CISS - University of Aalborg

Internship

Uppaal for Parametric Models

Author:
Romain Sertelon

Supervisor:
Alexandre David

November 2, 2010

Acknowledgements

First of all, I would like to thank my supervisor at Aalborg University, Alexandre David for giving me the opportunity to do this internship, for teaching me about model checking and for helping me during my project.

I would like to thank Anders Gregersen for his help with the cluster.

I would also like to thank Kim Larsen and the entire CISS team for their kind welcome.

I would also like to thank Aurélie Waesberghe, for helping me to have this internship.

Finally, I would like to thank my mother for having given me the idea of an internship abroad, Thomas Lorrain, my friend, without whom I would not have done this internship and my girlfriend for her patience and understanding during this internship.

Contents

Acknowledgements **ii**

1 Introduction **1**
1.1 CISS . . . 1
1.2 My mission . . . 2

2 Problem description **3**
2.1 Overview . . . 3
2.2 Motivation . . . 5
2.2.1 Parametric Model Checking . . . 5
2.2.2 Related Work . . . 5
2.2.3 Problem . . . 6
2.2.4 Requirements . . . 8

3 Issues **10**
3.1 Parametric Search Algorithms . . . 10
3.1.1 Problem . . . 10
3.1.2 Exploration Algorithms . . . 10
3.2 Web Interface Framework . . . 15
3.2.1 Using a Framework . . . 15
3.2.2 A Framework for Uppaal PM . . . 16
3.3 Verification Feedback . . . 17
3.3.1 Text view . . . 17
3.3.2 Graphical view . . . 18

4 Design of the tool **19**
4.1 Architecture of Uppaal PM . . . 19
4.1.1 Inputs and Outputs . . . 19
4.1.2 Tasks . . . 20
4.1.3 Components . . . 20
4.2 Database . . . 21
4.2.1 Role . . . 21
4.2.2 Data Model of the application . . . 21
4.2.3 Entities . . . 22

4.3 Web Interface . . . 23
4.3.1 Component tasks and auxiliary features . . . 23
4.3.2 Site Navigation . . . 25
4.3.3 Look of the interface . . . 25
4.4 Engine . . . 26
4.4.1 Component tasks . . . 26
4.4.2 Architecture . . . 28
4.5 Feedback . . . 30
4.5.1 Textual representation . . . 30
4.5.2 Graphical representation . . . 31

5 Implementation **33**
5.1 *Ruby on Rails* Application . . . 33
5.1.1 Presentation . . . 33
5.1.2 Creating UPPAAL PM . . . 33
5.2 Database Component . . . 34
5.2.1 Creation of the model . . . 34
5.2.2 Models associations . . . 34
5.2.3 The Job Model . . . 35
5.3 Web Interface . . . 37
5.3.1 Controllers . . . 37
5.3.2 Nested Forms . . . 38
5.3.3 Views . . . 39
5.3.4 Gems . . . 39
5.4 Verification Engine . . . 39
5.4.1 Overview . . . 39
5.4.2 Modularization . . . 41
5.4.3 Components . . . 43

Conclusion and Future Work **47**

A Cluster **50**
A.1 Presentation . . . 50
A.2 ARC Interface . . . 51
A.2.1 XRSL Files . . . 51
A.2.2 ngsub . . . 51
A.2.3 ngstat . . . 51
A.2.4 ngget . . . 52

B Development architecture **53**

Bibliography **55**

Chapter 1

Introduction

1.1 CISS

CISS[1] is a Danish company based in the Computer Science Department of Aalborg University, in Danmark.

CISS activity field is research in embedded systems. It has been created around the year 2000 to accelerate the transfer of knowledge from research to industry in Danmark, to improve the competitiveness of Danish companies.

There are about thirty five employees working at CISS; among them, there are PhD students, researchers and professors as the company works with Aalborg University.

[1]CISS stands for *Center for Indlejrede Software Systemer* or *Center for Embedded Software Systems* in English

1.2 My mission

I integrated CISS on the first of April as an intern for six month. I was supervised during my internship my Mr Alexandre David, associate professor at CISS and responsible for the development of Uppaal , a model checker for timed automata models.

