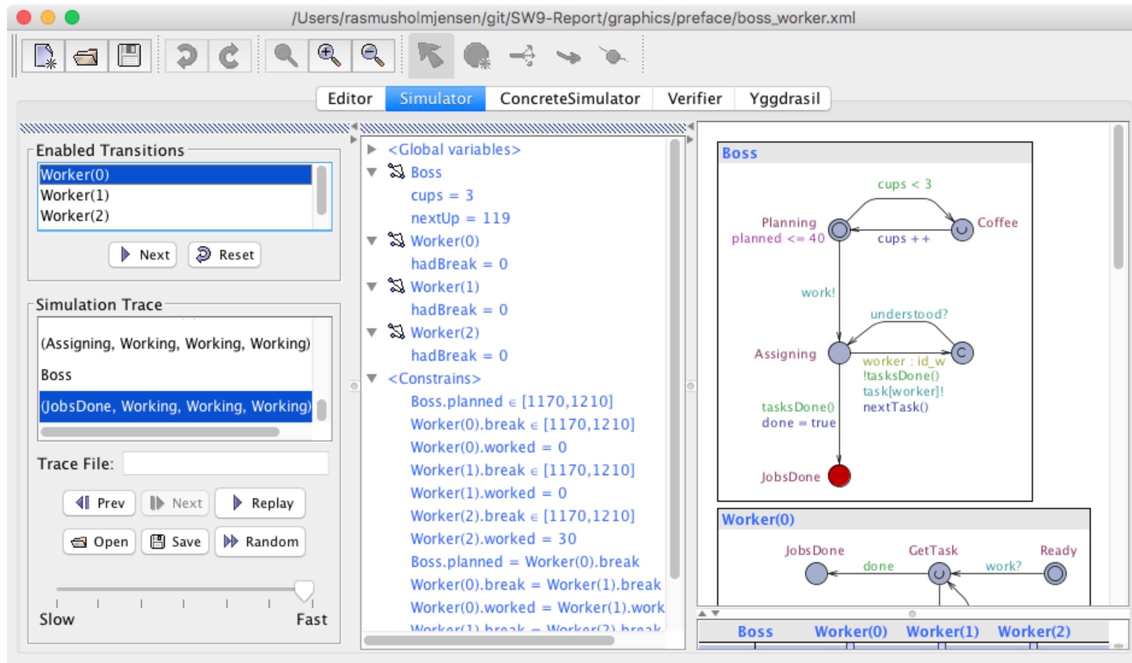


Figure 2.4: The *Simulator* tab.

3 UPPAAL Issues

UPPAAL is a powerful model checking tool and contain many useful and interesting features. We would like to investigate if there is space for improvement in such a tool. This chapter will, through an example based on "Compositional Schedulability Analysis of An Avionics System Using UPPAAL" [Boudjadar et al., 2014], show some issues with the current version of UPPAAL. This paper uses the tool to produce evidence for their findings, and they present the model seen in Figure 3.1 to explain how they came to these findings. Figure 3.1 shows a complex template, which illustrates a common use case of the tool, and is far from the most complex model we have seen in scientific papers. This chapter, will through this example, explain issues and concerns regarding model checking using UPPAAL.

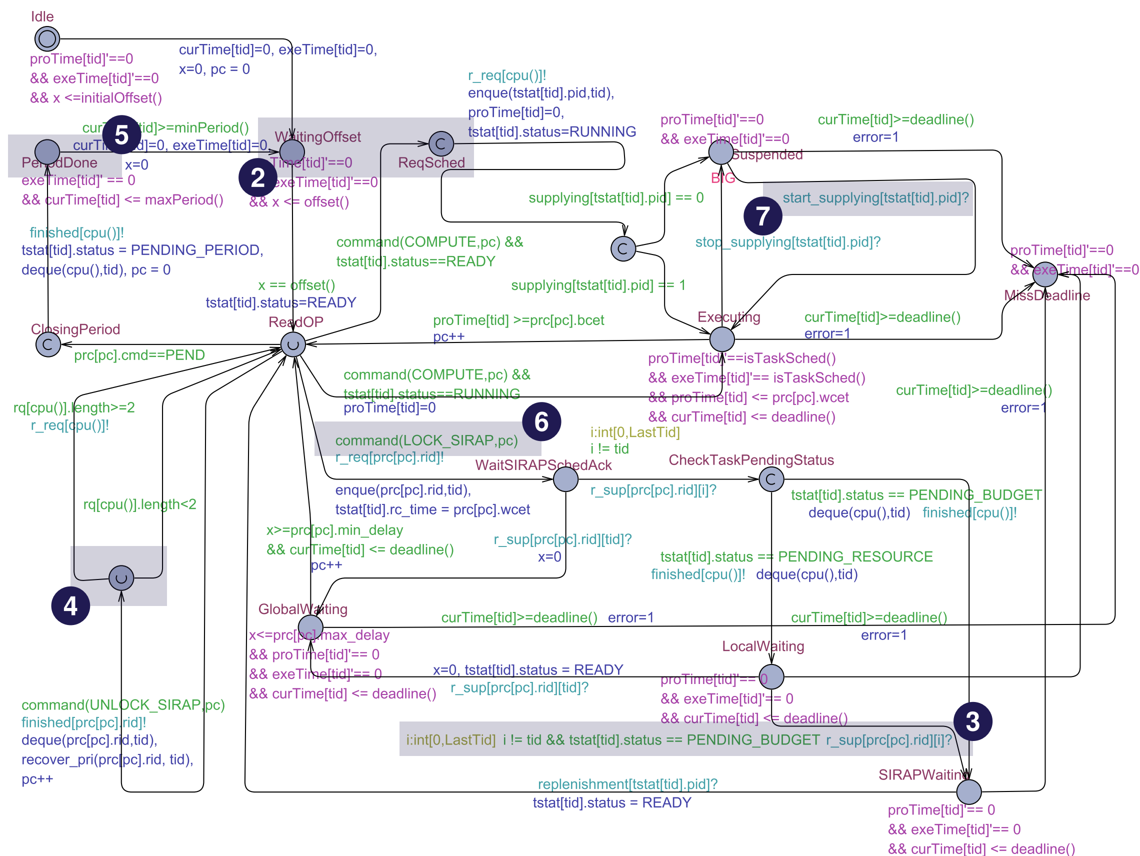


Figure 3.1: UPPAAL model for analyzing scheduling strategies [Boudjadar et al., 2014].

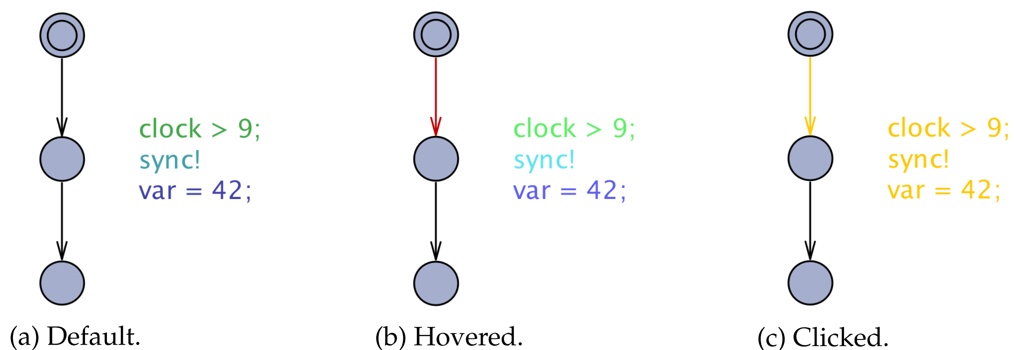
Issue 1 – Limited Screen Real Estate. Screen real estate is limited, and varies greatly depending on computer setup. This, combined with a complex model, makes it hard to fit large and complex templates on the screen. To achieve this, modelers tend to cramp information into a tight space, compromising overview and reproducibility.

As seen in Figure 3.1, it is clear that the modelers were limited on space, especially considering that the model would also have to be presented in a paper. Even though this example is far from the most complex model we have seen, it might better show a normal use-case of the tool.

Issue 2 – Movable properties. Properties for locations and edges can be moved around in the template. This is to allow the modeler to place elements in such a way that he, and other readers, find the model intuitive and easy to read. However, this may cause some issues because people tend to understand objects by their relation to other objects [Benyon, 2010, pp.637–639]. If the modeler does not handle this with care, the model may become less readable, e.g. a property located closely to two different edges.

Figure 3.1 at (2) shows that properties can be moved to the modelers liking, hence the relation between the labels and their respective locations is not consistent in respect to position. `WaitingOffset` has its label placed above, where `ReqSched` has it label placed below. This type of freedom can make it harder to read the model, because extra effort goes into mapping related objects together.

UPPAAL tries to address this problem, as seen in Figure 3.2, by giving the modeler some visual cues of where properties belong. Visual cues are given both when hovering (Figure 3.2b) and clicking (Figure 3.2c) an edge, which assists the modeler in mapping the properties to their edge. However, these visual cues are not preserved when the model is printed. This corresponds to not interacting with the edge (Figure 3.2a).



(a) Default. (b) Hovered. (c) Clicked.

Figure 3.2: Visual cue of relation between edge and its properties in UPPAAL.

Issue 3 – Ordering of Properties. Properties for edges do not have an enforced ordering. Currently, UPPAAL has an implicit ordering of properties for an edge, this ordering is implied in the properties pane as:

$$Select \rightarrow Guard \rightarrow Synchronization \rightarrow Update$$

Even though UPPAAL colors these four different properties to disjunct them, it seem to be a problem that there are no strict ordering of these properties. This problem is another inherent problem from Issue 2 (Movable properties), where these properties can be harder to read because we identify object in relation to each other.

This issue can be seen at 3 , where the modelers have the different properties (*Select*, *Guard*, and, *Synchronization*) in a horizontal fashion, rather than vertical as of other parts of the model.

Issue 4 – Anonymous Locations. Sharing knowledge regarding a model requires that members of the conversation are aware of the subject of the conversation. When communicating information regarding UPPAAL models, this can sometimes prove to be difficult, due to the fact that locations can have little to no identity at all besides their position in the model.

At 4 we see a location that has no label. This particular location is not directly used to describe a state in the system, but is instead used to update variables. When one wants to pass on knowledge about this part of the model, it becomes rather difficult to infer this particular location. As of this model the identity of this could be: "the urgent location in the bottom left corner".

Issue 5 – Overlapping Objects. Overlapping objects cause information to be hidden. In UPPAAL it is possible to place objects on top of each other, possibly hiding information. It can in many cases disrupt the readability of the model, and in some cases even disrupt the understanding of the model entirely.

Even though no objects in Figure 3.1 are completely hidden, there are some problems in respect to this issue. At 5 we see that when objects are allowed to overlap text, like labels, it can be hard to read, as in the label for the `PeriodDone` location.

Issue 6 – Forgotten Declarations. Declarations are a core part of a UPPAAL model, however, more often that not they are left out of scientific papers, leaving the readers to guess the implementation of functions and types. If the authors do not provide the model in its entirety, it can to some extend disrupt the scientific approach of the article because it can be close to impossible to recreate the experiment. In other words the reproducibility of the results in articles are, for this reason, weakened.

At 6 , the guard of the edge calls a function `command(COMPUTE, pc)` that returns a boolean. Even though this function is explained in the article, the exact implementation is not available from the article. This is sometimes solved by authors hosting the model file for readers to download. However, it seems that this approach often results in dead links or no links at all. The article does to some extend rely on the readers ability to reproduce this function on their own, decreasing the reproducibility. However a trend in publications

of scientific papers is that the authors can attach additional files when publishing, so this issue might not persist in the future.

> **Issue 7 – Hidden Channel Information.** In UPPAAL we work with a network of timed automata to model systems. The communication between the automata uses channels. In complex systems where there are extensive communication between automata, it can be hard to figure out how these automata are linked up. Besides this, UPPAAL also allow for various types of channels namely handshake (regular) and broadcast. Furthermore, channels can be urgent, where no time may pass if synchronization is possible. None of this information is available in the visual representation of the model. This is somewhat related to Issue 6 (Forgotten Declarations).

If we take a look at (7), we see the synchronization on the channel `start_supplying[tstat[tid].pid]`. This is the receiving end of the channel hence the `?` at the end of the channel name. To completely understand this synchronization, we will have to find corresponding templates where a sending version (ending with `!`) is used. After this we would have to go to the declarations (if they are available) to get the type of channel and the urgency of the channel, to get the full description. Disregarding how automata are linked, some information regarding this is simply not shown in the visual representation.

4 Scope of Project

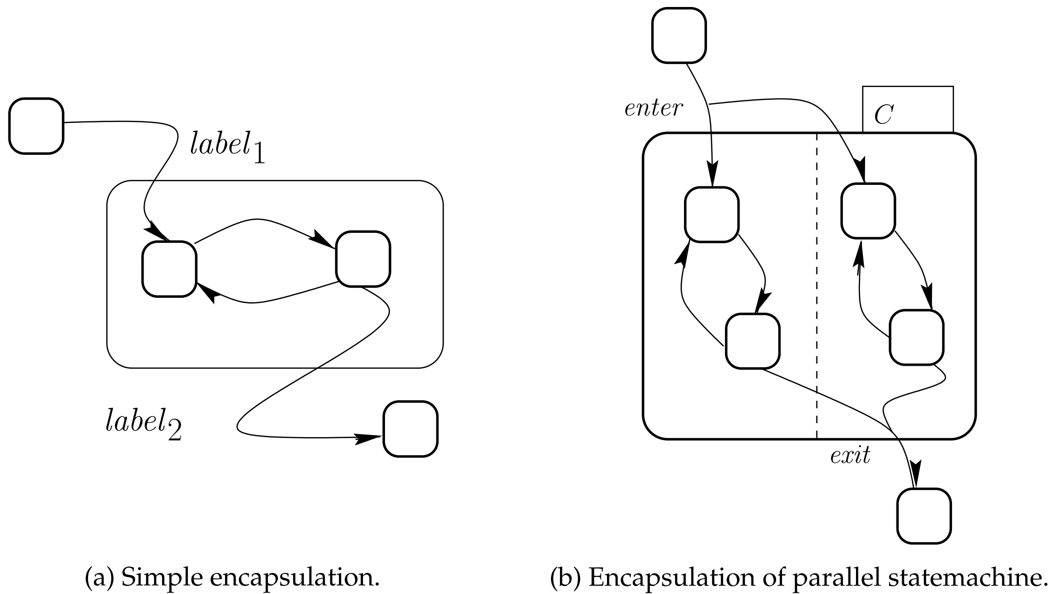
The overall goal of this project is to explore the possibility for creating additional value for users that are interested in model checking and tools like UPPAAL. There might be several ways of achieving this, but what we intend to do, is to explore the possibility for creating a tool that allows for creating hierarchical models in an environment that facilitates automation of mundane or tedious tasks. We propose a new tool that makes use of the UPPAAL engine, but tries to include a new way of modeling systems. This project focuses on investigating the idea of a new front end merging the idea of components in networks of timed automata, while trying to shift the new model checking tool towards a more modern development environment. We would like to add value for existing users of UPPAAL, mostly students and scientist, however, a new version of the tool should also consider the industry and developers in general as a potential audience.

4.1 Why Hierarchies?

We propose hierarchies as a way of solving some of the issues pointed out in Chapter 3. By introducing components we hope to improve the level of abstraction when creating models. Components can help encapsulate details in a black box, where these details are secondary for understanding the overall purpose of the model. Furthermore, a hierarchical structure will also allow for users to design models with higher decomposability. This allows for easier separation and replacement of functionality, alongside a clearer view of responsibility for any given part of the model. In general by introducing hierarchies, we see the potential for including common software development practices, like the use of design patterns and refactoring.

4.1.1 Previous Work

A hierarchical version of timed automata is not a novice idea and was introduced in *From HUPPAAL to UPPAAL: A Translation from Hierarchical Timed Automata to Flat Timed Automata* [David and Möller, 2001]. This paper is greatly inspired by state charts [Harel, 1987], and the way these describe complex information system with concurrent elements. An example of this suggested language can be seen in Figure 4.1, which shows that we can hide some complexity in components (Figure 4.1a), but also the idea that we can have two internal components to run in parallel (Figure 4.1b).



(a) Simple encapsulation.

(b) Encapsulation of parallel statemachine.

Figure 4.1: Suggestion on hierarchies in models.

The focus of *From* HUPPAAL *to* UPPAAL*: A Translation from Hierarchical Timed Automata to Flat Timed Automata* was to describe and explain formally how such a language could be constructed and how it could be flattened to the language of UPPAAL. The focus of this project will however, not be to formally show that this is possible, but will rather be to create a tool that can become the foundation for investing how the concept of components could be useful to include in tools like UPPAAL.

4.2 Integrated Development Environment

Modern IDEs have many features that assist the user during development. We would like to investigate if ideas and features from these tools could be useful to include in a new model checking tool. The following sections tries to draw parallel between common IDE features and model checking.

4.2.1 Continuous Syntax Checker

A common feature in modern IDEs is the continuous analysis of syntax, whenever the programmer enters a piece of code that can not build, the IDE informs the user. In a similar way, one could imagine that a new version of the UPPAAL tool would continuously indicate whenever an invalid model is present. As for the current version of the tool, syntax checks is only shown when the model is “compiled” (when being simulated, queried, or when issuing manual checks).

4.2.2 Static Code Analysis

Another common feature in modern IDEs is static code analysis. This allows developers to find sources for common mistakes or bad practices, like boolean variables that are always false or unused method declarations. In a tool like UPPAAL, one could introduce similar warnings for things like pointing out deadlocks or unbound integers (potential state space explosion). As of the continuous syntax checker parts of this code analysis could prompt the user whenever changes to the model occur, like linters in IDEs.

4.2.3 Refactoring

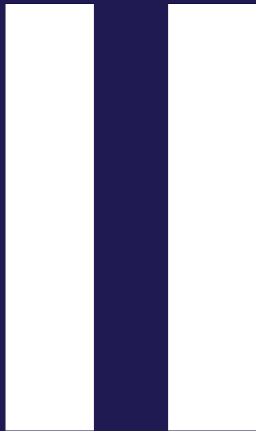
Programmers like to refactor their code, by changing names or structure of the code. This makes it a very important technique in software development [Mens and Tourwé, 2004]. In IDEs there are features that support these actions, like the renaming of variables or extracting logic to a new class. Such features would also help users change their model incrementally if a variable name is to be changed or some part of the model ideally should be encapsulated differently.

4.2.4 Shortcuts

Modern IDEs supports shortcuts for easy navigation, which could directly be mapped to a new version of UPPAAL as well. Especially features like *find usage* and *find declaration* could assist the user in quickly finding where items like channels are defined and used. Moreover, a project-wide search, like we know it from modern IDEs, could help the user find details more efficiently. Furthermore, IDEs tend to have shortcuts not only for finding a piece of code but also for accessing functionality, like static code analysis or refactoring in the development environment.

4.3 Delimitation

The main focus of this project will be to create a new version of UPPAAL, where hierarchies and IDE features will be in focus. For this reason, we have chosen not to focus on integrating the trace and simulation functionality of UPPAAL into the new tool. This means that the tool should be able to answer queries based on the new hierarchical structure, but should not be able to produce proofs or counter proofs of these queries in form of a trace as we know from UPPAAL. If the introduction of components comes to show potential, exploration of state space and traces as evidence for queries is functionality that is important to consider.