When I arrived, I was given a project that I had to conduct alone for the six months of my internship. This project was the creation of a new software for parametric model checking to extend the possibilities of Uppaal Tools Suite.

The following report presents my work during this internship. Chapter 2 presents the problem raised by my project, chapter 3 discusses different issues of this problem. Chapter 4 presents the design of Uppaal PM and chapter 5 its implementation.

Chapter 2

Problem description

2.1 Overview

Uppaal UPPAAL [1] is a tool for verification of real-time system jointly developed by Aalborg University in Denmark and Uppsala University in Sweden. It is designed to check systems that can be modeled as a network of timed automata (TA) extended with integer variables, structured data types, channel synchronization and other properties.

As shown on figure 2.1, UPPAAL is mainly composed of two parts: the client and the server. The server is written in C++; it manages the computations for the model checker and the simulator, and it can be executed on a (more powerful) remote system. The client is written in Java; it manages only user interaction, offering an editor to create models, a simulator to simulate the behavior of the tool and a property verifier.

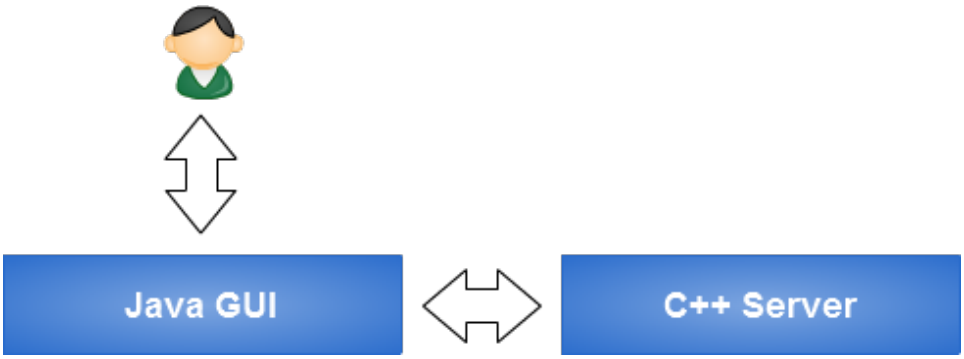

Figure 2.1: UPPAAL architecture

To process to the verification of a model, UPPAAL 's server will ask for the model and the properties to be verified. Once the verification is done, it will show the results of the properties. Moreover, it can give the trace of the verification as shown on figure 2.2, so you can use it in its simulator

afterwards.

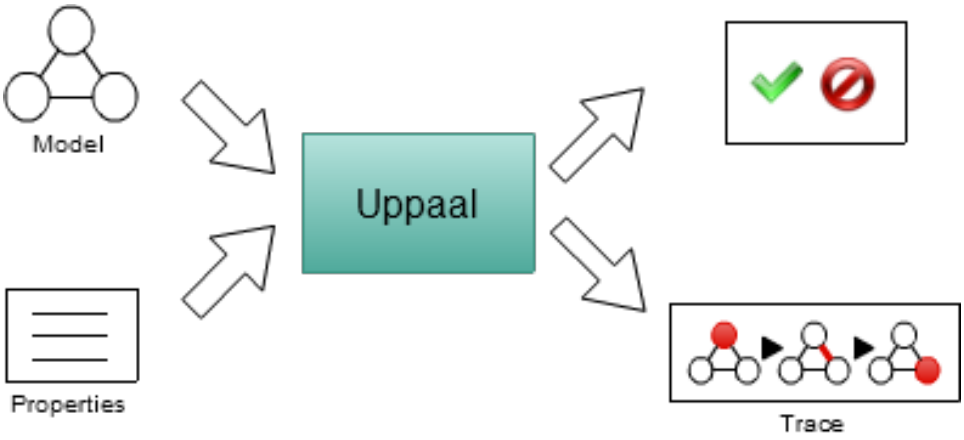

Figure 2.2: UPPAAL 's inputs and outputs

Uppaal tools suite Several tools are born from UPPAAL . UPPAAL CORA [2] is a branch of UPPAAL for Cost Optimal Reachability Analysis, it uses an extension of timed automata to annotate the model with the notion of cost; it is used for planning and scheduling. UPPAAL for Testing Real-time systems ONline (TRON) [3] is a testing tool based on UPPAAL engine suited for black-box conformance testing of timed systems. UPPAAL for TImed GAmes (TIGA) [4] is an extension of UPPAAL and it implements the first efficient on-the-fly algorithm for solving games based on timed game automata with respect to reachability and safety properties.

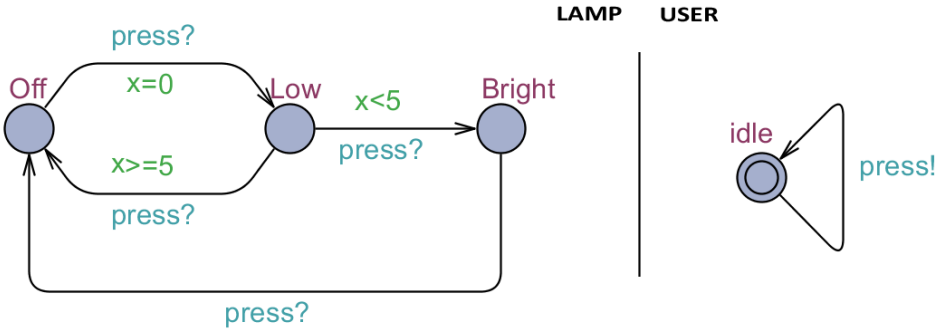

Figure 2.3: Timed Automata representing a lamp

Timed Automata Models Networks of Timed Automata models are used to describe and verify real-time systems, figure 2.3 shows an example

of this kind of model.

These models are composed of locations, that represent states of the system, in this example, the lamp has three states: off, low light and bright light and the user stays idle. The edges are here to define the transitions between the states; these transitions indicates the possible changes of states of the system.

On this figure, x is a clock, it represents the time passing. On the transitions, there are guards that add clock constraints on the transition which can be crossed only if these constraints are satisfied. To allow interactions between automata, we can add channels; on the figure, we can see one channel named *press*. When a transition is crossed with a channel name with a *!*, an order is given to the channels with an *?* which activates the corresponding transition.

2.2 Motivation

2.2.1 Parametric Model Checking

Parameters are often present inside models, they can add constraints on them, or, for example, they can parameterize clock constraints. One may want to know what happens to certain properties if the value of one parameter changes. Will the property stay verified or not ?

Parametric model checking allows one to answer this question, it can find whether a property is verified or not and for which values of some parameters. Another purpose of parametric model checking would be to find the maximum (or minimum) of a variable inside the model.

2.2.2 Related Work

Parametric model checking can be a powerful tool for real-time systems conception, therefore, other model checkers have implemented it. They all try to automatize the verification of parametric models.

HyTech The HYbrid TECHnology Tool [5] is an automatic tool for the analysis of embedded systems. Hybrid systems are specified as collections of automata with discrete and continuous components, and temporal requirements are verified by symbolic model checking. HyTech aids in the design of embedded systems by not only checking systems requirements, but also performing parametric analysis. Given a parametric system description, HyTech returns the exact conditions on the parameters for which the system satisfies its safety and timing requirements exploring state-space either by partition refinement or forward reachability.

IMITATOR II IMITATOR II [6] is a tool used with parametric models to find the values of the parameters that will verify the properties specified by the user. It uses the "inverse method" [7] described like this by the author: *given a reference valuation of the parameters, it generates a constraint such that the system behaves the same as under the reference valuation in terms of traces.* In other words, the "inverse method" starts from a point of the parametric space known to verify a property, and it tries to find the maximum set of points around this one that will verify this property. This method can be used by Uppaal PM to optimize the search of the solutions.

Another interesting feature of IMITATOR II is its "behavioral cartography algorithm" [8]. It is a step-wise refinement method where, at each step, parameters are constrained.