H-UPPAAL

5 The H-UPPAAL Manifesto **29**

6 Formalism **31**
6.1 Notation
6.2 Initial Formalism

7 Design **35**
7.1 Location
7.2 Edges
7.3 Components
7.4 Subprocedures
7.5 Setting the Layout for a New IDE

8 Showcasing the H-UPPAAL Tool **45**
8.1 Context Menus
8.2 Query Pane
8.3 Displaying Errors
8.4 Declarations
8.5 Making Models
8.6 Storing Models

9 Flattening **55**
9.1 From Components to Templates
9.2 Execution of Subprocedures

10 Queries **61**
10.1 Single Subcomponent Queries
10.2 Multiple Subcomponent Queries
10.3 Long Queries

5 The H-UPPAAL Manifesto

To better steer the project, a common practice is to use a set of principles you want to adhere to. These principles will act like guidelines throughout development and may be redefined if needed.

One example of such principles can be found in the Agile Manifesto [Beck et al., 2001], used in agile methodologies such as SCRUM, Extreme Programming (XP), and Unified Process (UP). This manifesto is quite broad but still, focuses on what is important in an agile setting. Another example is the GNU Manifesto [Stallman, 1985-1987] which has been used as a support tool during development of the GNU systems. This manifesto is more detailed since the area it covers is more specific.

To help us during the development of H-UPPAAL we have defined the following 6 principles.

> **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.

Given that H-UPPAAL is a new tool based on the well-established tool UPPAAL, we highly prioritize Principle 1 (Backward Compatible), since it will allow current UPPAAL experts and users to use the new tool with little to no adaptation needed. By respecting the ways of old and similar tools we cohere with the *Consistency* design principle given in *Designing Interactive Systems: A Comprehensive Guide to HCI and Interaction Design (2nd Edition)* [Benyon, 2010, p. 90].

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

The process of developing models contains many tedious or mundane tasks, when building the model, updating it, but also when writing queries that we want the model to adhere to. Currently, there exists a good amount of feature-rich IDEs, such as *IntelliJ*, *Visual Studio*, and, *Xcode* just to name a few. All of these tools assists their users in automating specific tasks that will increase the efficiency of the user. We see many different possibilities for creating such features in a tool used for model checking and do therefore prioritize exploring some of these during this project.

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

As mentioned in Issue 1 (Limited Screen Real Estate), we only have so much screen real estate before we run into problems. A way of solving this is to collapse some of this information. We prioritize this principle since we believe that it will be a good technique to utilize when designing models in H-UPPAAL in order to increase the overall overview of the model rather than seeing a complete picture with every detail.

> **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.

As discussed previously in Issue 4 (Anonymous Locations) and Issue 3 (Ordering of Properties), communication can be hard when locations are not named, forcing people to try and describe the location based on placement, incoming or outgoing edges. With the introduction of Principle 4 (Identity and Relation), we try to solve this issue, by always having an explicit identity for different elements in the model. This will allow modelers to directly target specific elements with no ambiguity or uncertainty.

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

The UPPAAL tool is heavily used in scientific papers. However, as mentioned in Issue 6 (Forgotten Declarations), it is not always the case that the authors of such papers include all of the details in their article. This might be due to strict space requirements, or the authors deeming the left out details secondary. This can cause confusion with readers, and may even hurt the reproducibility. Principle 5 (Printable) is established so that authors will have a harder time adding models to their paper with missing details. But of course, it is still possible to simply leave out models.

> **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.

As described in Issue 5 (Overlapping Objects), elements in UPPAAL do not currently take up space, sometimes resulting in problematic situations where the users of the tool are confused about the meaning of the model (imagine two locations on top of each other, with different incoming and outgoing edges). To ensure that such situations do not occur, or at least occur rarely, we would like to enforce that elements take up space.

6 Formalism

The purpose of this project is not to invent and proof correctness of a new formalism for a hierarchical structure in networks of timed automata. This project is instead focused around realizing a tool that will support such a hierarchical structure. We do, however, suggest a draft for a potential formalism covering the essential parts. We have not provided semantics for this formalism, but intend do so in the future. Note that the formalism is strongly inspired by *From* HUPPAAL *to* UPPAAL*: A Translation from Hierarchical Timed Automata to Flat Timed Automata* [David and Möller, 2001].

6.1 Notation

This section will introduce some new notation that will be useful for explaining components and their hierarchical structure.

Notation 6.1 $c_1 : C$ 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 6.2 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 6.3 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 6.4 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, $Loc(C) = \{l_0, l_1\}$.

6.2 Initial Formalism

This section will give draft for a formalism that later will be the basis for implementing the H-UPPAAL tool. This formalism is heavily inspired by definitions given in *From* HUPPAAL *to* UPPAAL*: A Translation from Hierarchical Timed Automata to Flat Timed Automata* [David and Möller, 2001]. Definitions 6.2.2, 6.2.3, and 6.2.4 are all only adjusted slightly to better conform with some of the concepts in UPPAAL. Please note that this formalism is far from complete, and will only describe what we deem to be the essential parts.

Definition 6.2.1 — Data Elements, Clocks and Channels. Let D be a finite set of data variables, Cl a finite set of clock variables, and Ch a finite set of synchronization channels.

The data elements (variables and constants) of the formalism is a part of annotating each state of the system. In other words, for each state in the system, these items is a property of the same state. Channels are, on the other hand, a way of defining how to progress from one state to the next via synchronization.

Definition 6.2.2 — Guard and Invariants. A data constraint is an expression that evaluates to a boolean value. This expression can consist of arithmetic and boolean operators over the set D and data constants. A clock constraint is an expression of the form $cl_1 \bowtie n$ or $cl_1 - cl_2 \bowtie n$, where $cl_1,\ cl_2 \in Cl$ and $n \in \mathbb{N}$ with $\bowtie \in \{<, \leq, =, \geq, >\}$. A clock constraint is *downward closed* if $\bowtie \in \{<, \leq, =\}$. A guard is a finite conjunction over data and clock constraints. An invariant is a finite conjunction over *downward closed* clock constraints. Let $Guard$ be the set of guards, and $Invariants$ be the set of invariants. Both contain the constants $true$ and $false$.

These sets define what transitions are enabled from one state to the next, i.e. it tells what edges is enabled based on the current state. If a guard is not evaluated to $true$ the edge is not in the set of next possible transitions. Likewise, if an edge goes to a location, where the invariant evaluates to $false$, we exclude this transition.

Definition 6.2.3 — Synchronization. Let $Sync$ be a finite set of channel synchronizations. We have that $\forall c \in Ch \mid c?, c! \in Sync$.

Like a guard, a synchronization limits which transitions are available. Edges with synchronization are only enabled if another edge of a subcomponent running in parallel has the opposite synchronization on them. If an edge with the synchronization $c!$ is enabled then the state will also contain a subcomponent where an edge with the opposing $c?$ is also enabled.

Definition 6.2.4 — Assignment. A data assignment is of the form $v := x \mid v \in D,\ x \in X$ and where X is the set of expressions and where v and x are of the same type. A clock reset is of the form $cl := 0$, where $cl \in Cl$. Let $Assignment$ be set set of data assignments and clock resets.

Lastly, an assignment is a way to update the sate (resetting clocks and updating variables) when an edge with said assignment is included in a transition.

Based on the definitions above, we are now able to give our own definition for a hierarchical system alongside the definition for a component, and in effect, subcomponent and subprocedures.

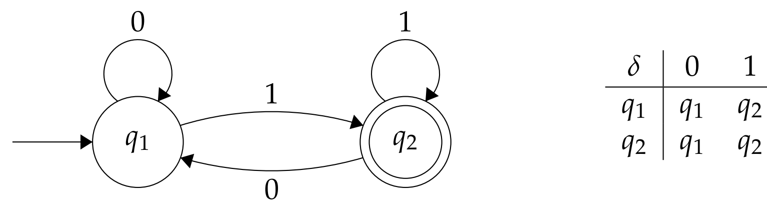
Definition 6.2.5 — Component. A component, c, is a tuple $(L, l_0, l_1, D, Cl, Ch, E, Sp, F, J)$:
1. L is a set of locations in the component
2. l_0 is the initial location and l_1 is the final location. $\{l_0, l_1\} \subseteq L$. $l_0 \neq l_1$
3. D, Cl, and, Ch are sets of data variables, clocks, and channels, which populates the sets $Guard$, $Sync$, and $Assignment$.
4. $Sp \subseteq F \times \mathcal{P}(Sc(c)) \times J$ is the set of subprocedures, where F is the set of forks, and J is the set of joins
5. $E \subseteq (L \setminus l_1 \cup Sp) \times Guard \times Sync \times Assignment \times (L \setminus l_0 \cup Sp)$ is the set of edges
6. $Inv : L \rightarrow Invariant$ maps every locations l to an invariant.

Definition 6.2.6 — Hierarchical System. A hierarchical system is a tuple (C, c_0):
1. C is the set of all components in the system
2. c_0 is the main component

With this draft, we are now able to start looking into how we can design a tool that will support the hierarchical system.

7 Design

This chapter will cover some design ideas and decisions, based on issues found in Chapter 3 and principles defined in Chapter 5. Let us first consider the origin of the automata. Figure 7.1 (given in *Introduction to the Theory of Computation* [Sipser, 2012, p. 37]) shows a visualization of simple automata recognizing the language $L = \{w \mid w \text{ ends in } 1\}$, e.g. $w_1 = 0001 \in L$ or $w_2 = 1101 \in L$. The behavior of this automata is very simple, and does not have any notion of time.



δ	0	1
q_1	q_1	q_2
q_2	q_1	q_2

Figure 7.1: Automata recognizing $L = \{w \mid w \text{ ends in } 1\}$.

In Figure 7.1 we see that all states are labeled. This is due to the fact that people describe the transitions in the automaton using a transition table rather than only having a visual representation (like current versions UPPAAL).

This model has later been expanded on to include time, and in UPPAAL, we have a network of timed automata describing the system. These types of automata are visualized in a similar fashion to the one we have already seen. However, with the addition of clocks and synchronization, the transition table gets more complex, motivating the need for drawing the automata in order to describe the behavior of the system. The shift from writing transition tables to only drawing the automata caused the labeling of states to be some what unnecessary. The only real reason as to why people would label states is so for the sake of communication. This could be particularly useful when talking with a co-worker or formulating property queries similar to what is possible in UPPAAL.

Just like timed automata, UPPAAL has also gone through some changes. For instance with the introduction of *select* statements, which is a shorthand for creating multiple edges. Furthermore, different versions of UPPAAL introduce new changes. For instance, UPPAAL TIGA[1] which introduces the concept of dashed edges. All of these changes and additions motivates a clear and orderly way of representing information to the user.

[1] http://people.cs.aau.dk/~adavid/tiga/

7.1 Location

The following sections will cover some of the ideas we had through development in regards to locations, especially in regards to presenting the user with a name which can be used as a basis for communication between collaborators, see Principle 4 (Identity and Relation).

7.1.1 Expanding Locations

This idea is based on allowing the user to hide unnecessary information as per Principle 3 (Information Hiding). This is visualized in Figure 7.2, where Figure 7.2a shows the location with the panel collapsed, and thus hides both the `brewing`-label and the invariant `a > 2 && k != 3`. Figure 7.2c shows the location with the panel expanded, showing both the label and invariant. Figure 7.2b shows one intermediate step in the expansion and contraction of the panel. By allowing the panel to be expanded and contracted we achieve a cleaner and more uncluttered model, where unnecessary information is hidden, with Principle 3 (Information Hiding).

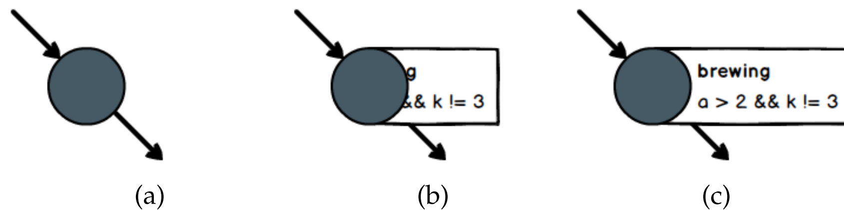


Figure 7.2: Visualization of expansion for locations.

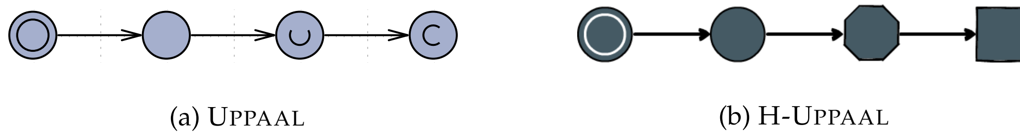
A problem with this approach is that it contradicts Principle 4 (Identity and Relation) since it allows the user to hide the label of the location, which gives it identity, potentially making it harder to communicate. This might also conflict with Principle 6 (Objects Require Space), since the panel will require space to be shown when expanded, but require none when collapsed. A possible workaround for this would be to require the space even though the panel is hidden. One could argue that this is not a big problem in the example since it is clear to see that the edge is coming from the location. The idea does, however, conform with Principle 5 (Printable), since the printed version would simply contain the location with the panel expanded, as shown in Figure 7.2c.

7.1.2 Changing Shape

In current versions of UPPAAL, the content of locations are used to indicate either that the location is *urgent* (marked with `U`) or *committed* (marked with `C`). This space is also somewhat used to indicate if it is the initial location. All of this can be seen in Figure 7.3a. We would like to explore the idea of using the inside of locations to indicate identity as of timed automata seen in Figure 7.1. Before doing this, we need to find an alternative to mark urgency of a location.

To display the urgency of a location, without using the content of the location, we suggest that the shape of the location could be altered in such a way that normal, urgent and committed locations each have distinguishable shapes. We suggest that normal locations would continue being a circle, while urgent and committed locations have a more strict shape (to accommodate their stricter nature). We suggest that the shape of urgent locations would be an octagon so that it still maintain some of its circular shape. We also suggest

that the shape of committed locations would be square since this urgency is the most strict. These ideas are illustrated in Figure 7.3b.



(a) UPPAAL (b) H-UPPAAL

Figure 7.3: Changing the shape to accommodate the location type.

It is important to realize that this idea contradicts Principle 1 (Backward Compatible) since current users of UPPAAL would probably have to adjust to this change. However, it is important to keep in mind that the formalism for automata have gone through a lot of changes from when it was first suggested. Take for instance the automaton in Figure 7.1 - here the final state is marked in the same way as initial locations would be marked in UPPAAL, where as the initial location is marked with a simple arrow.

Another problem with this idea is the fact that the space inside the locations is fairly limited, only allowing for a few characters to be displayed. This might hurt the communication since the locations do not have saying names. By requiring locations to have a name, while only allowing a few characters, modelers might be discouraged to provide meaningful names (because they find it hard or infeasible) and end up with labeling locations `L1`, `L2`, `L3`, etc. This ultimately sacrifices the ability to give locations saying names, and might cause communication issues or issues with understanding the behavior of the automaton.

7.1.3 Unique- and Descriptive Identifiers

This idea is based on the problems described in Section 7.1.2. The idea here is to provide modelers with additional freedom to pick a descriptive label, while still maintaining an automatically generated unique identifier.

Naming locations are not always straight forward, and some people tend to skip it. Take for instance the automaton in Figure 7.4, where we try to model the front door of a local dance club. This club will not open the door until all of the staff members are ready. To do this three channels have been established: `bar_ready`, `dj_ready`, and, `bouncer_ready`. The door goes from the `Closed` location to the `Opened` location when each of these channels have been synchronized on. We have no real need for labeling the two intermediate locations since the entire model is a simple sequence of actions.

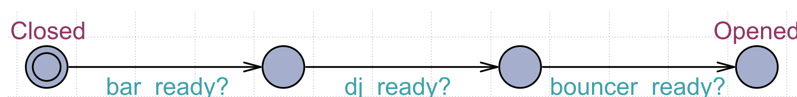


Figure 7.4: Model for a club door.

However, when conforming with Principle 4 (Identity and Relation), all locations must have some identity. We might leave this up to the modeler, who would then be forced to come up with labels for all locations. Some modelers might, as previously mentioned, simply call these `L1`, `L2`, `L3` etc. A more saying name for the locations could be `WaitForX` since that is what the location does - it waits for the synchronization to happen. This is all

good, but some people might be annoyed that the `Closed` location is not named with the same logic, so they introduce a new location as seen in Figure 7.5.

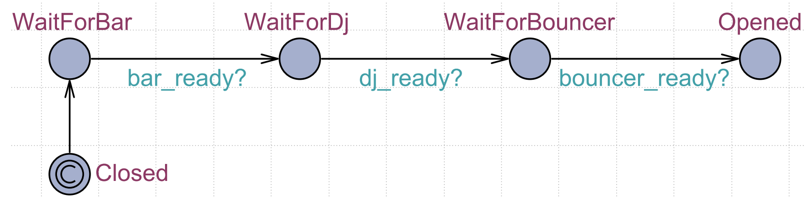


Figure 7.5: Introducing labels for all locations while still keeping *Closed* and *Opened*.

The idea is then to generate short unique identifiers for all locations placed in the center of locations but still allow the user to further label the location with an optional name. This idea is illustrated in Figure 7.6, where we see both the `Closed` and `Opened` locations, but also have the labels `L1`, `L2`, `L3`, and `L4`, which might be useful for communication or debugging the model.

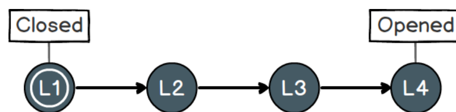


Figure 7.6: Illustration of both unique identifiers and descriptive identifiers.

A problem with this approach is that it might be confusing with two names for the same locations, e.g. `Closed` = `L1`. Also, adding text to all locations might compromise overview, conflicting with Principle 3 (Information Hiding).

7.2 Edges

The following sections will cover some ideas we had during the development of edges, primarily regarding the representation of the four properties that can be assigned to an edge: *select*, *guard*, *synchronization*, and, *update*.

7.2.1 Boxed Properties

The idea here is to apply the gestalt law regarding containment [Tidwell, 2010], by wrapping the four properties in a box. The properties box is structured almost like a table, making the user unable to re-arrange the ordering. This will probably limit the severity of Issue 2 (Movable properties). We also thought of using icons indicators for each type of property (instead of using colors, as UPPAAL currently does). As seen in Figure 7.7, *select* properties are indicated by the `:` icon, *guard* by `<`, *synchronization* by `!?`, and *update* by `=`.

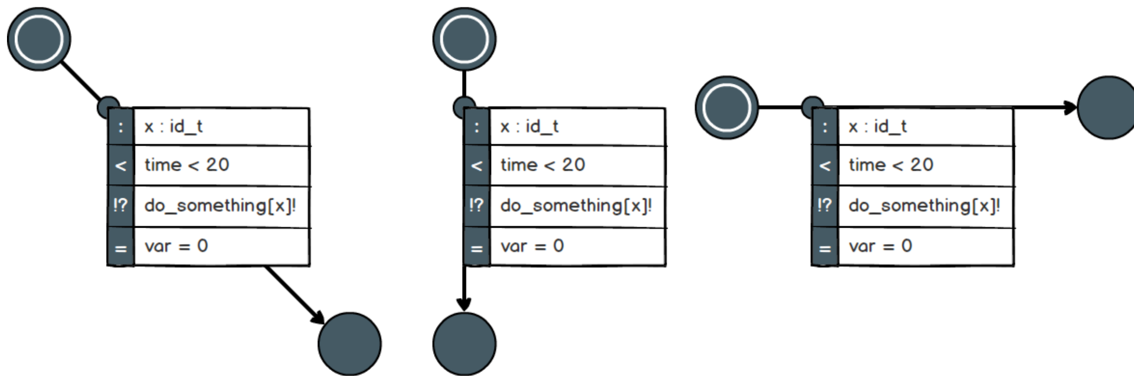


Figure 7.7: Boxed properties idea.

In Figure 7.7, we see that all four properties are given. This is not required by UPPAAL, so the user could simply leave some (or all) of these blank if wanted. This causes a lot of screen estate to be wasted on displaying an empty structure. This might even make Issue 1 (Limited Screen Real Estate) even more critical.

To give the modelers some freedom over where the properties box is placed, we have added a small circular area in the top left of the box, which the user can use to drag the box around. This idea conflict with Principle 6 (Objects Require Space), since it could allow modelers to move the box in such a way that it covers critical information, such as locations. However, this might be hard to avoid with this idea, since the box should probably resize whenever the content of one of it fields are changed. For instance, if the update statement was longer, the whole box would have to be wider in order to show the entire update statement.

7.2.2 Free Floating Properties

This idea is a further expansion on Section 7.2.1, but instead of wrapping the four properties inside a box, now each property has its own box. Each property will also have a circular shape that, as previously described, can be dragged around on the edge. However, to avoid Issue 3 (Ordering of Properties), the four circular shapes must come in the correct order, starting from the source location. This idea is visualized in Figure 7.8.

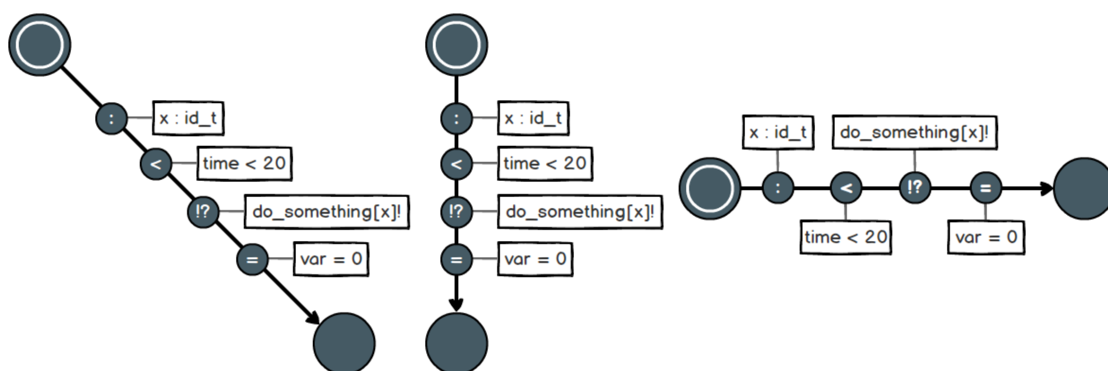


Figure 7.8: Free floating properties idea.

As previously mentioned, UPPAAL does not require users to set all properties. Since this idea is based upon single individual boxes, we can now simply hide the empty boxes if the corresponding property is not set, and show it to the user when the corresponding icon is pressed. This is visualized in Figure 7.9, where only the *select* and *synchronization* properties are set. By pressing the guard or update property, a new empty box would appear, that the user could then fill in the new property.

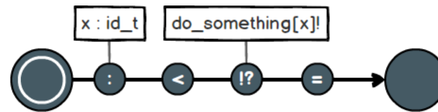


Figure 7.9: Free floating properties idea with *select* and *synchronization* properties set.

7.3 Components

The following section will cover some ideas we have had regarding the design of the new component concept. Figure 7.10 shows the basic shape of a component. The idea here is to use the gestalt law of consistency [Tidwell, 2010], meaning that all components (and subcomponents) should feel similar due to their shape, and placement of the *initial* (top left) and *final* (bottom right) locations. These two special locations are placed on the border of the component box to symbolize that they act as an interface for the component.

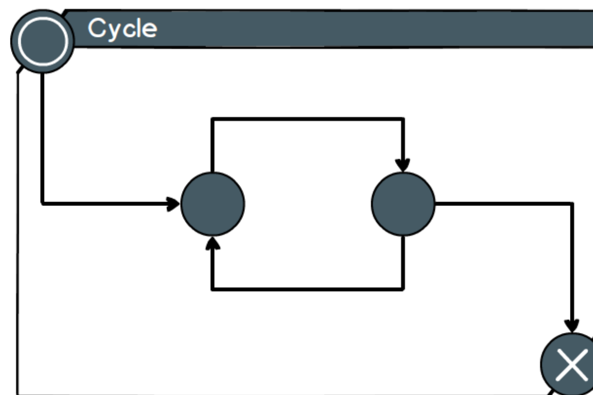


Figure 7.10: Visualization of a component.

To cohere with Principle 4 (Identity and Relation), the component design includes a top bar, which displays the name of the component. As seen in Figure 7.10, the component is labeled `Cycle`.

7.3.1 Subcomponents

When adding a subcomponent into another component, it should be easy to recognize that the element is, in fact, a subcomponent. To achieve this effect, we have enforced that the visualization of a subcomponent is very similar to the actual component, however, without content other than the initial and final locations. Instead of showing the content of the component, it now shows a descriptive text for the component, allowing modelers to see the subcomponent as a black box. An example of a subcomponent can be seen in Figure 7.11, where the subcomponent `Cycle` is used inside the `Controller` component.

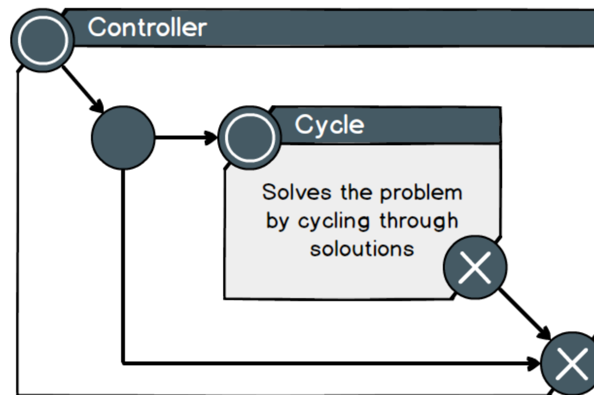


Figure 7.11: Visualization of a subcomponent inside a component.

7.4 Subprocedures

To allow the user to run components in parallel we introduce a fork and join construct. The initial idea was to to simply implement a similar construct as we know it from activity diagrams in UML [Object Management Group (OMG), 2015], as seen in Figure 7.12a. The problem with this is that we have certain modeling rules for the start and conclusion of a subprocedure, see Section 6.2. If we simply chose to inherit the element from UML, there would be one visual construct that represents both join and fork. In combination with the rules of these constructs, we deemed it would be rather confusing that the same visual element represented two different behaviors. For this reason we have the idea that a fork and a join should look similar but distinct enough to know the difference between the two, and thereby resolve some of the confusion that might be when creating subprocedures. As seen in Figure 7.12a, we replace the line of UML with a trapezoid. The trapezoid of the fork is designed to associate the user with the thought of spreading parallel components, whereas the join is designed to symbolize a funnel, merging these parallel subcomponents.

(a) UML activity diagram.

(b) New design suggestion.

Figure 7.12: Visualization of subprocedures.

7.5 Setting the Layout for a New IDE

Based on Principle 2 (Integrated Development Environment), it is desired to make the tool look and feel more like an IDE than the current version of UPPAAL does. For this reason we had a look at how different development environment were organized. In programming environments, the code is, of course, the primary focus of the tool, which is often placed centrally in these tools. A screenshot of a Java IDE called IntelliJ can be seen in Figure 7.13. Here we see that we have the code placed in the center, we always have access to some part of the code, and we use the rest of the tool to write this code as efficiently as possible. If you scrape all features that are not needed to write code you end up with a simple text editor.

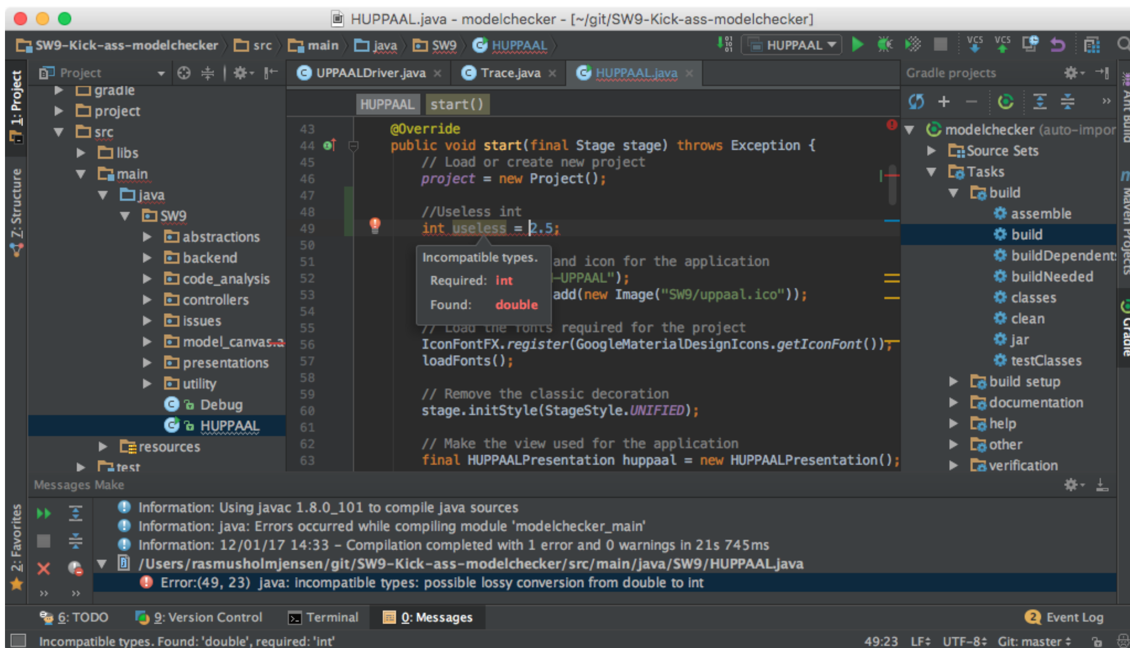


Figure 7.13: The IntelliJ environment.

Similarly to IDEs, the model is the primary focus of a model checking tool. For this reason, we believe that having the modeling canvas as the central concept will increase the overview and productivity for the task of performing model checking. We have created some mockups to try to express this idea, in Figure 7.14, we see an outline of a tool where we have the modeling canvas as the basis of the tool. The idea is that everything but the canvas can be collapsed, and any features that might assist you in creating these models is a mere panel that can be opened or closed.

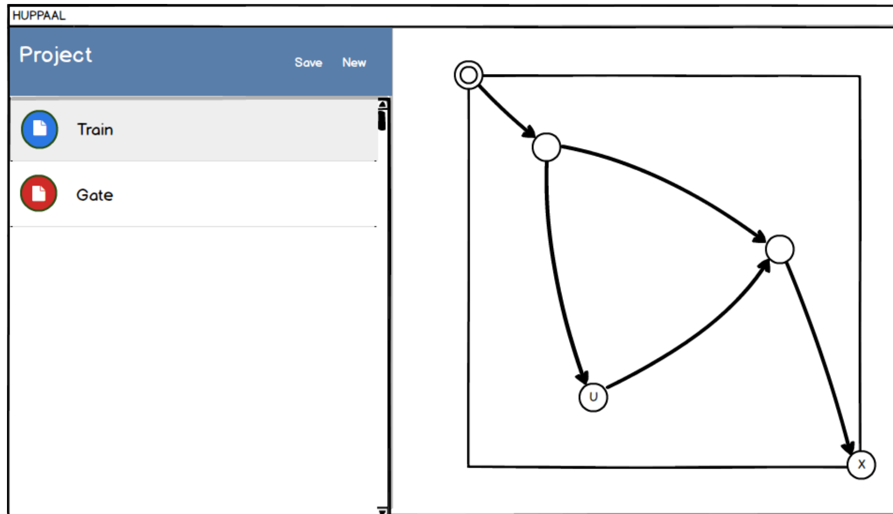


Figure 7.14: Mockup of layout.

As of Figure 7.14, we can use the file panel to the left to navigate through the model, changing the content of the canvas by clicking them. This panel should not be required to be there at all time, we should be able to collapse it and use as much screen real estate we want on the canvas.

Other features that assist the user of a model checking tool should, as many IDEs does it, be allowed to take up some screen real estate when it is needed. On Figure 7.15 we see a mockup of how the tool should behave with respect to features like querying the model. We see here that we are able to close the file panel and open a pane where we have this particular feature available.

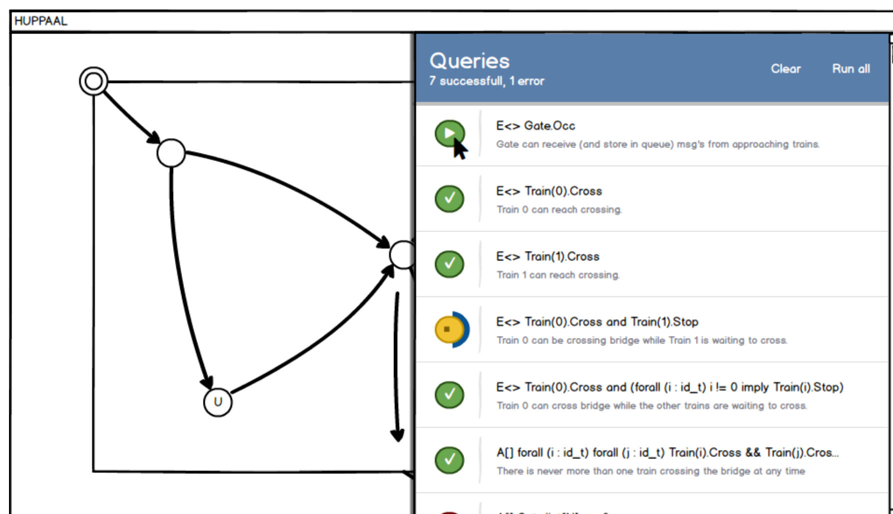


Figure 7.15: Mockup of query pane.

In general the design of the tool should be consistent with this layout idea. The model canvas is the core of the tool like the text editor is of programming IDEs. The other parts of the tool is merely a way to utilize the model and be as efficient and productive as possible when designing them.

8 Showcasing the H-UPPAAL Tool

This chapter showcases the developed H-UPPAAL tool and most of its features. The H-UPPAAL tool has been developed in Java as a JavaFX application. JavaFX is a framework that is designed to create rich powerful GUI applications and supports technologies like hardware acceleration. Furthermore, JavaFX is available on many systems, including most desktop and mobile operating systems.

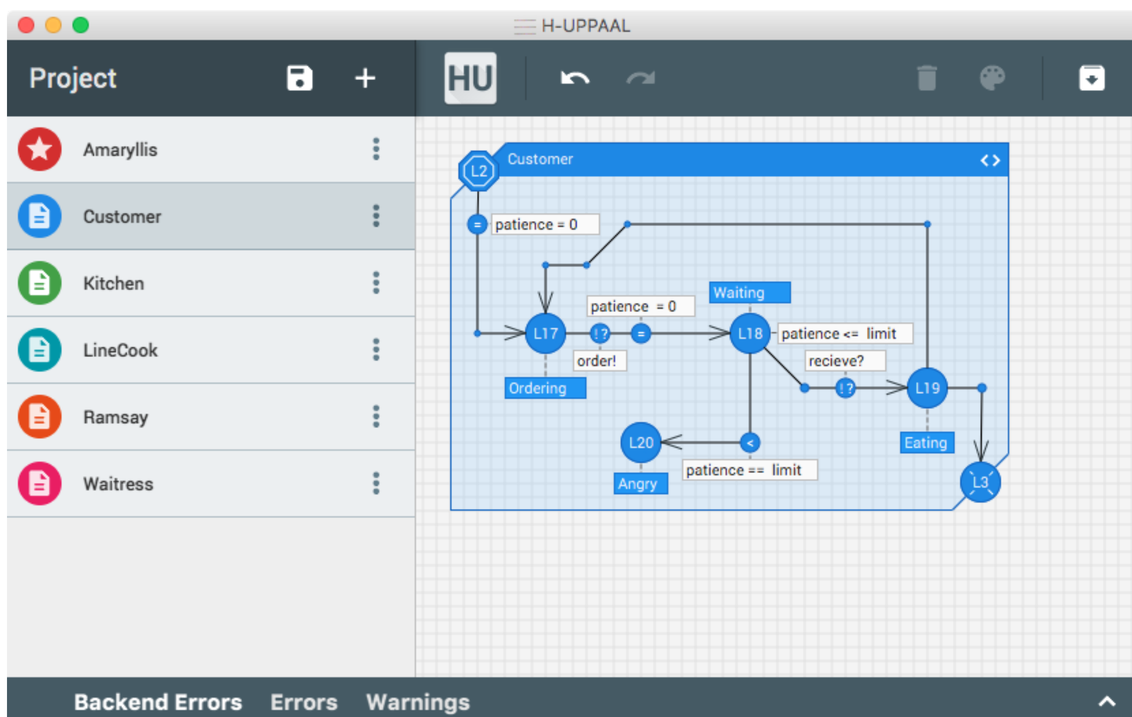


Figure 8.1: Initial screen of the H-UPPAAL tool.

Figure 8.1 shows how the tool looks when opened. In this case, we have already done some work in the tool. On the left, we see a list of all components currently in our project. This project panel can be opened and closed by pressing the `f` key. In this panel, we see that all of the components are named, and are colored. The topmost component, *Amaryllis*, is marked with a star, indicating that this is the main (or root) component. We also see that the component *Customer* is highlighted, indicating that it is opened in the canvas, which can be seen on the right, where we see the model for the *Customer* component. All of the models used in this section will be described in more detail in Chapter 11. In the top most part of the window we see buttons. Starting from the left, we have a save button, which will store the project on disk. The next button will add a new component. Next, we have undo and redo buttons. These are also accessible through keyboard shortcuts `ctrl + z` and `ctrl + shift + z`, where `ctrl` is dependent on operating system, e.g. `cmd` for macOS. Note that these buttons will be enabled or disabled depending on if their respective actions

are available. In this case, we see that it is possible to undo something, but not to redo anything. Next, we have a delete button, that is enabled whenever an element has been selected, for instance, a location. This functionality is also available by pressing `delete` on the keyboard. The next button is used to color selected elements, and will only be enabled when something is selected. Coloring elements are also possible by right-clicking the element and then selecting the desired color. The last button will generate a UPPAAL model based on the project. We introduced this button for the purpose of debugging the models generated by the H-UPPAAL.

8.1 Context Menus

In Figure 8.2 and Figure 8.3 we see two different context menus enabled by the mouse in H-UPPAAL. The first menu is shown by right clicking the mouse inside a component. From this menu, it is possible to add a location (this is also possible to do by pressing the `l`-key), fork, join, and subcomponent. From this menu, it is also possible to generate a query that will check if the component contains a deadlock. This query is generated as described in Chapter 10. Lastly, we see a color wheel, which will make is possible to change the color of the component.

Figure 8.3 shows another context menu. In this case, the three vertical dots in the project panel have been clicked. In this menu, we are presented with the possibility of configuring the component. We can either mark the component as main, or we can mark the component to be included in periodic checks. Note that this is a feature that is not implemented yet, but is described in Section 13.2. This menu also makes it possible for users to describe the component. This description will be showed inside subcomponents. Next, we see a color wheel similar to the one we already saw. Lastly, it enables the user to delete the component.