Extension of Uppaal Thomas Hune found an extension of timed automata models [9] that can represent parametric models and allow its verification with a plug-in of Uppaal . Unlike Uppaal PM, this extension uses symbolic notation of the parameters that are limited to clock constraints. It can verify a parametric model only if the parametric model belongs to a specific class of parametric timed automata; and it cannot find maxima or minima.

2.2.3 Problem

Current limitations Current implementations of parametric model checking are all based upon symbolic representation of parameters inside the model. This representation has some limitations. For instance, parameters can be defined only for certain elements of the models, usually clock constraints only. Moreover, these implementations are qualitative as they focus on the verification of properties, they cannot find quantitative information such as the maximum of a value.

There are also some solutions that could exist but could not be seen by such implementations, in the case where solutions can exist in two separated zones of the state space as shown on figure 2.4; indeed, some methods, such as the inverse method, find a set of solutions that are connected, thus missing other solutions.

Uppaal PM The aim of Uppaal PM is to go beyond these limitations and bring new features to parametric model checking. We will achieve this with a new approach: instead of using symbolic parameters, we will declare the parameters and "instantiate" all the models that are inside the parameter space, as shown on figure 2.5.

With our approach, we can:

- Choose parameters anywhere inside the model;

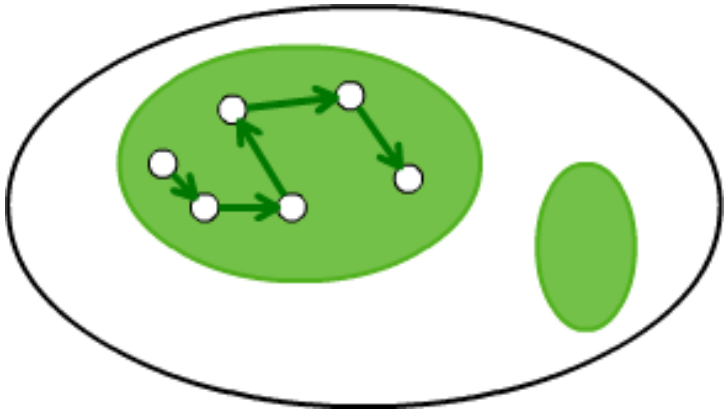

Figure 2.4: Symbolic parametric exploration in the left zone may not reach the right zone