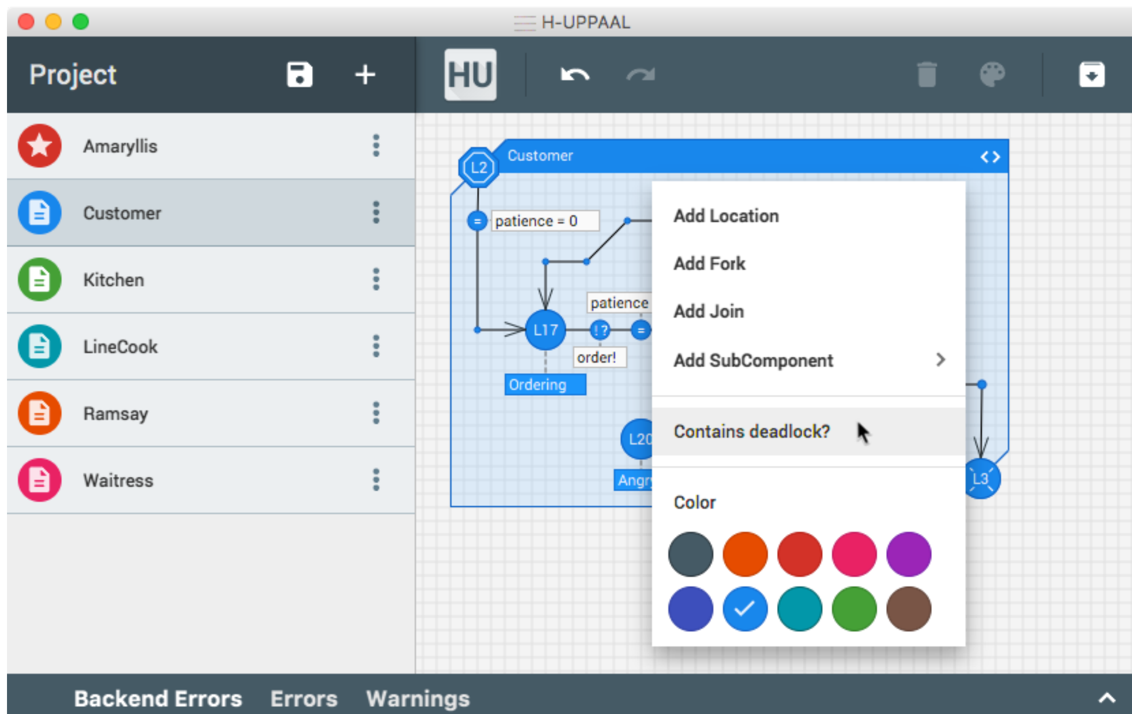


Figure 8.2: Editing a component from the canvas.

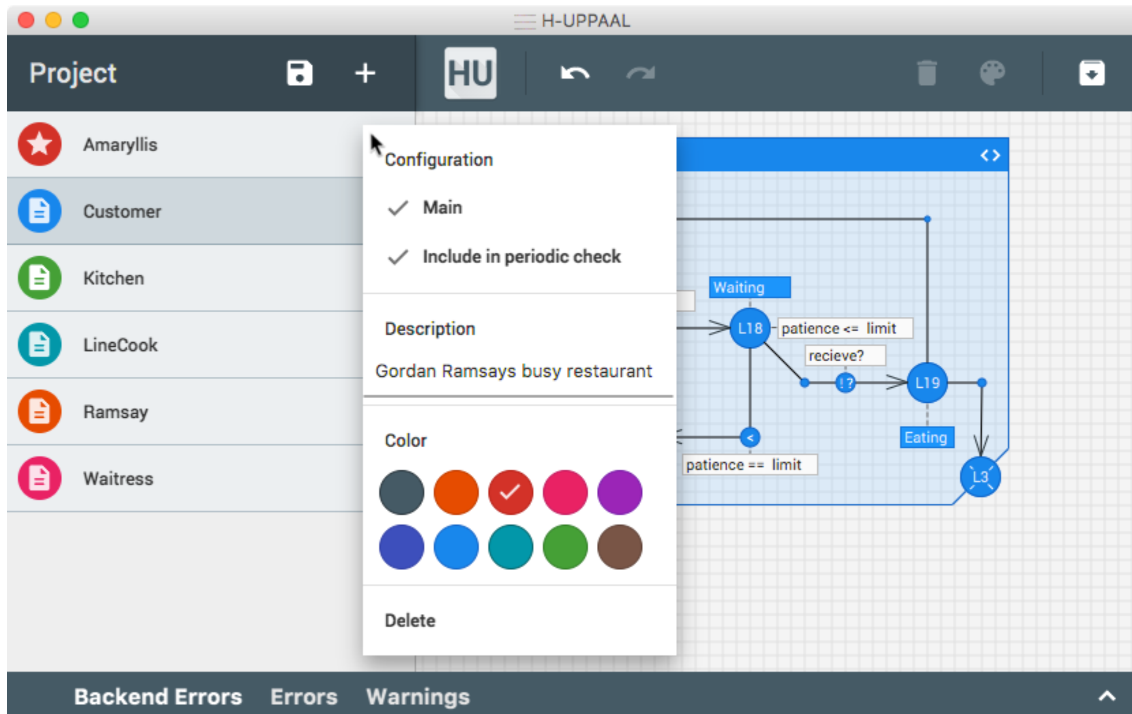


Figure 8.3: Editing a component from the file panel.

These context menus is a way of adding functionality directly on an element inside the UI. By adding these type of menus on elements in the canvas we allow for easy access of shortcuts and changing of properties. At the same time, we associate said properties to the items that are being interacted with by the mouse. The visual expression of these menus is the same regardless of how the are opened, to create a more consistent design. Hence the context menu that pops up by from vertical dots, and the ones opening when right-clicking items in the canvas are implemented to have the same look and feel.

8.2 Query Pane

To query the model, we have introduced the query pane. This pane can be seen in Figure 8.4 and can be opened and closed using the `q`-key. Opening the query pane shows two new buttons in the toolbar. From the left, we have a "run all queries" button. Pressing this button will run all of the queries using the verification engine. The next button is "clear status". Pressing this button will clear the status of all queries. Below the toolbar, we see a list of all the queries created for the project. To the left of each query, we have a colored bar, indicating the answer of the last run of the query. To the right of each query, we have two buttons. From the left, we have a "run query" button, which will do exactly this. Once the query has started running, an indeterminate progress bar will be shown until the query has concluded. While the query is running, it is possible to stop it. To the right of the run-button is a button, that, when pressed, will show a context menu with the possibility of deleting the query. This menu is intended to include more functionality, such as marking the query as periodic (see Section 13.2), however, this functionality is not present in the current version.

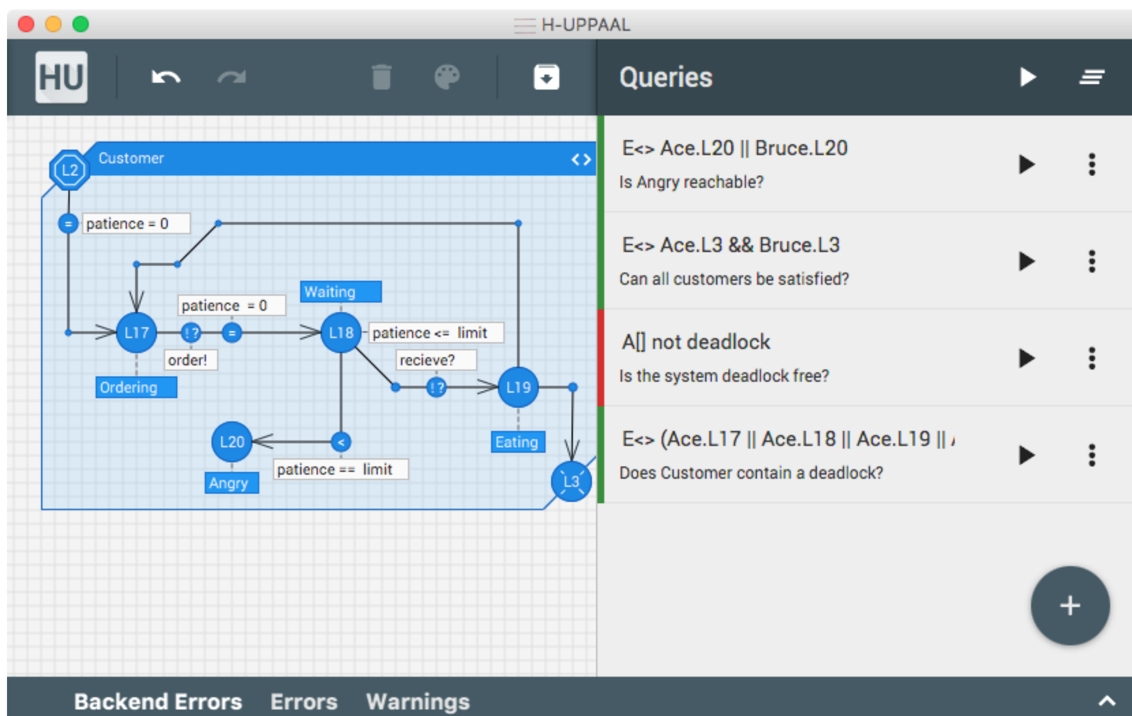


Figure 8.4: The query pane.

8.3 Displaying Errors

To provide the user with errors and warning, we have introduced a panel that expands from the bottom of the tool, inspired by the ones of IDEs. This panel can be seen in Figure 8.5. In this example, we have introduced an error in the `Kitchen` component. The error-panel is split into three tabs: *Backend Errors*, *Errors*, and *Warnings*. Ideally, we would not have the *Backend Errors*-tab, but unfortunately, errors from UPPAAL have not yet been meaningfully translated to ones we can use in the hierarchies of H-UPPAAL. So instead, we introduce some H-UPPAAL specific errors, which can be found relatively easy, without using the UPPAAL engine. These errors include checking the validity of forks, joins, names of locations, etc. In Section 13.1 we describe some additional warnings and errors that we would like to provide to the user.

8.4 Declarations

To write declarations for a component, the "toggle declarations"-button must be pressed. This button is located on the top right corner of every component. Once pressed, the declarations for that component will be shown. In Figure 8.6 we see the declarations for the `Customer` component, which features line numbers, and has syntax highlighting.

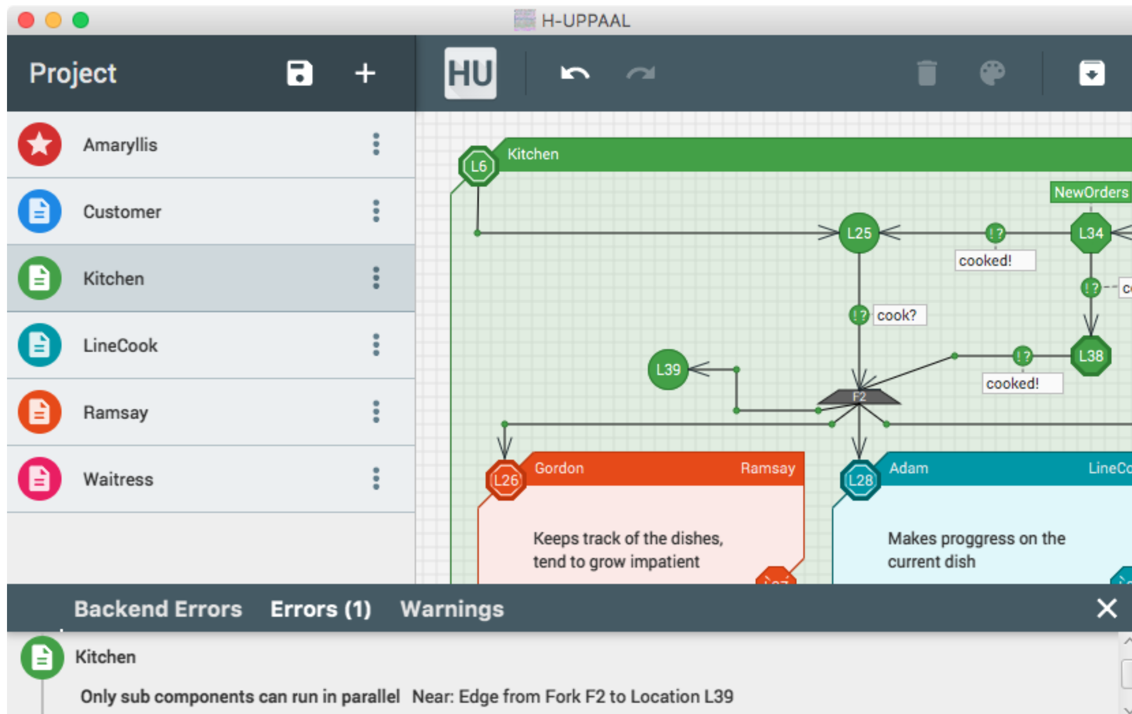


Figure 8.5: The errors tab.

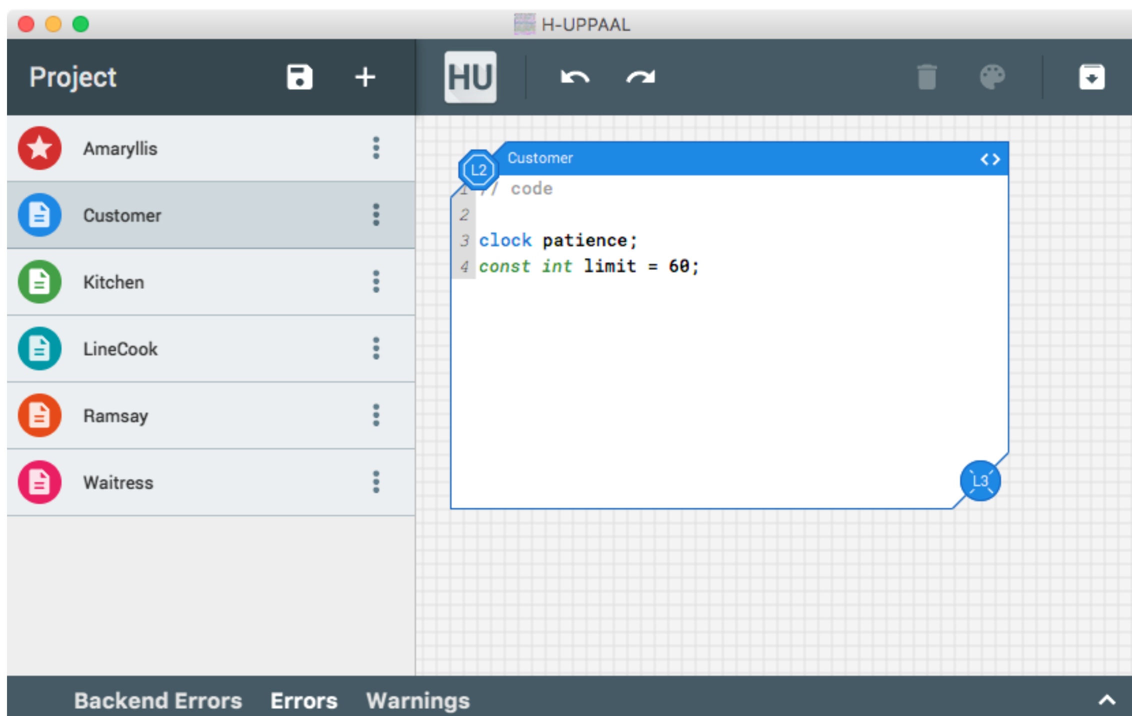


Figure 8.6: The declarations of a component.

8.5 Making Models

With the layout of the tool in place, let us now consider the activity of creating a new model. As previously mentioned, you create a component by pressing the `+`-button in the project panel. This will create a new empty component for us to work in. To create a new location, we can press the `l`-key on the keyboard, which will as seen in Figure 8.7, create a new location that will follow the mouse around until placed in the component by pressing the mouse. We can also create a new location by right-clicking the component and selecting the *Add Location*-entry. This will add a location where we initially right-clicked. If we want to change urgency of the newly added location to be something else than the default, we can open a context menu for a location by right-clicking. This context menu allows us to check either "Urgent" or "Committed". Another option for changing this urgency is to hover the location and press the `U`-key, to make it urgent, or the `C`-key to make it committed.

Now that we know how to add locations, let us consider how edges are drawn. An edge is initialized by holding the `shift`-key while clicking on a location. This will, as seen in Figure 8.8, create an edge originating in the clicked location and follow the mouse around. If we now press somewhere in the canvas, we will create a nail at that position. It is possible to add as many nails as needed. An edge is finished by pressing either another location, a subcomponent, a fork, or a join.

Selecting elements are done simply by pressing on them. When pressed, they will become orange, indicating that they are selected. Currently, it is not possible to select multiple elements. However, we would like to add such a feature.

All of the text-fields throughout the user interface are editable by simply pressing on them. Doing this will place the cursor inside the text-field, allowing the user to edit the value, as you would expect.

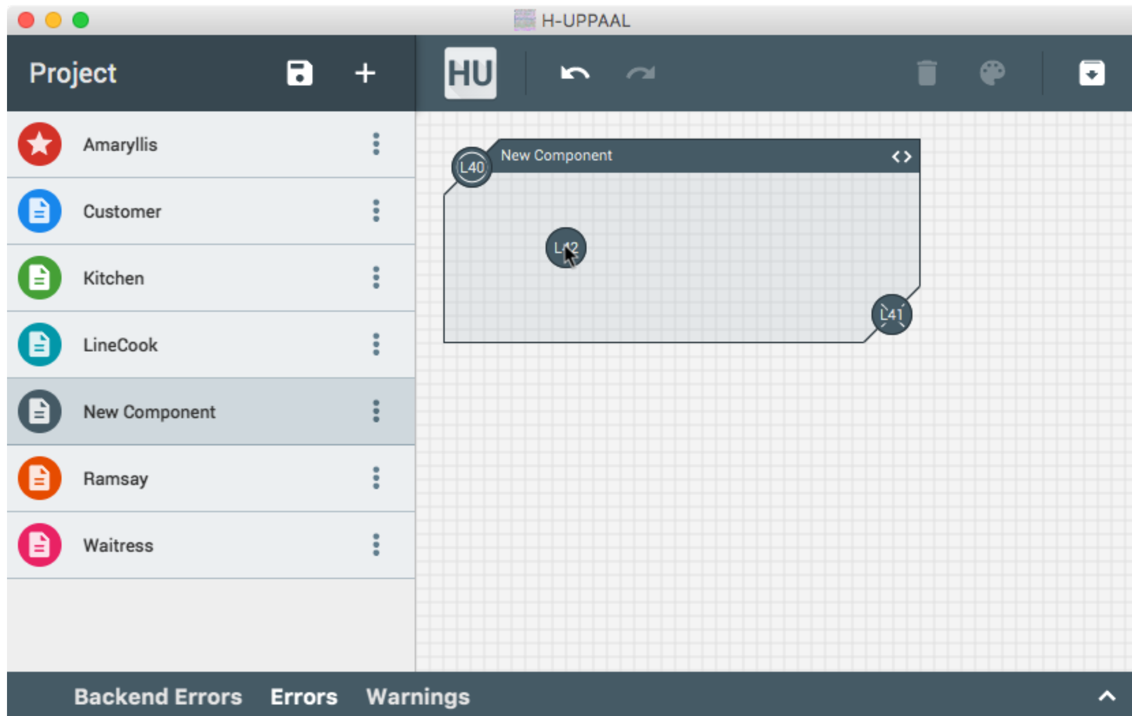


Figure 8.7: Creating a new location.

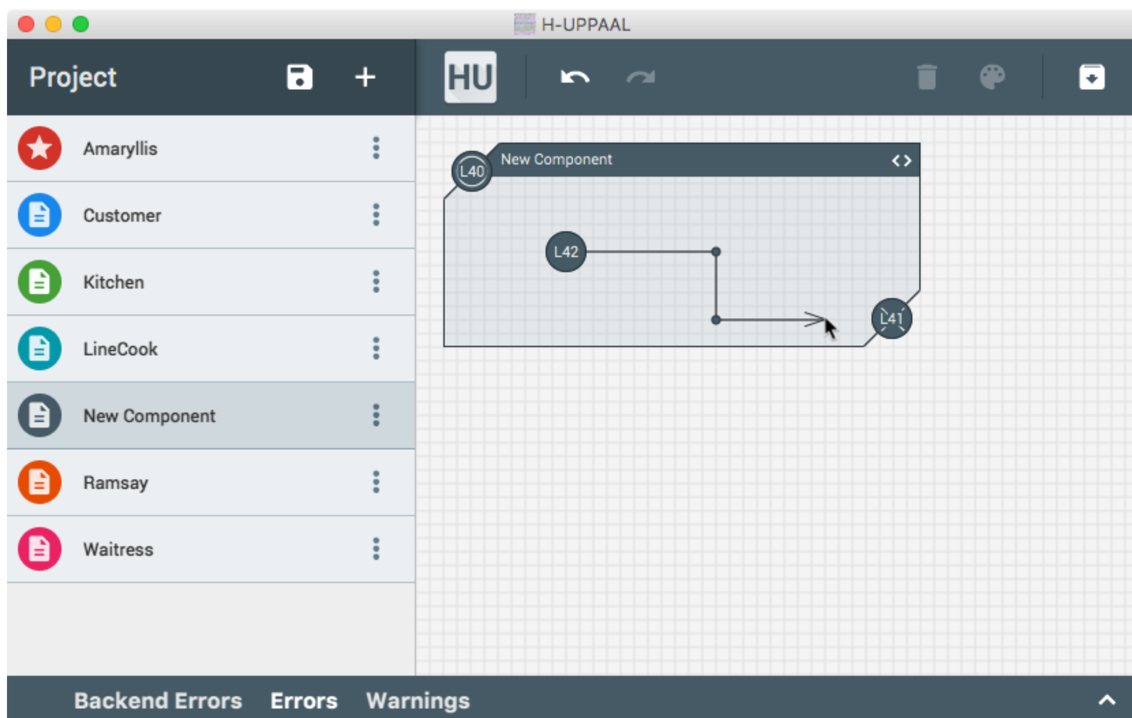


Figure 8.8: Creating a new edge.

8.6 Storing Models

To save the models generated in the tool, we choose to go with a JSON format, since we have had previous knowledge with this format. We also have previous experience with the library used to implement the serialization and later deserialization, GSON[1]. Please note that we did not spend much time in designing this format, instead, we focused on having a *working* format that we could use to save models. This allowed us to save time when developing the tool since we did not have to produce the same models multiple times.

8.6.1 Multiple Files

We generate a JSON file for each of the components in the a project, each named `x.json`, where `x` is the name of the component. We have separated the components in an attempt to better accommodate version control. The idea here is that multiple developers can work on different components without modifying the same file. The idea of creating a file for each component is also inspired by OOP, where a class typically has its own file, e.g. in Java where the `Person` class typically would be located in `Person.java`.

A problem with this approach is placing queries. One could argue that a query should be placed with the component that it queries, but then what about queries such as `E<> deadlock`, or queries concerned with multiple components? To solve these problems, we introduced a file named `Queries.json`, containing all of the queries for the system. This does, unfortunately, mean that it is impossible to have a component named *Queries*. A possible solution to this problem is to introduce a folder specifically for the components, as seen in Figure 8.9b. However, we did not think this was a big problem and ended up with simply implementing the structure seen in Figure 8.9a.

```
project/
├── Component1.json
├── Component2.json
├── ...
├── Componentn.json
└── Queries.json
```

(a) Implemented file structure.

```
project/
├── components/
│   ├── Component1.json
│   ├── Component2.json
│   ├── ...
│   └── Componentn.json
└── Queries.json
```

(b) Structure supporting *Queries* component.

Figure 8.9: File structures.

8.6.2 Component Example

In this section, we will consider the simple model showed in Figure 8.10. This model does not contain any subcomponents due to simplicity. The corresponding JSON can be seen in Code Snippet 8.2. Please note that some of the objects have been collapsed in order to save space. A full example of a JSON file can be seen in Chapter 15.

Let us start by considering the `Full` location (`L2`), which has the invariant `hunger > 20`. To describe this, along with the position and color, we use Lines 5-18 in Code Snippet 8.2. Here, we also see that the `type` is `"NORMAL"`, i.e. not initial (`"INITIAL"`) or final

[1] https://github.com/google/gson

("FINAL"). We also see that the urgency is "NORMAL", i.e. not urgent ("URGENT") or committed ("COMMITTED"). We have similar constructs for the other location Hungry (L3). Note that all of the locations inside the component are placed in the locations-array, while the initial and final location are placed in the root of the object under properties initial_location and final_location respectively.

Next, let us consider the edge L0 → L2 . This edge has a single nail and no properties. This is described using Lines 26-42 in Code Snippet 8.2. Here we see the source and target locations for the edge (source_location and target_location), followed by the properties on the edge (select, guard, sync, and update). Next, we have an array of nails in the edge (nails). To show the edge properties, they must be bound to a nail, resulting in the attribute property_type becoming one of the following: "SELECT", "GUARD", "SYNC", or "UPDATE". Depending on this, the visual state of the nail will also change (showing the corresponding icon inside the nail). The file contains very similar edge-objects corresponding to edges L2 → L3 , L3 → L2 , and L3 → L1 . All edges are encapsulated in the edges array in the root.

Similarly, the file also contains a list of joins, forks, and subcomponents. All of these work in a very similar way to locations and edges. We invite you to take a look at Code Snippet 15.1, which shows a complete example of a serialized component. Please note that instead of having two arrays names forks and joins, we combined these into the same array, jorks due to easier implementation.

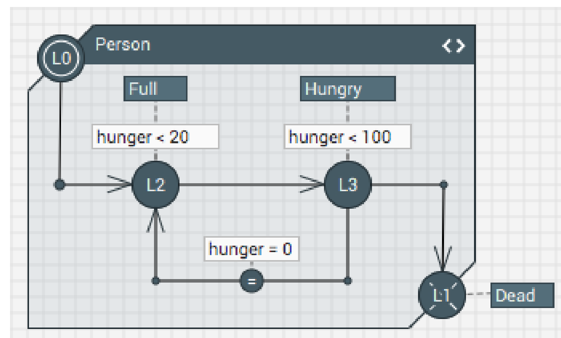


Figure 8.10: Model of the *Person* component.

```
{
  "name": "Person",
  "declarations": "clock hunger;",
  "locations": [
    {
      "id": "L2",
      "nickname": "Full",
      "invariant": "hunger < 20",
      "type": "NORMAL",
      "urgency": "NORMAL",
      "x": 80.0,
      "y": 100.0,
      "color": "0",
      "nickname_x": -20.0,
      "nickname_y": -70.0,
      "invariant_x": -40.0,
      "invariant_y": -40.0
    },
    {"id": "L3", ...}
  ],
  "initial_location": {"id": "L0", ...},
```

```
22    "final_location": {"id": "L1", ...},
23    "jorks": [],
24    "sub_components": [],
25    "edges": [
26      {
27        "source_location": "L0",
28        "target_location": "L2",
29        "select": "",
30        "guard": "",
31        "update": "",
32        "sync": "",
33        "nails": [
34          {
35            "x": 20.0,
36            "y": 100.0,
37            "property_type": "NONE",
38            "property_x": 0.0,
39            "property_y": 0.0
40          }
41        ]
42      },
43      {"source_location": "L2", ...},
44      {"source_location": "L3", ...},
45      {"source_location": "L3", ...}
46    ],
47    "main": true,
48    "description": "",
49    "x": 5.0,
50    "y": 5.0,
51    "width": 280.0,
52    "height": 190.0,
53    "color": "0"
54  }
```

Listing 8.1: Example of a `Person.json` file containing information about the *Person* component.

8.6.3 Queries Example

In Code Snippet 8.2 we see an example of a `Queries.json` file containing two queries. The root element of the file is an array of objects. Each object describes a query, that consists of the strings `query` and `comment`. One could consider expanding this format to also include the latest results of the query, e.g. `"SUCCESS"`, or `"SYNTAX ERROR"`.

```
1  [
2    {
3      "query": "E<> Boss.cups == 3 && Boss.JobsDone",
4      "comment": "Is it possible for the boss to drink three cups of coffee and still get the job
          done?"
5    },
6    {
7      "query": "A<> Boss.JobsDone",
8      "comment": "Is it guaranteed that the boss eventually gets the job done?"
9    }
10 ]
```

Listing 8.2: Example of a `Queries.json` file containing 2 queries.

9 Flattening

The formalism proposed in Chapter 6 allows the users of H-UPPAAL to express systems in a hierarchical way. To be truly useful, this formalism will have to allow for inquiry of properties in respect to the models that it describes, as we know them from UPPAAL. This chapter will describe how we can flatten the component models to networks of timed automata in order to utilize the verification engine from UPPAAL.

9.1 From Components to Templates

In H-UPPAAL, we allow components to be inside other components in the form of subcomponents, where as in UPPAAL all timed automata in the network operate on the same level (defined by the system declaration). As seen in Figure 9.1, the `CoffeeMachine` component consists of the subcomponents `WaterBoiler` and `Roaster` running in parallel.

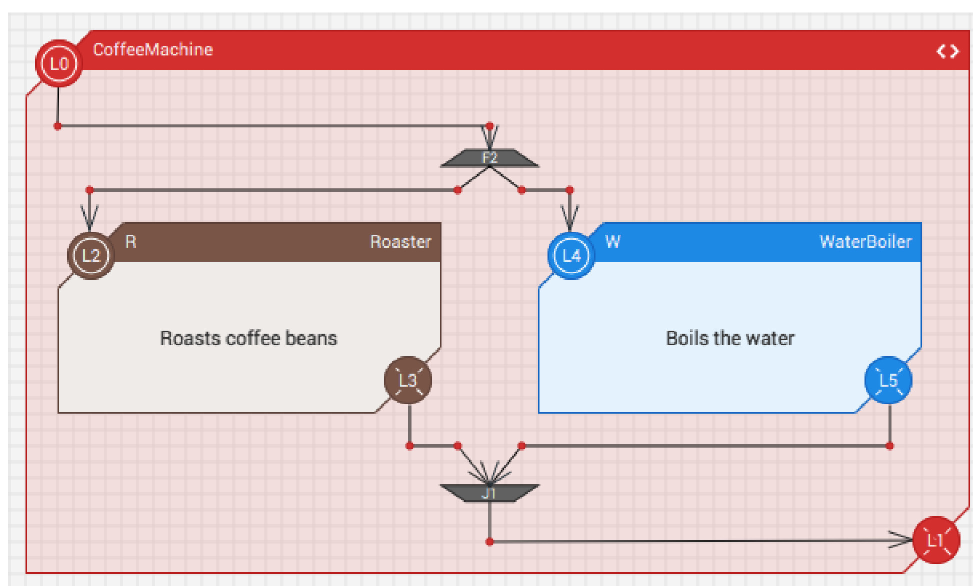
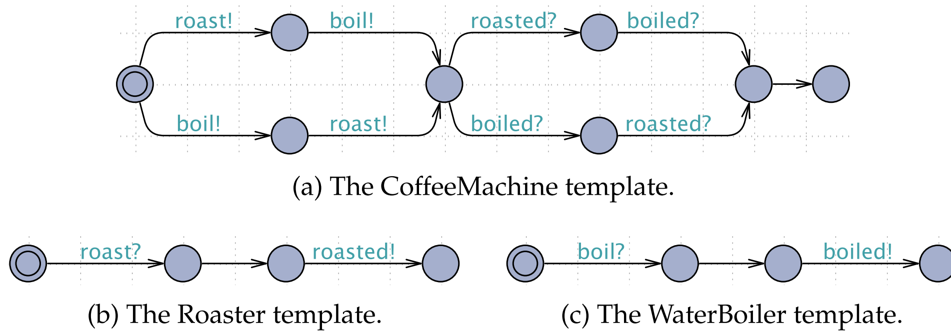


Figure 9.1: The CoffeeMachine component.

In comparison a UPPAAL model, describing the same system, can be modeled using three templates as seen in Figure 9.2. Here there are no direct correlation between the three templates of the system shown in the model, instead, the connection is expressed purely in the communication channels: `roast`, `boil`, `roasted`, and `boiled`.



(a) The CoffeeMachine template.

(b) The Roaster template.

(c) The WaterBoiler template.

Figure 9.2: Templates used to describe a simple coffee machine.

The idea of compiling components and their subcomponents into templates is the core idea behind flattening of H-UPPAAL models. The process starts by creating a template for the component marked as **main** (also know as the root component). And then, for all of the subcomponents of the main component we create a new UPPAAL template. A process for each of these templates are declared in the system declaration, in other words we instantiate an automaton for each template we generate.

Figure 9.3 shows how component C is used in both components A and B. This effectively mean that we should generate two templates for the C-component since they might be started at different times i.e. A starts both c_1 and b_1 in parallel. In this example, we generate a total of *four* templates for the *three* components. Algorithm 1 describes the way templates are generated based on components and their subcomponents.

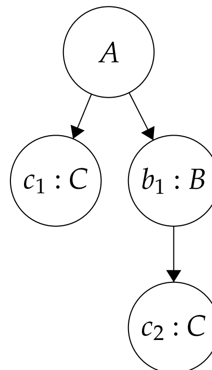


Figure 9.3: Component structure for the main component A.

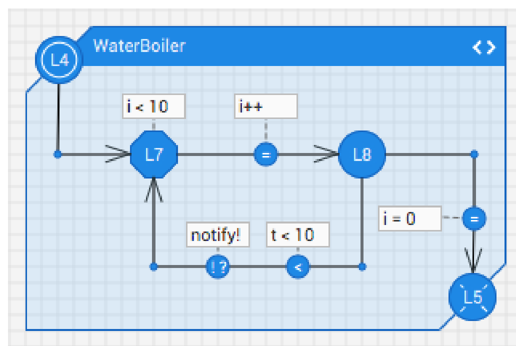
Algorithm 1 — Template generation.

Input: A main component, M

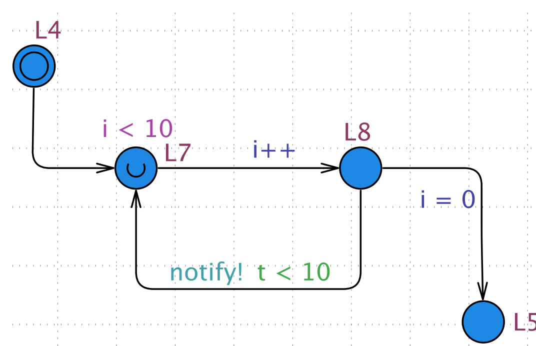
Let S be a set of subcomponents;
Generate a template for M;
$S \leftarrow Sc(M)$;

while $S \neq \emptyset$ **do**
  Pick $s \in S$;
  Generate a template for s;
  $S \leftarrow S \setminus s \cup Sc(T) \mid s : T$;
end

In the flattening process, it is desired to inherently keep all of the locations, edges, and properties from the H-UPPAAL component to the UPPAAL template. This part of the process is pretty straight forward, we take all of the locations in the component and add them to the template. After this, we add all edges between locations and annotate the model with the various properties accordingly. An example of such a simple flattening can be seen in Figure 9.4 where the simple component (Figure 9.4a) is flattened to a simple template (Figure 9.4b).



(a) Simple H-UPPAAL component.



(b) Simple UPPAAL template.

Figure 9.4: Flattening of the simple component WaterBoiler.

9.2 Execution of Subprocedures

We now know that a template for all subcomponents have can be generated. This section will cover how subprocedures are executed by components in the flattened UPPAAL model. Let us start by considering the `CoffeeMachine`-component seen in Figure 9.1. This component starts the subcomponents `R` : `Roaster` and `W` : `WaterBoiler` in parallel, i.e. a subprocedure. The wanted behavior here is that the `CoffeeMachine`-component initiates execution of the two subcomponents, and waits until both of these are done. i.e. `R` is in location `L3` and `W` is in location `L5`. At this point, `CoffeeMachine` can stop waiting for them, and continue.

9.2.1 Starting Subcomponents

We know how we can create templates for subcomponents, so let us now introduce a mechanism for starting them. To facilitate this, we generate a channel for each subprocedure in the system. This will allow us to trigger a synchronization on this channel, which indicates that all templates in this subprocedure should start. For this purpose, a broadcast channel works perfectly. Algorithm 2 describes how these *start*-channels are generated. This algorithm returns a set that is used to determine what channel should start which subcomponents. We can now extend the already generated subcomponent-templates with a new initial location. This new initial location has an outgoing edge to the previous one. This edge synchronizes on the *start*-channel corresponding to the subcomponent. This extension can be seen in Figure 9.5, where we synchronize on the `start0`-channel.

Algorithm 2 — Generation of start-channels.

Input: A set of subprocedures, P

Output: A set, Ch_s, of pairs on the form $(c, start)$, where $start$ is a channel that starts subcomponent c

Let $Sc(p) \mid p \in P$ be the set of subcomponents in subprocedure p;
$i \leftarrow 0$;
$Ch_s \leftarrow \emptyset$;

foreach $p \in P$ **do**
  Generate channel $start_i$;
  foreach $c \in Sc(p)$ **do** $Ch_s \leftarrow Ch_s \cup (c, start_i)$;
  $i \leftarrow i + 1$;
end

return Ch_s

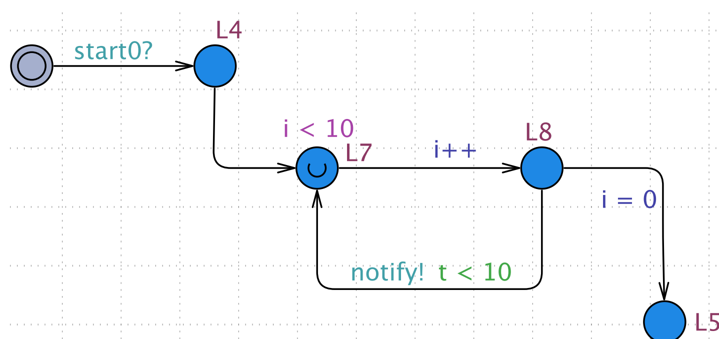


Figure 9.5: Starting the flattened template for W.

9.2.2 Stopping Subcomponents

Now that subcomponents can be started using their corresponding *start*-channel, we need some notion of when they are done, so that the parent component can continue running once a subprocedure concludes. To do this, we introduce a channel `subDone`, which is a broadcast channel. Synchronization on this channel will tell all parents, that are currently waiting on subprocedures, that a subcomponent just finished. However, this is not necessarily one that they were waiting for. This effectively means that we need a way of determining which subcomponent-automaton are done, and which are not. To do this, we introduce a boolean variable for each of the subcomponents in the system, indicating if that particular subcomponent is done or not. For a subcomponent $W : WaterBoiler$, the boolean would be called `isDoneW`. This allows us to, whenever we synchronize on the `subDone` channel, check the status of all the subcomponents that we are waiting for.

What we have to do now, is to update the `isDone`-boolean and to synchronize on the `subDone`-channel when a subcomponent concludes. Remember that, a subcomponent is finished once it reaches its final location. This means that we have to add the update, and synchronization on all edges going into the final location. To achieve this, we introduce two new committed locations, `LE1` and `LE2`. We alter all edges that are going into the final location, to instead go into the `LE1` location. Then, we introduce a new edge, `LE1 → LE2`, where we update the status of the subcomponent. Next, an edge going from `LE2` to the final location is created. This edge contains the synchronization-statement `subDone!`.

To facilitate the possibility of using the same subcomponent-template multiple times, an edge going from the final location to the initial location is also added. The idea behind this edge is to reset the `isDone`-boolean. However, this edge should only be available when all of the subcomponents in the subprocedure are finished. To do this, we introduce a new set of *end*-channels, $Ch_e = \{(c, end_i) \mid (c, start_i) \in Ch_s\}$. The generation of this set is done just as the one for starting them, seen in Algorithm 2. All of these new *end*-channels are, like the *start*-channels, broadcasts. When all subcomponents are done executing, synchronization on these channels will inform all of them to reset, i.e. go to the initial location, and wait to be started once again.

All of these additions can be seen in Figure 9.6, which shows the subcomponent-template for `WaterBoiler`, where the `isDoneW`-boolean is set to true just before the subcomponent reaches its final location, `L5`. In the figure, you also see that the boolean is reset once the subcomponent synchronizes on the `end0`-channel.