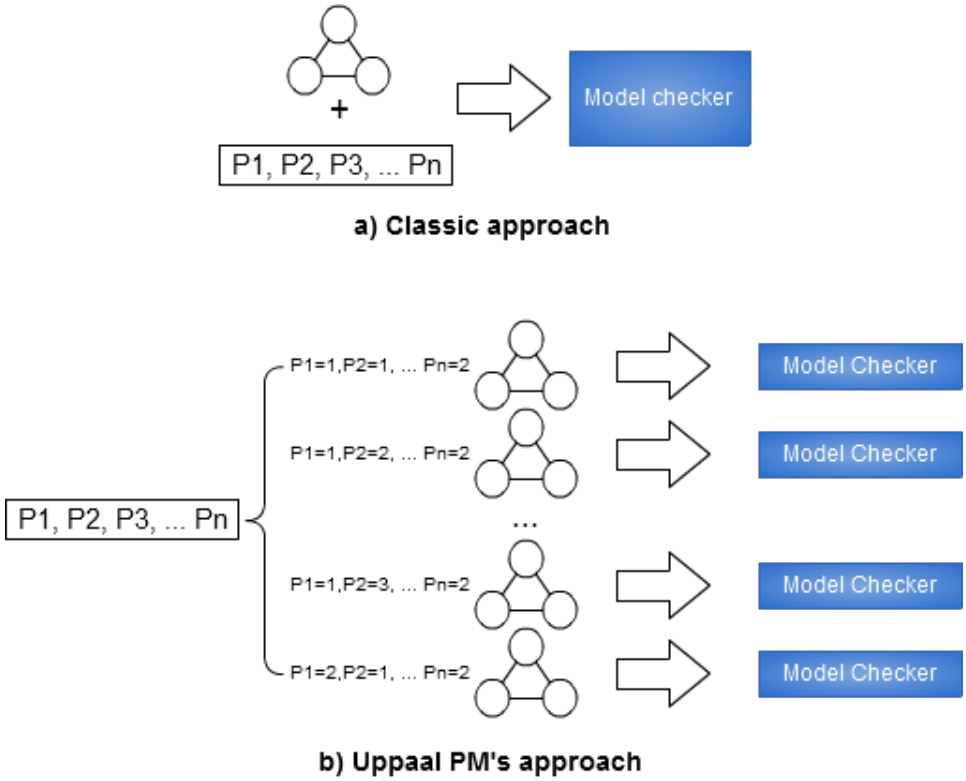

Figure 2.5: Difference between UPPAAL PM's approach and other parametric model checkers

- Find all the values of a set of parameters for which a property is verified;
- Find the values of a set of parameters for which a variable inside the model is maximized (or minimized);
- Show a graphical map of these values.

2.2.4 Requirements

Computing verifications As explained before, UPPAAL PM "instantiates" all the models that are inside the parameter space, therefore, there are lots of models generated by the tool. All these models will be verified by a model checker (for instance UPPAAL , but as we will see later, it can be another one) which implies that the tool needs a lot of computing time. To be able to launch several verifications at the same time, we need a computing architecture that allows parallel computing.

A cluster offers a capability of hundreds of even thousands of simultaneous verifications. The department of computer science of Aalborg University hosts a cluster that we can use to compute all the verifications; UPPAAL PM will be designed to launch verifications on this cluster.

Parametric search optimization With its "instantiation" approach, UPPAAL PM offers great features, but, even if we use a cluster to compute the intermediate verifications, the number of them can still be very huge. To avoid extremely long computing time, UPPAAL PM will use parametric search algorithms, as they can decrease the number of models to be checked by the tool. They will be discussed in the next chapter.

User Interface The user interface of the tool will help users to verify their parametric models and see the results on a graph as stated before. To achieve this, there are two kinds of client applications : Heavy clients and a web interface.

As we use a cluster, we need to access it from a particular server. Therefore, the engine of the tool will have to send commands from this server implying that the tool will have a server component.

Thus, using a heavy client is not a good solution, it adds a code layer for communication with the server, it has to be deployed, and it would still need a web interface for back end administration.

The easiest solution and the most practical one is a web interface as everything will be centralized in one place, it will be easier to upgrade, to maintain and for the users, it will be accessible from anywhere as long as they can access a web browser.

Modularization UPPAAL PM will be designed to be modular; it will be able to realize a verification without any knowledge of the models or the methodology used to verify it.

This abstraction allows easy switch of the tool used for verification, in this report, UPPAAL PM will use UPPAAL model checker but it could use another branch of UPPAAL , or even another model checker.

Moreover, we can apply this modularization to the search algorithms, to add them easily in the tool and to allow one to choose the algorithm to be used. One could then choose the best algorithm for its problem.

Finally, modularization can be applied on the cluster interface, so that UPPAAL PM can be installed with any kind of cluster.

Chapter 3

Issues

3.1 Parametric Search Algorithms

3.1.1 Problem

Combinatorial Explosion To verify a parametric model, UPPAAL PM will generate one model per point in the parameter space to verify it and then gather the results. This method will give us an exhaustive map of the results for us to see how the model responds to parameters' values.

Although it is really useful, this method generates a number of models exponential in number of parameters, and so is the verification time. This is a problem as we don't want to monopolize the cluster or the tool for one verification only, and, of course, it is annoying for the end-user who will have to wait for days before his verification is finished.

Optimizing the Verification In order to optimize the computation of the verification, a cluster is being used to compute intermediate verifications in parallel, thus reducing the global computing time used. But this will not solve the problem raised before. To solve this problem, we have to find a way to decrease the number of intermediate verifications; this can be done using various algorithms and heuristics.

3.1.2 Exploration Algorithms

There are two kinds of problem that we want to solve using parametric model checking. The first one is to know whether a property is or is not verified for all the values of the parameters; the second one is to find the maximum or the minimum of a variable inside a model. For each of these problems, there are algorithms or heuristics that can help finding a solution with less computation.