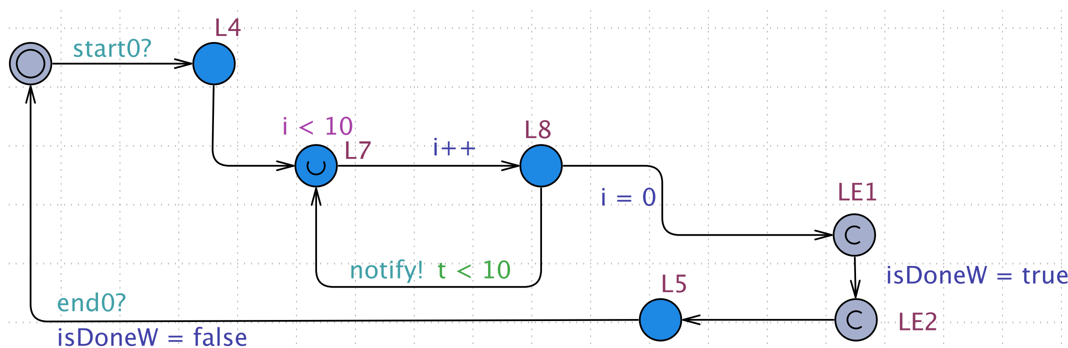


Figure 9.6: The final flattened template for W.

9.2.3 Starting Subprocedures

With all of the groundwork laid out, starting a subprocedure is now possible. All that has to be done in order to start the subprocedure, p is:

1. Broadcast on channel $start_i \mid (c, start_i) \in Ch_s,\ c \in Sc(p)$
2. When synchronization on the `subDone` channel happens: Continue if all subcomponents $c \in Sc(p)$ have finished executing, otherwise, continue waiting.
3. Broadcast on channel end_i

To realize this, four new locations are introduced, namely `Enter`, `Waiting`, `Exiting`, and `Exit`. All incoming edges to a subprocedure are now updated to go to into the `Enter`-location instead. Likewise, all the outgoing edges are updated to originate from the `Exit`-location. This can be seen in Figure 9.7. Four new edges are also introduced:

`Enter → Waiting`, which is responsible for broadcasting on the correct *start*-channel, indicating to subcomponents involved in the subprocedure to start.

`Waiting → Waiting` and `Waiting → Exiting` which are responsible for checking if all of the subcomponents involved are finished. If at least one subcomponent is unfinished, only the first edge should be enabled, and when all subcomponents are finished, only the second edge should be enabled.

`Exiting → Exit` is responsible for broadcasting on the *end*-channel corresponding to the *start*-channel, indicating that subcomponents involved should reset.

These four new locations, and four new edges can be seen in Figure 9.7, which shows how the flattened `CoffeeMachine` starts its subprocedure. Here, we see the *start*- and *end*-channels, `start0` and `end0`. We also see the booleans `isDoneR` and `isDoneW`, which indicate if their respective subcomponent is done.

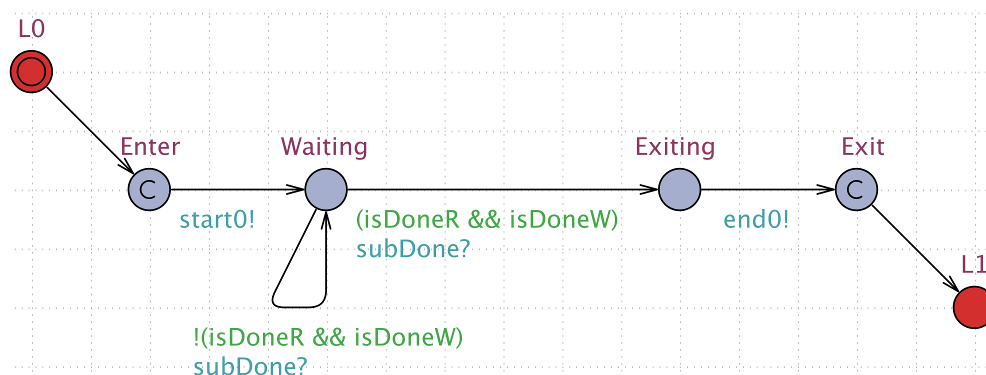


Figure 9.7: Component CoffeeMachine as a UPPAAL template.

10 Queries

With the addition of subcomponents, a new type of inquiry appears e.g. "*Does component C contain a deadlock?*". Let us consider the component in Figure 10.1. Here we see a relatively simple component, with three locations. Notice that location `L2` is urgent. Using this component may result in a deadlock in case where synchronization on the channel `communicate` is not possible.

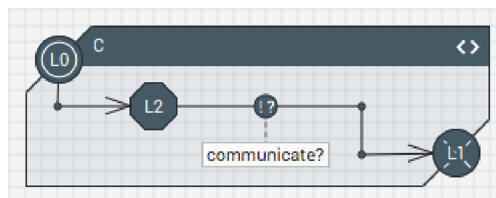


Figure 10.1: Component with a possible deadlock.

These types of inquiry are not immediately easy to answer, since we do not know where the component is used, or how many instances of the component is used. To see how we can formulate new queries using UPPAAL query-syntax, the Example 2 is formulated. This example will be used in the following sections. Figure 10.2 shows the relation of the components. Note that in this chapter we use the term process, which from the flattened UPPAAL model, corresponds to a subcomponent.

Example 2 We have three components, A, B, and, C with the following configuration:

Component (x)	A	B	C
$Sc(x)$	$\{b_1 : B, c_1 : C\}$	$\{c_2 : C\}$	$\emptyset$
$Ic(x)$	$\emptyset$	$\{b_1\}$	$\{c_1, c_2\}$
$Loc(x)$	$\{l_0, l_1, l_2, l_3\}$	$\{l_4, l_5\}$	$\{l_6, l_7, l_8\}$

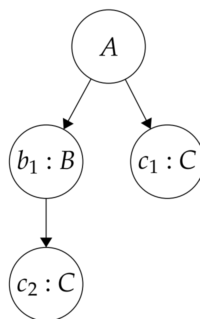


Figure 10.2: Component structure.

10.1 Single Subcomponent Queries

Let us first consider queries that is only concerned with a single subcomponent instance, e.g. b_1. We might want to answer questions such as "*Is location l_4 reachable in subcomponent b_1?*". From Section 9.1, we know that H-UPPAAL will generate a specific template for each subcomponent. Using this knowledge, we are now able to write the query $\exists \diamond b_1.l_4$, which will check if there is a path leading to location l_4 in subcomponent b_1.

Another type of question we might want to ask could be "*Does the subcomponent b_1 contain a deadlock?*". Before looking at how we can answer this type of question, let us first look at how we find a deadlock. Normally we would run the query $\exists \diamond$ `deadlock`. However, this would check if there is a deadlock anywhere in the system, whereas we only want to consider the subcomponent. We know that we can check for a deadlock in a location (l) in the process (p) using the query $\exists \diamond p.l \land$ `deadlock`. All we have to do now, is to expand this query to consider the set of location for a given component:

$$\exists \diamond (\bigvee_{l_i \in Loc(C)} p.l_i) \land \texttt{deadlock}$$

Considering the example, where we would like to know if there is a deadlock in the b_1 subcomponent, the query would look like this:

$$\exists \diamond (b_1.l_4 \lor b_1.l_5) \land \texttt{deadlock}$$

10.2 Multiple Subcomponent Queries

Let us now consider queries regarding all instances of a component. From Example 2, this could be all instances of the component C, namely c_1 and c_2. In this context, questions might sound like "*Does any of the C-subcomponents contain a deadlock?*"

Before answering this question, let us first consider how to answer the intermediate question: "*Does any of C-subcomponents contain a deadlock in location l_7?*". To solve this, we use an approach which is similar to the one used in Section 10.1. However, now we have to check all instances of a specific component. Let us consider that we want to check if we can reach the location l_7 in component C. To do this, we would find the set of all subcomponents of C, which is given by $Ic(C)$. We know that all of these have their own template and processes. We also know that all of these templates contain the same locations. So to formulate the final reachability query, we would have to concatenate a reachability query for every subcomponent instance. This results in the following query, where l and C are the location and component which we consider.

$$\exists \diamond (\bigvee_{p_i \in Ic(C)} p_i.l) \land \texttt{deadlock}$$

If we consider the previous example, where we ask if there exists a deadlock in the location l_7 in any of the C-subcomponents, this would yield the following query:

$$\exists \diamond (c_1.l_7 \lor c_2.l_7) \land \texttt{deadlock}$$

Now we know how to check for a deadlock in a specific location of a subcomponent, so all we have to do now, is to expand the query to consider all the locations in the component, given by the function *Loc*. Using this, we expand the query to:

$$\exists \diamond (\bigvee_{p_i \in Ic(C)} \bigvee_{l_i \in Loc(C)} p_i.l_i) \wedge \texttt{deadlock}$$

Considering the original question, which was to check if any of the subcomponents of C contains a deadlock, would yield the following query:

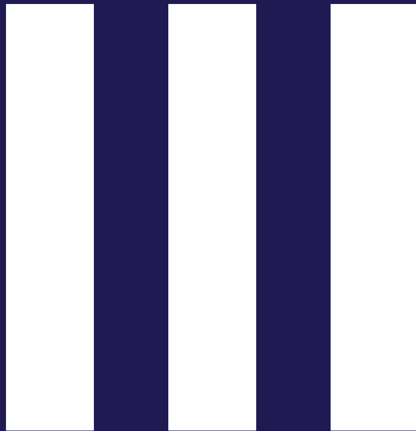
$$\exists \diamond (c_1.l_6 \vee c_2.l_6 \vee c_1.l_7 \vee c_2.l_7 \vee c_1.l_8 \vee c_2.l_8) \wedge \texttt{deadlock}$$

10.3 Long Queries

It is clear to see that the size of these queries grows with the number of locations in a component and the number of subcomponent instances. Depending on the severity, this might cause problems with readability and might cause users to give up on understanding the query. Imagine having $\{k_0, k_1, k_2 \dots k_9\} : K$ where $Loc(K) = \{l_0, l_1, l_2 \dots l_9\}$. This would result in a deadlock-query considering 100 different locations:

$$\begin{aligned} \exists \diamond (&k_0.l_1 \vee k_0.l_2 \vee k_0.l_3 \vee k_0.l_4 \vee \dots \vee k_0.l_9 \vee \\ &k_1.l_1 \vee k_1.l_2 \vee k_1.l_3 \vee k_1.l_4 \vee \dots \vee k_1.l_9 \vee \\ &k_2.l_1 \vee k_2.l_2 \vee k_2.l_3 \vee k_2.l_4 \vee \dots \vee k_2.l_9 \vee \\ &\dots \\ &k_9.l_1 \vee k_9.l_2 \vee k_9.l_3 \vee k_9.l_4 \vee \dots \vee k_9.l_9) \wedge \texttt{deadlock} \end{aligned}$$

One could imagine that most people would find this type of queries too tedious to write manually, and we have, as mentioned in Section 8.1, created some shortcuts for common queries, like "*Does K contain a deadlock?*" that will produce this type of query.



Final Thoughts

11 Evaluation 67
11.1 Working with H-UPPAAL
11.2 Adhering to the Manifesto
11.3 Thoughts on the Development
11.4 Involving Users

12 Conclusion 77

13 Further Work 79
13.1 Error and Warnings
13.2 Periodic Queries
13.3 Version Control of JSON-files
13.4 H-UPPAAL Specific Queries
13.5 Printability
13.6 Refactoring
13.7 Utilize the UPPAAL Engine

11 Evaluation

The intention of this chapter was to convert the models of the article "Compositional Schedulability Analysis of An Avionics System Using UPPAAL" [Boudjadar et al., 2014] mentioned in Chapter 3. With the lack of time left in the project and lack of functionality in the current version of H-UPPAAL this was simply not done. However, we still want to give some sort of evaluation of what have been done to progress towards the goals described in Chapter 4. In order to justify that there have been done some progress in this direction we have constructed a semi-complex system, that shows how we can use the tool to develop hierarchical models using H-UPPAAL. This chapter includes a showcase of such an example where we discuss how it can be modeled using the tool, and how we can query it. We then evaluate how the tool, in general, cohere with the conjured manifesto, and reflect upon the development of said tool. This chapter concludes with some thoughts on how one could evaluate a tool like this by involving users.

11.1 Working with H-UPPAAL

This section will show how hierarchies can be used to model a life-like environment. We will also show how to verify specific properties in this environment. Throughout this section, we will consider Example 3.

Example 3 – The Amaryllis Restaurant. 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 distribute the dishes amongst the customers, Gordon has hired Sofia, who will be the waitress in the restaurant. However, since all dishes are surprises, Sofia has no chance of knowing which customer ordered what, resulting in her delivering a dish to whatever table she feels like.

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. Because of the limited crew, the kitchen can only handle one order at a time. 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.

The Amaryllis restaurant is visited by 2 customers, Ace and Bruce. When making an order, the customer will grow impatient after waiting 30 minutes without receiving a dish. However, a customer may, after finishing one dish, request another. The portion size in Amaryllis is not very big, resulting in customers being able to consume an infinite amount of dishes without getting full.

Finally, Gordon would like an answer to the following questions:

1. Is it always possible to satisfy all customers?
2. If not, is it always possible to satisfy some?

Based on Example 3, we will now have a look at how we can model this environment using H-UPPAAL. We start by considering the system as a whole. We have a restaurant that have some employees and some customers. These employees each have different responsibilities, Gordon, and his line cooks must prepare the dishes where the waitress will interact with both the customer and the kitchen. To model this we create have the red `Amaryllis`-component seen in Figure 11.1. This component is introduced in order to model that the customers, the waitress, and the kitchen acts in parallel with each other. Notice that we have two subcomponents of the `Customer` component since Ace and Bruce have the same behavior.

We want to model a customer that can order a dish, wait for his food, grow impatient or get his dish where he can either be full or go back and order a new dish. The state of the customer is reflected in the `Ordering`, `Waiting`, `Angry` and `Full` locations as seen in the blue `Customer`-component in Figure 11.1. The edges of this component are annotated to model the way these customer grows impatient and becomes angry. When a customer orders some food (goes from `Ordering` to `Waiting`) we reset a clock called `patience`. To model how a customer gets angry is done via the invariant `patience <= limit`, and the guard `patience == limit` on the edge from `Waiting` to `Angry`, where `limit` is how long the customer will wait, which in this case is set to 30. By this component, we can now model the two customers in the `Amaryllis` component and let them have the same behavior.

Our customers are happy to be served by Sofia the waitress. Her job is to take orders, notify the kitchen and bring back the dishes to the hungry customers. This behavior has been modeled as seen in the pink `Waitress`-component in Figure 11.1. The job description for Sofia is reflected by her three locations, `Ready`, `TakingOrder` and `GettingFood`. The loop towards `TakingOrder` is started by a customer, using the channel `order`, which enforces Sofia to tell the kitchen to cook a meal, using the channel `cook`. The loop towards `GettingFood` is oppositely when the kitchen tells her that a dish is ready, using the channel `cooked`, where she is forced to deliver the food to a customer over the channel `receive`.

Let us have a look on how we can model Ramsay's kitchen. We know that Gordon is the chef of the kitchen and that he has two line cooks employed to prepare his famous dishes, as seen in the green `Kitchen`-component in Figure 11.2. When the kitchen starts it goes to the `L25`-location where it is ready to work. Here the kitchen awaits orders from the customer, which is brought to them by Sofia, using the channel `cook`. When this happens Ramsay and his line cooks start working on the dish, which is modeled as the subprocedure consisting of a `Ramsay`-subcomponent and two `LineCook`-subcomponents. When this subprocedure ends, the kitchen returns to the `NewOrders`-location, where the kitchen can either just give the dish to Sofia (return to `L25`) or they can be asked to cook again while given Sofia the recently prepared dish (starting the subprocedure via the `L38`-location).

Now we know how the kitchen is asked to prepare dishes, we can take a look at how Gordon manages the kitchen. The behavior of chef Gordon Ramsay can be seen in the orange `Ramsay`-component in Figure 11.2. This component shows how we model his instructions of the line cooks in preparing his dishes. He starts out by reaching `L30`, where he has three options to proceed. The first option is to see progress on the dish, which

is represented in the edge looping directly back to `L30`, here he gets notice of progress using the channel `work` and he increments `p` which is an integer keeping track of the progress. The second option is that he grows impatient, this is modeled with the edge to the `Impatient`-location, where he yells at his cooks, using the channel `yell`. He will always yell at his employees if no progress have been made for 2 minutes, modeled by the invariant `patience < 2`. When he yell at his workers he expects progress immediately after since the `Impatient`-location is urgent. Lastly, all 10 steps in the meal have been prepared, Gordon now goes to his final location (`L27`) and tell his cooks that the dish is done.

The last behavior we need to model is how the line cooks `Adam` and `James` work. Their behavior has its root in the teal `LineCook`-component in Figure 11.2. These lazy line cooks can either be cooking and making progress while staying in the `Cooking`-location. They can also be motivated by Ramsay when they get yelled at, using the channel `yell`, here they are forced to immediately work, since the location `Motivated` is committed. This location is committed to ensure that if the dish is completed Adam and James must be ready to go to their final location `L29` when told by Ramsay using the channel `complete`.

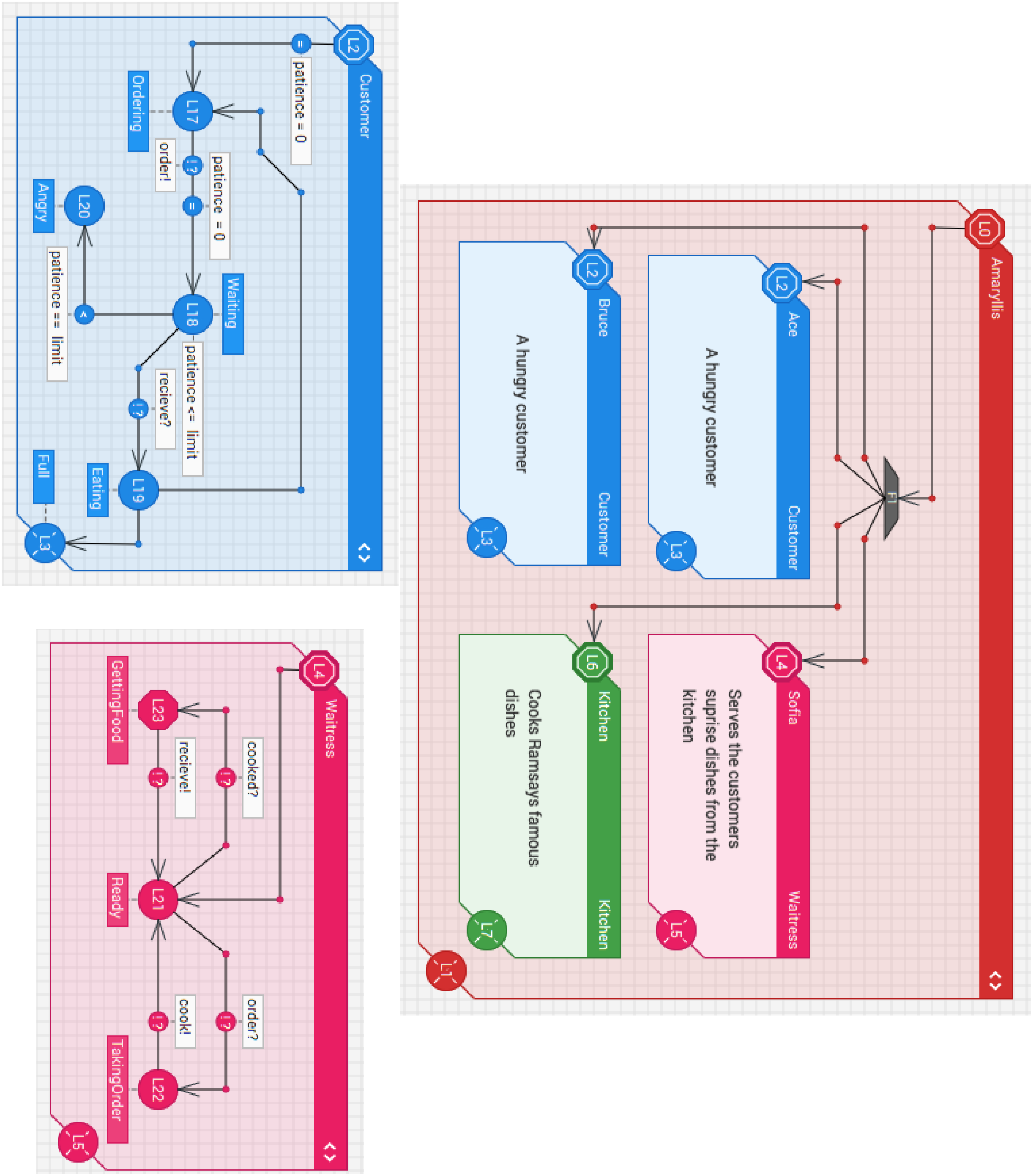


Figure 11.1: The Amaryllis, Customer and Waitress components.