Brute Force Algorithm

The Brute Force Algorithm is the simplest one, and the most intuitive. It consists in an exhaustive exploration of the parameter space, all the values are explored one by one until there is no other value to be checked.

This algorithm can be applied to both problems and gives the most accurate results. However, its complexity is in $O(r^n)$ where r is the maximum range of the parameters and n the number of parameters, which makes it not suitable for a large number of parameters.

Hill Climbing

Hill climbing attempts to maximize (or minimize) the value of a variable inside the parametric model which is a function of the parameters. This algorithm improves the brute force search a lot when it comes to find the maximum or the minimum of a variable of the parametric model.

Explanations This algorithm is based on the idea that, to go to the top of a mountain, one always walk in direction of the ascent. Hill Climbing starts from one point of the parameter space and calculates which neighbor has the greatest (or lowest) value and goes to this point; it continues like this until it cannot go up (or down) anymore, as shown on figure 3.1.

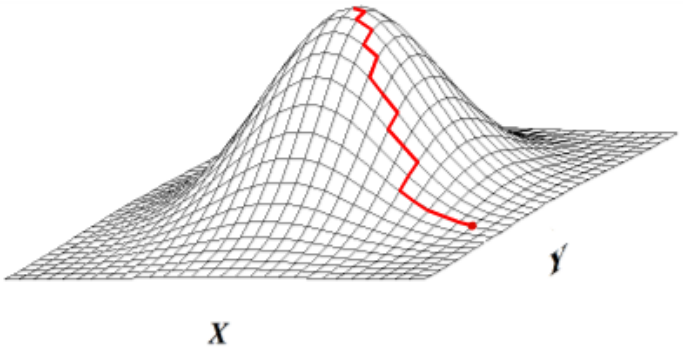

Figure 3.1: Hill Climbing algorithm

Limitations This method have some limitations that can compromise the accuracy of the result. Indeed, Hill Climbing can be stuck in a local maximum (or minimum) as shown on figure 3.2 where the red path leads to a local maximum and the termination of the algorithm although it is not the maximum expected. This problem can be avoided by starting the algorithm from different points hoping that if one of the paths leads to a local maximum (or minimum) the others will lead to the global maximum (or minimum).

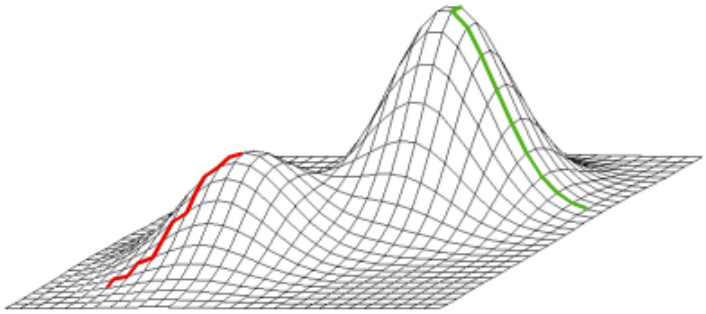

Figure 3.2: Hill Climbing stuck in a local maximum

This method is heuristic, therefore, it is hard to foresee its behavior but still we can say that in the worst case, it behaves like the brute force algorithm.

Genetic Algorithms

Presentation Genetic algorithms (GA) [10] belong to the evolutionary algorithms which mimic natural evolution. They are inspired by the evolution of species at a genetic point of view.

In these algorithms, we define the solutions as individuals; these individuals are composed of chromosomes which are represented by the variables of the solution. The concepts of reproduction and evolution will be applied on these individuals to find the best solution. Before explaining the algorithm itself, we present some definitions of terms that are originally used in biology.