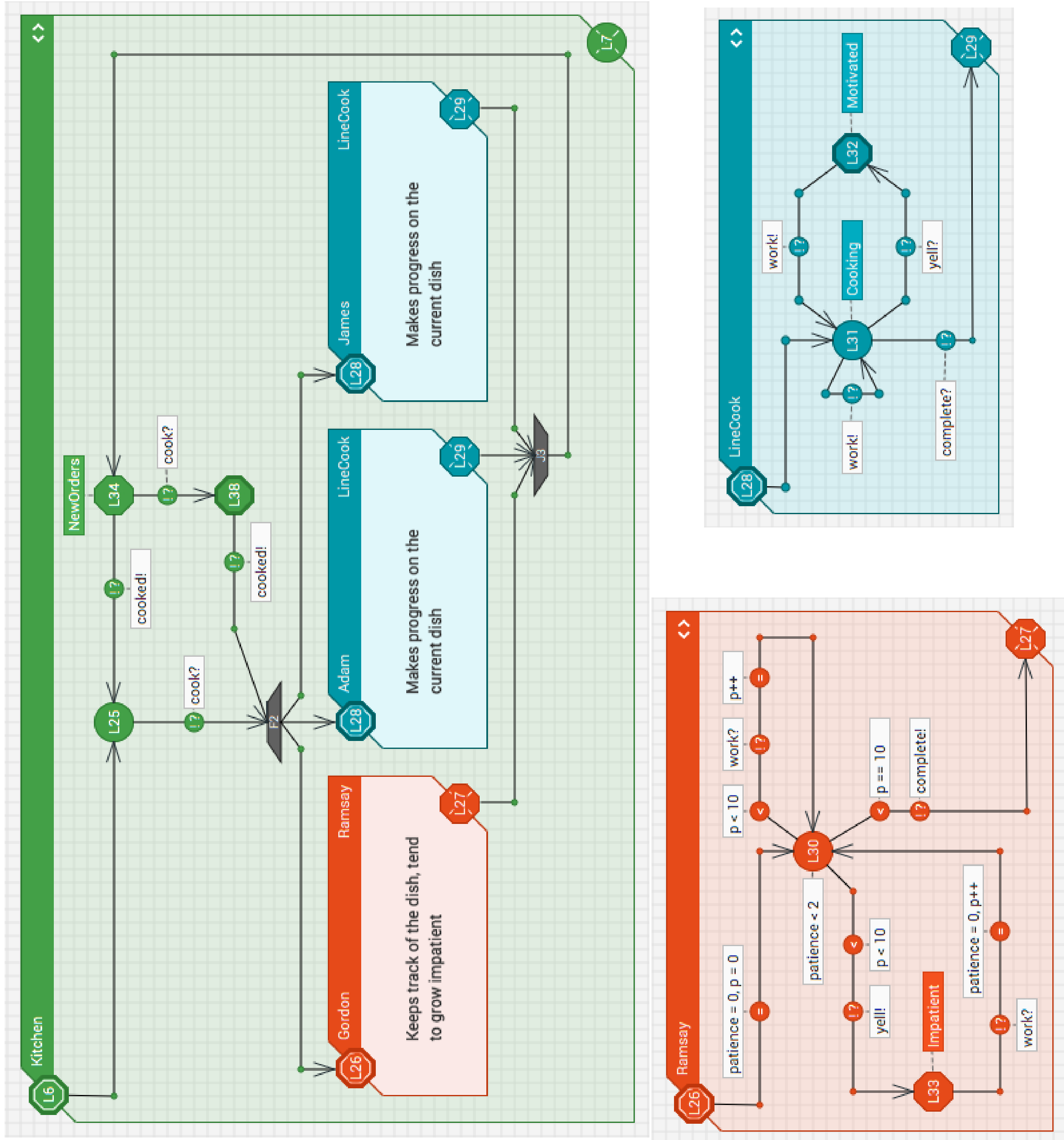


Figure 11.2: The Kitchen, Ramsay and LineCook components.

11.1.1 Possibility for angry customers

Now that we have modeled the Amaryllis, we can now start answering Gordons questions. He wants to know no matter what, his customers will always be satisfied, in other words, he would like to know if any customer can get angry, i.e. reaching the `Angry` location in the `Customer` component. H-UPPAAL allows us to simply use the build-in reachability checker by right clicking the location and pressing `Is Reachable?` in the drop down menu, as seen in Figure 11.3. This will result in the following query being generated:

```
E<> Ace.L20 || Bruce.L20
```

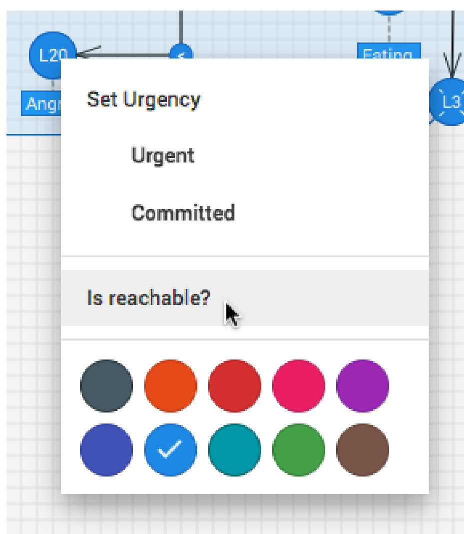


Figure 11.3: Context menu reachability query helper.

Note that the query uses the unique identifiers rather than the descriptive one, since we are still using classical UPPAAL query language. In Section 13.4 we discuss the possibility for creating unique queries for interacting with the new H-UPPAAL constructs.

Running this query shows that it is fact possible for at least one of the customers to become angry. Using UPPAAL, we were able to find a trace supporting this. This is possible when one of the customers *starves* the other by repeatedly ordering new dishes, while the waitress keeps delivering the dishes to the "evil" customer.

11.1.2 Possibility for satisfying customers

To check if it is possible to satisfy all customers are similar to checking if any customer can get angry. However, instead of checking if location `Full (L3)` is reachable by any of the subcomponents, we ask if there exists a trace, where both of the subcomponents are in location `Full (L3)`, resulting in the following query:

```
E<> Ace.L3 && Bruce.L3
```

Running this query showed that there is, in fact, a state where both the customers are satisfies. We used UPPAAL to take a look at one of these states and explored the trace leading to it. Based on this, we now know that customers might get angry or satisfied depending on how Sofia delivers the orders.

11.1.3 Contains Deadlock

It might also be interesting to see if the model we generated contains any deadlock, other that the obvious, intentional deadlocks. Namely the ones in locations `Full (L3)` and `Angry (L20)`. Normally, when checking for deadlocks, you would run a query similar to `E<> deadlock`. Doing this, would find the deadlocks in said locations, so we have to ignore these. This can be done using the following query:

```
E<> deadlock && !(Ace.L3 || Bruce.L3 || Ace.L20 || Bruce.L20)
```

Running this query shows that the system contains no other deadlocks, meaning that other components, such as `Kitchen` and `Waitress` will always run unless the `Customer` is angry or have finished eating.

11.2 Adhering to the Manifesto

Now that we see how the tool works, let us go back, and consider how we conform with the 6 principles defined in our manifesto in Chapter 5. The purpose of these is to steer the development of the newly developed tool. This section argues, to what extent we deem that H-UPPAAL adhere to these principles.

11.2.1 Principle 1 (Backward Compatible)

Most of the concepts in H-UPPAAL, e.g. *urgency* and *declarations*, have originated from the UPPAAL tool. We believe that this is a helping hand for users that are used to working in UPPAAL and might help ease the transition between the two tools. Besides this, a lot of effort have been put into ensuring that the new models look familiar to the ones in UPPAAL. However, the concept of *templates* is not present in H-UPPAAL directly, since it is replaced by the *component* concept. This might confuse some users and will probably make the transition a bit harder. Another potential problem is that users are forced into building hierarchies due to the single *main component* idea i.e. there is no system declaration where the user can specify which components should run in parallel. Even though the introduction of components caused us to remove the template-idea from the tool, we deem that H-UPPAAL adheres quite well to Principle 1 (Backward Compatible).

11.2.2 Principle 2 (Integrated Development Environment)

Some features inspired by IDEs have been added to show that this source of inspiration can have great potential. Features like allowing users of the tool to query for properties like "*Is there a deadlock in this component?*", with little to no effort, have been implemented. Furthermore, we have a lot of ideas for features that will help to alleviate mundane or tedious tasks. Based on this, we believe that we have created a good basis for adhering to Principle 2 (Integrated Development Environment).

11.2.3 Principle 3 (Information Hiding)

One of the core concepts introduced in this project was the idea of hierarchies of timed automata components. This structure allows users to break functionality down into subcomponents. This enables hiding of complexity like a black box, with only a description showing the functionality of the subcomponent. We have not yet implemented any way of "opening" a subcomponent from within a component, which will require the user to find the component in the project panel manually, however, there should be no problem in adding such a feature in the future. Furthermore, by allowing users to expand and

collapse both the project and query panels, we are able to utilize almost the entire screen for the model canvas. Based on this, we deem that the project adheres to Principle 3 (Information Hiding).

11.2.4 Principle 4 (Identity and Relation)

To allow collaborators to easily communicate information regarding a model, we have introduced a concept of *unique* and *descriptive* identifiers for locations. Unique identifiers ensure that all locations have a name, and descriptive identifiers allow users to describe a location with a specific name. It is not yet possible to use these descriptive identifiers in queries, but we intent do make this possible in future work on the project. We have also ensured that all components and subcomponents have unique names, which also should make communication of information between collaborators easier. Based on all of this, we deem that the project adheres to Principle 4 (Identity and Relation).

11.2.5 Principle 5 (Printable)

The H-UPPAAL tool does not, in this state, have a feature that allows users to export the models that can be used for papers. In general, the only way to print models at the moment is using screenshots, this is obviously not enough to satisfy this principle. However, there have been put effort into clearing up the relation between properties and objects in the model. An example of these efforts could be the way labels are anchored to the objects that they describe, which clarifies the ownership of these properties. This is the principle that have been put the least effort into, and we deem that we do not adhere to Principle 5 (Printable). However, we do believe that we have not constrained the tool from adhering to this principle in the future.

11.2.6 Principle 6 (Objects Require Space)

Little to no measures have been made to ensure the elements take up space. This effectively mean that elements such as locations and nails can be put on top of each other. It is even possible of creating edges that run on top of each other. The only thing currently implemented is a measure to ensure that locations and edges are created inside of the component. All of this effectively means that the project, in its current state, does not adhere to Principle 6 (Objects Require Space).

11.3 Thoughts on the Development

In the early stage of the development, most of our efforts went into investigating the UPPAAL tool and researching previous work on the subject of hierarchies in model checking. We quickly realized that a new tool in this domain could be introduced. For this, a canvas where we could draw the new models was required. Based on inspiration from modern IDEs we wanted this canvas to be the center of the tool and not just a tab for editing as in UPPAAL. From here on we spent a lot of time trying out design ideas in regard to how we could visualize a hierarchical structure of timed automata. Many of the design considerations have been discussed in Chapter 7. However, many ideas and prototyped elements have not been included in the tool or the report for that matter. In this stage of the development we were trying to be as creative as possible, but found it necessary to try to implement them, to some degree, into the tool to get a feel of how they could work together with the rest of the tool. We have been pretty confident from the beginning of the semester that it would be possible to flatten a hierarchical version of UPPAAL since some papers and other internal notes have been produced on the subject. For this reason our

attention have been focused around how you could improve the user interaction, while introducing this new structure. This resulted in a mere draft of a formalism, but also resulted in a rather stable tool that does this exactly. We are rather pleased that we had the time to include some flavor from IDEs and is convinced that these will show to have some positive effect in one way or the other.

11.4 Involving Users

We have as of now, only presented or involved the tool for users that are biased in one way or the other. For this reason, this section will discuss how we could involve users and hopefully get some actual user evaluations that can be helpful to determine the state of the project, but also guide it to become the best it can possibly be. In general, we want a tool that is usable, and for that reason, we want to involve users in evaluating the tool. The following sections discusses how we can prepare evaluations with various purposes. These evaluations should give detail to some of the concerns we have, regarding the usability, and clarify the purpose of the tool. Besides this, each of the evaluations will most likely also help us find some bad smells and, in general, spot some usability issues that this young tool contains.

11.4.1 Finding a New and Better Interaction

Because H-UPPAAL is, to some extent, simply a redesign of the interaction with the UPPAAL verification engine, we want this interaction to be as good as possible. This is especially important inside the modeling canvas, where the behavior of the systems are described. We have during the development had some ideas in respect to the modeling language. We want to find the best syntax for the modeling, in terms of values like familiarity, readability, and writability. To steer towards a better syntax for modeling, we intend to perform A/B testing [Nielsen, 2005] where two versions, of the system, are tested. By doing this we can test opposing ideas against each other, and make progress by determining if one version of the system is better than the other. In other words, we will compile two versions of the tool, one version with one set of ideas implemented, another version with another set of ideas. By having a group of subjects perform the same tasks in each version we will acquire some insight regarding these ideas in comparison.

11.4.2 Comparing Across Experts to the Novel

It is desired that H-UPPAAL is useful and usable for both experts of model checking, but also for novels in this field. For this reason, we intend to do some qualitative analysis involving a selective number of users in the span of the completely novel to the domain of verification, to the scientist that on a regular basis uses model checking for research purposes. The main purpose of this analysis is to indicate issues with respect to the learning curve of the tool. At the same time, this analysis should also point out issues where the system is lacking in terms of utility for expert users. Because this span of users have varying knowledge and skill, in this domain, the analysis would have to be fit the skill level of the individual subjects. A simple outline for this type of qualitative analysis could be to use the think-aloud protocol [Nielsen, 2012] and have the participant do a modeling exercise, that fits their skills. Novel users could, for instance, be tasked with modeling one of the infamous coffee machine while expert users could be tasked with modeling a more complex problem like a communication protocol. Regardless of the participant, we could get valuable insights for H-UPPAAL in both respects with lowering the learning curve while keeping up with the required utility.

11.4.3 Comparing to UPPAAL

One of the core user evaluations we intend to perform is a comparison between the H-UPPAAL tool and UPPAAL tool. We would like to construct a test of some sorts where we can figure out if the introduction of new concepts in H-UPPAAL creates value. It is still a bit uncertain what form this type of evaluation should have. One suggestion is to perform an A/B test, where one partition uses H-UPPAAL and the other uses UPPAAL. This would measure if certain jobs are easier, or less tedious, in one of the two tools. But since the newly developed tool is far from as mature as UPPAAL, this might not be a fair comparison. Another way to investigate this, could be to have participants use H-UPPAAL to do some task. After this, let them do the same task in UPPAAL and ask if they found hierarchies or IDE features useful regardless of the overall usability of H-UPPAAL. The ladder is a more fair comparison when the goal is to figure out if the new concepts create value.

12 Conclusion

Model checking is a technique used today for various purposes. With tools like UPPAAL, this technique allows users to check models of a system for certain properties in a rather efficient way. This tool is mostly used in research, as a tool to produce and prove the correctness of new ideas in different research areas, but have also shown to be useful when developing time critical software to guarantee properties on the conceptual level of system design.

We looked at UPPAAL and identified some issues with the tool. We found that models can become rather complex, effectively compromising the comprehensibility. This issue has previously been addressed with the introduction of hierarchical formalisms. The idea is to use the powerful mechanisms found in hierarchical modeling to achieve a greater modularity, better encapsulation, and, in general, a higher abstraction of a system. When looking at UPPAAL, we also discovered areas of concern in regards to the usability of the tool and saw areas that could be improved or simplified with the introduction of features commonly found in IDEs.

This motivated us to explore how a new tool that includes a hierarchical structure and IDE-inspired features could add value to UPPAAL users. To steer the project, and ensure that the issues found are addressed, a manifesto consisting of 6 principles have been formulated. Guided by this manifesto, we have introduced and created the H-UPPAAL tool which is based on these principles which should make the tool easy to use, facilitate better communication between collaborators, and provide users with a better overview when modeling. The tool is inspired by IDEs and facilitates a better environment for performing different tasks in regards to model checking, for instance generation of specific queries.

The modeling language of H-UPPAAL is strongly inspired by previous work on making UPPAAL hierarchical. We have not yet formulated formal semantics for it, but intend to do so in the next semester. We are, however, able to flatten hierarchical models into networks of timed automata used in UPPAAL. This allows us to access the power that lies in the UPPAAL verification engine, which comes from years of research and development.

To see if H-UPPAAL is true to the manifesto, a semi-complex model has been developed using the tool. This showed that the tool does not entirely adhere to the manifesto. We do, however, believe that the tool creates a solid foundation for further development, where it is possible to investigate if hierarchies and IDE features can create value for users.

We believe that introducing the component concept into the world of model checking will allow for better control and overview of complex models. By creating a tool inspired by modern IDEs, we start to see a tool form, where common programming practice such as reuse of code, refactoring, and code analysis becomes an easier task.

13 Further Work

This chapter covers some of the thoughts and ideas, we have had regarding this project, which we would like to realize when continuing work on the 10th semester. This chapter covers ideas regarding IDE features, an extension of the query language, but also covers some shortcomings in the project that we would like improved.

13.1 Error and Warnings

Currently, in H-UPPAAL, the number of different errors and warnings is very limited. We would like to introduce more errors and warnings that will help modelers during development and in the debugging of models. This section cover some ideas we have had in regards to both errors and warnings.

13.1.1 Markings in the Model

Currently, when getting an error, you are given a message similar to:

Nicknames for locations must be alpha-numeric. Near: Location a@a (L3)

Even though the error describes precisely what is wrong, and where it is wrong, it might be more intuitive for users to visually see where, in the model, the problem occurs. One could imagine that hovering the mouse over the error would open the component where `L3` is placed, and highlight the location, making it trivial to find the location. In this particular case, one could imagine that clicking the error would initiate a refactoring process where the user is prompted for a new nickname for the location (see Section 13.6.3).

13.1.2 Continuous Error Checking

In the current version of H-UPPAAL, most errors come from the verification engine, meaning that we do not catch most errors before actually running a query. This might be a problem, since the period between running queries might be long, causing a lot of changes to happen before realizing that an error has been introduced. We propose implementing a continuous error checker that will give the modeler faster feedback when introducing an error, like syntax checkers in programming IDEs.

13.1.3 Warnings

In the current system we have prepared for, but not implemented any warnings. We would like to introduce some checks that could help developers spotting bad smells in the model. We have compiled the following list which reflects some of the ideas we would like to introduce warnings for:

- Location l does not have any incoming edges
- Some edges going from l_1 to l_2 is redundant
- Guard g is always true/false
- Channel c does not have a sender/receiver
- Component/channel c is never used

13.2 Periodic Queries

We find ourselves writing queries that we would run every once in a while. One example of such a query is `E<> deadlock`, which checks if our model contains a deadlock. However, we often find that the outcome of this query is the same as the last time we ran it. The idea is then to automate this progress, only informing the user once the status of one of his queries changes from the expected to unexpected e.g. "*The query `A<> Person.L2` went from being successful to being unsuccessful*". This is very similar to unit testing and continuous integration.

Similarly, you sometimes find yourself having a "debug query" that you would run right after changing the model to see how the change impacts properties in the model. You might do this several times before being satisfied with the outcome. We believe that, like errors and warnings, continuous status updates on specific queries might be useful. We suggest that you could mark specific queries to continuously provide the user with status updates, like the concept of watches when debugging in modern IDEs. Imagine the query pane opening and automatically running the selected queries once a change has been introduced.

We believe that both of these ideas could help developers to spot bad smells in the model properties faster than they normally would.

13.3 Version Control of JSON-files

We put a limited effort into designing a solid format for storing the models in H-UPPAAL persistently. However, since the communication and the collaboration of the design in this tool is important, this format might have to undergo some changes to become more friendly towards version control technologies like git or SVN. We have not investigated if the implemented format causes any issues with respect to this, however, some effort should be put into this so that sharing models of H-UPPAAL can become a seamless experience.

13.4 H-UPPAAL Specific Queries

As mentioned in Chapter 10, the introduction of components and subcomponents have caused the query language to become somewhat inadequate in some situations. This section will take a look at how the query language could be extended to better accommodate these situations.

In Section 10.3 we describe how it is possible to formulate queries that will answer questions such as "*Does component* `Person` *contain a deadlock?*", but unfortunately, these queries can become quite long, and hard to maintain. To shorten these queries, one could draw inspiration from the `forall` construct, which is already in the query language of UPPAAL. This construct allows the modeler to iterate over a specific type. Let us consider the following query from the train gate example, distributed with UPPAAL. In this query, we check if it is possible for `Train(0)` to be in the `Cross` location while all other trains are in the `Stop` location, i.e. waiting to cross.

```
E<> Train(0).Cross && (forall (i:id_t) i != 0 imply Train(i).Stop)
```

What we propose is something similar, but instead of iterating over a type, you would iterate over subcomponents. Let us consider the components and subcomponents defined in Example 4.

Example 4 We have three components, A, B and C, where:
- $Sc(A) = \{b_1 : B, c_1 : C, d_1 : D\}$
- $Sc(B) = \{c_2, c_3\} : C$
- $Sc(C) = \varnothing$
- $Sc(B) = \{c_4, c_5\} : C$
- $Loc(C) = \{l_0, l_1, l_2\}$

A visual representation of the component structure can be seen in Figure 13.1.

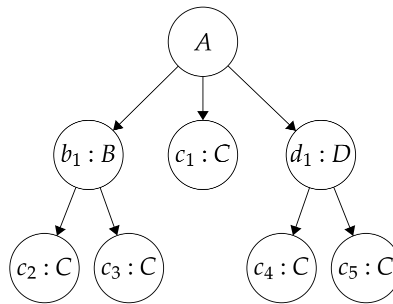


Figure 13.1: Component structure.

With this system, we might be interested in asking if it is possible to reach location l_1 in all, or any, of the subcomponents of type C. With the current UPPAAL syntax, this is done by writing a query that checks if any of the processes generated from the component can reach that specific location. As an alternative to this, one could consider implementing syntax similar to the one in Figure 13.2, where Figure 13.2a will check if all of the subcomponents can be in location l_1 at the same time, and Figure 13.2b will check if it is possible for any of the subcomponents to reach location l_1.

`E<> forall (p:C) p.`l_1

(a)

`E<> forany (p:C) p.`l_1

(b)

Figure 13.2: Example of extended query language.

13.5 Printability

Principle 5 (Printable) is the principle in our manifesto that has seen the least amount of attention. This principle is concerned with being able to export the generated models without losing information. We do not facilitate anything that helps modelers to distribute their models in medias such as articles without potentially losing information, it is still possible to include screenshots of the models like we did in this report. However, the way the models are designed should still increase the overview by being clearer on where properties belong. We would like to investigate possibilities for generating a tool that would assist users in exporting the model, guiding them in what should probably be included - e.g. providing them with options to include the declarations when exporting the model.

13.6 Refactoring

As previously mentioned, refactoring is a key technique in software development [Mens and Tourwé, 2004], and can be used to make the code (and in our case, model) simpler, or more easy to understand. This section will cover some of the ideas that might be worth implementing in order to save time for developers by doing automated refactoring.

13.6.1 Search

Most refactoring techniques require that the user, or the system, is capable of searching through the developed program, classes, text-files, databases etc. This is no exception in UPPAAL, where you sometimes find yourself looking through the different areas of your model to find specific elements, for instance, a channel, use of variables etc. Based on this, we believe that a simple search functionality would improve the efficiency of modelers.

13.6.2 Safe Delete

During development of models in H-UPPAAL, you might find yourself going through the models, adding new locations or deleting old ones. However, deleting locations might have unseen side-effects. Imagine deleting a class in some object-oriented language. This class might be used in other classes, causing the project to be unable to compile. This error is pretty easy to spot once it has been introduced. Now consider deleting a location in both H-UPPAAL and UPPAAL. Deleting a location will cause all incoming and outgoing edges to also be deleted. These edges might have some channel synchronization on them, e.g. `communicate?`, which might be receiving on a broadcast channel. Because of this, the project would still be able to "compile".

To solve this, modern IDEs, such as *IntelliJ*, introduces the concept of *Safe Delete*. By safe deleting something, you get a list of places that will be influenced by deleting a specific element. It is now up to the developer if he actually wants to delete the element, or if the place influenced should be updated before deleting. We believe that introducing the concept of safe deletion will benefit developers in H-UPPAAL, and potentially eliminate some problems.

13.6.3 Renaming

Similarly to deleting elements, renaming of elements could potentially also lead to some problems. Imagine having generated one of the long queries described in Section 10.3 and then changing the name of one of the subcomponents. This would cause the query to be unable to run. The users might even have introduced his own complex queries. Having to manually maintain these queries can be tedious and error-prone. This is why many IDEs have introduced the concept of renaming elements. In *IntelliJ* for instance, you may use this refactoring technique to rename a class, causing all places where this class is used to be updated.

In H-UPPAAL, we could consider the example with renaming subcomponents, but other areas where renaming is useful is channels. Channels are often spread out in different parts of the model due to the nature of channels, which is creating a media for communication between two, or more, areas in the model. This causes the developers to have to go around, searching for these channels in order to update them. However, if you introduce this technique, all of this work would be trivially fulfilled.

13.6.4 Updating Source or Target Locations on an Edge

Imagine having locations l_0 and l_2 and edge $l_0 \rightarrow l_2$. Let us now consider the activity of adding a new location l_1, which goes in between the two existing locations. In UPPAAL, this can be achieved by dragging the already existing edge onto the newly created l_1, and then creating a new edge $l_1 \rightarrow l_2$. In H-UPPAAL, however, you have to delete the old edge, and then create two new edges, $l_0 \rightarrow l_1$ and $l_1 \rightarrow l_2$. This quickly becomes a tedious, and sometimes very time-consuming task since you might have to re-write several edge properties. During development different models in H-UPPAAL, we find ourselves missing this feature, and could only imagine that other UPPAAL users would do the same. We therefore propose that this feature were to be introduced into H-UPPAAL.

13.6.5 Convert to Subcomponent

It might not always be trivial to see what components you are going to need before starting the modeling process. In the current version of H-UPPAAL, creating a new component will always result in it being completely empty (besides from the initial and final locations of course). However, you might find yourself in a situation where you would like to create a component based on some items already implemented in another component. Based on this, we propose that you could implement a feature, that would allow users to select a specific part of a component and somehow create a new component based on this selection.

13.7 Utilize the UPPAAL Engine

The verification engine in UPPAAL has a lot of neat features for performing verification with optimizations. As of the current version of H-UPPAAL none of these optimizations are propagated into the user interface. It is techniques like under- and over-approximation that could be nice to enable to the user. These techniques can lower the computing power and memory required when verifying queries. It would be nice to see these options available somewhere in the query pane. In the current version of UPPAAL, it is only possible to set these optimizations as a global setting, meaning that you cannot state what optimization to run on each query which could be rather helpful. For instance, an under-approximation with a reachability query will still guarantee that a model adheres to this property where as the same query using over approximation does not ensure anything. For this reason, a query-based optimization setting could be useful to include. This optimization and many others would be nice to see in a future version of H-UPPAAL.

References and Appendices

14 Bibliography

Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas (2001). *Manifesto for Agile Software Development*. URL: http://agilemanifesto.org/.

Behrmann, Gerd, Re David, and Kim G. Larsen (2004). "A tutorial on uppaal". In: Springer, pp. 200–236.

Benyon, David (2010). *Designing Interactive Systems: A Comprehensive Guide to HCI and Interaction Design (2nd Edition)*. Pearson Education Canada. ISBN: 0321435338.

Boudjadar, Jalil, Kim Guldstrand Larsen, Jin Hyun Kim, and Ulrik Nyman (2014). "Compositional Schedulability Analysis of An Avionics System Using UPPAAL". In: *Proceedings of the 1st International Conference on Advanced Aspects of Software Engineering, ICAASE 2014, Constantine, Algeria, November 2-4, 2014*. Vol. 1294. CEUR-WS, pp. 140–147.

Clarke, Edmund M. (2008). "The Birth of Model Checking". In: *25 Years of Model Checking: History, Achievements, Perspectives*. Ed. by Orna Grumberg and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–26. ISBN: 978-3-540-69850-0. DOI: 10.1007/978-3-540-69850-0_1. URL: http://dx.doi.org/10.1007/978-3-540-69850-0_1.

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.

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

Harel, David (1987). "Statecharts: A visual formalism for complex systems". In: *Science of computer programming* 8.3, pp. 231–274.

Mens, Tom and Tom Tourwé (2004). "A survey of software refactoring". In: *IEEE Transactions on software engineering* 30.2, pp. 126–139.

Nielsen, Jakob (2005). *Putting A/B Testing in Its Place*. URL: https://www.nngroup.com/articles/putting-ab-testing-in-its-place/.

— (2012). *Thinking Aloud The #1 Usability Tool*. URL: https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/.

Object Management Group (OMG) (2015). *Unified Modeling Language ™ (UML®) Version 2.5*. OMG Document Number formal/2015-03-01 (http://www.omg.org/spec/UML/2.5/PDF).

Sipser, Michael (2012). *Introduction to the Theory of Computation*. Thomson South-Western. ISBN: 1133187811.

Stallman, Richard (1985-1987). *The GNU Manifesto*. URL: `https://www.gnu.org/gnu/manifesto.en.html`.

Tidwell, Jenifer (2010). *Designing interfaces*. " O'Reilly Media, Inc."

15 Full JSON Example

Code Snippet 15.1 shows a full example of the JSON file for the *Kitchen* component described in Chapter 11. Here, we see how the different locations are described using objects, and how they are connected using edge objects.

```
1  {
2    "name": "Kitchen",
3    "declarations": "",
4    "locations": [
5      {
6        "id": "L25",
7        "nickname": "",
8        "invariant": "",
9        "type": "NORMAL",
10       "urgency": "NORMAL",
11       "x": 300.0,
12       "y": 70.0,
13       "color": "8",
14       "nickname_x": 30.0,
15       "nickname_y": -10.0,
16       "invariant_x": 10.0,
17       "invariant_y": 20.0
18     },
19     {
20       "id": "L34",
21       "nickname": "NewOrders",
22       "invariant": "",
23       "type": "NORMAL",
24       "urgency": "URGENT",
25       "x": 470.0,
26       "y": 70.0,
27       "color": "8",
28       "nickname_x": -30.0,
29       "nickname_y": -40.0,
30       "invariant_x": 30.0,
31       "invariant_y": 10.0
32     },
33     {
34       "id": "L38",
35       "nickname": "",
36       "invariant": "",
37       "type": "NORMAL",
38       "urgency": "COMMITTED",
39       "x": 470.0,
40       "y": 160.0,
41       "color": "8",
42       "nickname_x": 30.0,
43       "nickname_y": -10.0,
44       "invariant_x": 30.0,
45       "invariant_y": 10.0
46     }
47   ],
48   "initial_location": {
49     "id": "L6",
50     "nickname": "",
51     "invariant": "",
52     "type": "INITIAL",
53     "urgency": "COMMITTED",
54     "x": 21.0,
```

```
55      "y": 21.0,
56      "color": "8",
57      "nickname_x": 30.0,
58      "nickname_y": -10.0,
59      "invariant_x": 30.0,
60      "invariant_y": 10.0
61    },
62    "final_location": {
63      "id": "L7",
64      "nickname": "",
65      "invariant": "",
66      "type": "FINAl",
67      "urgency": "NORMAL",
68      "x": 809.0,
69      "y": 439.0,
70      "color": "8",
71      "nickname_x": 30.0,
72      "nickname_y": -10.0,
73      "invariant_x": 30.0,
74      "invariant_y": 10.0
75    },
76    "jorks": [
77      {
78        "x": 270.0,
79        "y": 180.0,
80        "id": "F2",
81        "type": "FORK"
82      },
83      {
84        "x": 470.0,
85        "y": 380.0,
86        "id": "J3",
87        "type": "JOIN"
88      }
89    ],
90    "sub_components": [
91      {
92        "component": "LineCook",
93        "identifier": "James",
94        "x": 540.0,
95        "y": 230.0,
96        "width": 240.0,
97        "height": 120.0
98      },
99      {
100       "component": "LineCook",
101       "identifier": "Adam",
102       "x": 280.0,
103       "y": 230.0,
104       "width": 240.0,
105       "height": 120.0
106     },
107     {
108       "component": "Ramsay",
109       "identifier": "Gordon",
110       "x": 20.0,
111       "y": 230.0,
112       "width": 240.0,
113       "height": 120.0
114     }
115   ],
116   "edges": [
117     {
118       "target_sub_component": "James",
119       "source_jork": "F2",
120       "select": "",
121       "guard": "",
122       "update": "",
123       "sync": "",
124       "nails": [
```

```
125         {
126           "x": 320.0,
127           "y": 210.0,
128           "property_type": "NONE",
129           "property_x": 0.0,
130           "property_y": 0.0
131         },
132         {
133           "x": 560.0,
134           "y": 210.0,
135           "property_type": "NONE",
136           "property_x": 0.0,
137           "property_y": 0.0
138         }
139       ]
140     },
141     {
142       "target_sub_component": "Adam",
143       "source_jork": "F2",
144       "select": "",
145       "guard": "",
146       "update": "",
147       "sync": "",
148       "nails": []
149     },
150     {
151       "source_location": "L25",
152       "target_jork": "F2",
153       "select": "",
154       "guard": "",
155       "update": "",
156       "sync": "cook?",
157       "nails": [
158         {
159           "x": 300.0,
160           "y": 130.0,
161           "property_type": "SYNCHRONIZATION",
162           "property_x": 10.0,
163           "property_y": -10.0
164         }
165       ]
166     },
167     {
168       "source_location": "L34",
169       "target_location": "L38",
170       "select": "",
171       "guard": "",
172       "update": "",
173       "sync": "cook?",
174       "nails": [
175         {
176           "x": 470.0,
177           "y": 110.0,
178           "property_type": "SYNCHRONIZATION",
179           "property_x": 20.0,
180           "property_y": -10.0
181         }
182       ]
183     },
184     {
185       "source_location": "L38",
186       "target_jork": "F2",
187       "select": "",
188       "guard": "",
189       "update": "",
190       "sync": "cooked!",
191       "nails": [
192         {
193           "x": 420.0,
194           "y": 160.0,
```

```
195           "property_type": "SYNCHRONIZATION",
196           "property_x": -30.0,
197           "property_y": 10.0
198         },
199         {
200           "x": 370.0,
201           "y": 160.0,
202           "property_type": "NONE",
203           "property_x": 0.0,
204           "property_y": 0.0
205         }
206       ]
207     },
208     {
209       "source_location": "L34",
210       "target_location": "L25",
211       "select": "",
212       "guard": "",
213       "update": "",
214       "sync": "cooked!",
215       "nails": [
216         {
217           "x": 400.0,
218           "y": 70.0,
219           "property_type": "SYNCHRONIZATION",
220           "property_x": -30.0,
221           "property_y": 10.0
222         }
223       ]
224     },
225     {
226       "source_location": "L6",
227       "target_location": "L25",
228       "select": "",
229       "guard": "",
230       "update": "",
231       "sync": "",
232       "nails": [
233         {
234           "x": 20.0,
235           "y": 70.0,
236           "property_type": "NONE",
237           "property_x": 0.0,
238           "property_y": 0.0
239         }
240       ]
241     },
242     {
243       "target_location": "L34",
244       "source_jork": "J3",
245       "select": "",
246       "guard": "",
247       "update": "",
248       "sync": "",
249       "nails": [
250         {
251           "x": 500.0,
252           "y": 410.0,
253           "property_type": "NONE",
254           "property_x": 0.0,
255           "property_y": 0.0
256         },
257         {
258           "x": 800.0,
259           "y": 410.0,
260           "property_type": "NONE",
261           "property_x": 0.0,
262           "property_y": 0.0
263         },
264         {
```

```
265           "x": 800.0,
266           "y": 70.0,
267           "property_type": "NONE",
268           "property_x": 0.0,
269           "property_y": 0.0
270         }
271       ]
272     },
273     {
274       "source_sub_component": "James",
275       "target_jork": "J3",
276       "select": "",
277       "guard": "",
278       "update": "",
279       "sync": "",
280       "nails": [
281         {
282           "x": 760.0,
283           "y": 370.0,
284           "property_type": "NONE",
285           "property_x": 0.0,
286           "property_y": 0.0
287         },
288         {
289           "x": 520.0,
290           "y": 370.0,
291           "property_type": "NONE",
292           "property_x": 0.0,
293           "property_y": 0.0
294         }
295       ]
296     },
297     {
298       "source_sub_component": "Adam",
299       "target_jork": "J3",
300       "select": "",
301       "guard": "",
302       "update": "",
303       "sync": "",
304       "nails": []
305     },
306     {
307       "target_sub_component": "Gordon",
308       "source_jork": "F2",
309       "select": "",
310       "guard": "",
311       "update": "",
312       "sync": "",
313       "nails": [
314         {
315           "x": 280.0,
316           "y": 210.0,
317           "property_type": "NONE",
318           "property_x": 0.0,
319           "property_y": 0.0
320         },
321         {
322           "x": 40.0,
323           "y": 210.0,
324           "property_type": "NONE",
325           "property_x": 0.0,
326           "property_y": 0.0
327         }
328       ]
329     },
330     {
331       "source_sub_component": "Gordon",
332       "target_jork": "J3",
333       "select": "",
334       "guard": "",
```

```
335         "update": "",
336         "sync": "",
337         "nails": [
338           {
339             "x": 240.0,
340             "y": 370.0,
341             "property_type": "NONE",
342             "property_x": 0.0,
343             "property_y": 0.0
344           },
345           {
346             "x": 470.0,
347             "y": 370.0,
348             "property_type": "NONE",
349             "property_x": 0.0,
350             "property_y": 0.0
351           }
352         ]
353       }
354     ],
355     "main": false,
356     "description": "Cooks Ramsays famous dishes",
357     "x": -5.0,
358     "y": -5.0,
359     "width": 830.0,
360     "height": 460.0,
361     "color": "8"
362   }
```


Listing 15.1: Full JSON file for the component *Kitchen* described in Chapter 11.