Chromosome Chromosomes are what compose an individual DNA. In GA, they are parts of the solutions. For example, in our problem, a chromosome is a parameter value as shown on figure 3.3

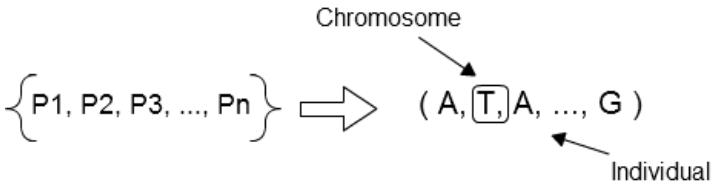

Figure 3.3: Chromosomes in Genetic Algorithms

Crossover The Crossover is an operation that consists in mixing two parents' chromosomes to get two children as shown on figure 3.4

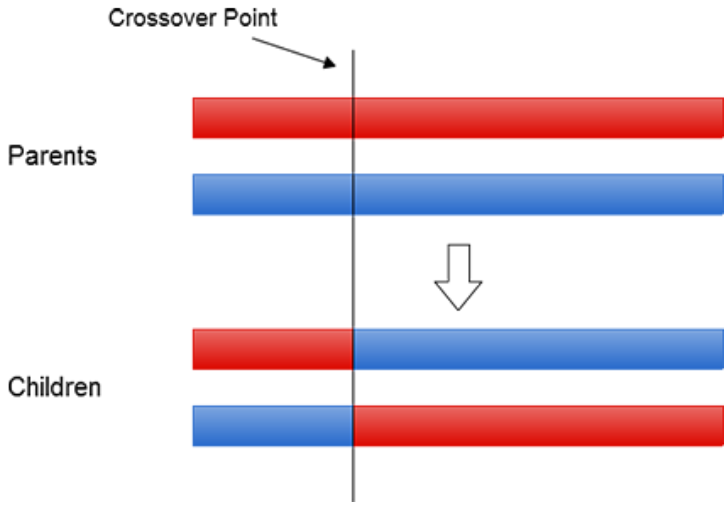

Figure 3.4: Crossover mix two parents' chromosomes to create two children

Mutation The mutation of a chromosome consists of a slight change of its value. It happens randomly and rarely.

Fitness Function Evolution is often referred as by the survival of the fittest, this function will define a criteria that allows to select the fittest individuals of the population.

Algorithm A genetic algorithm is divided into four steps: the initialization, the selection, the reproduction and the termination. These steps are represented on figure 3.5 and detailed afterwards.

Figure 3.5: Genetic Algorithm

1. **Initialization** On the first step, we create several individuals ran-

domly, this will be the first generation of individuals from whom children will be born.

2. **Selection** Then, we use the fitness function to select the best individuals of the population.
3. **Reproduction** A small part of individuals from the resulting population are chosen randomly to create the next generation by crossover. This is done until there is a population of satisfying size.
4. **Termination** The last two steps are repeated until termination. Termination is reached when:
   - A solution is found that satisfies a minimum criteria.
   - The algorithm has looped N times.

Limitations Genetic algorithms usually give good results, but sometimes, they tend to converge too quickly, thus giving a solution that is not the best, and could be considered as a local maximum.

To avoid this problem, the fitness function can be changed to keep individuals even if they are not the fittest ones, mutation can be used or, if already used, the rate of mutation can be increased. These techniques will decrease the chances of being stuck in a local maximum (or minimum).

Simulated Annealing

Presentation Simulated annealing [11] is another algorithm that mimics nature, but this time it is based upon physics. Annealing is a technique used in metallurgy industry to get better materials by making the temperature of a liquid metal decrease slowly. Indeed, this tends to minimize the internal energy of the metal, giving more uniform crystals which make the material more solid. As metals have this property of being more solid when cooled slowly because it minimizes the internal energy, we can find a reasonably good solution minimizing some parameter inside our parameters space.

Advantages and Drawbacks

+ Generally effective. This algorithm, after fine tuning, gives good results in few time (compared to the Brute Force search).
+ This algorithm often gives a reasonably good solution, as before, a very good solution, but with no guarantee that it is actually the best one.
- To give good results, it is needed to tune it very precisely, finding the functions is sometimes very difficult.
- Hard to implement.

3.2 Web Interface Framework

3.2.1 Using a Framework

When developing an application, there are several problems that are raised all along the development of the project. A framework can help the developer to solve these problems and focus only on the important parts of the coding : the features of the application.

Model View Controller Model View Controller (MVC) [12] is a widely recognized design pattern among web developers that separates out database and business logic from the presentation layer; this pattern is shown on figure 3.6.

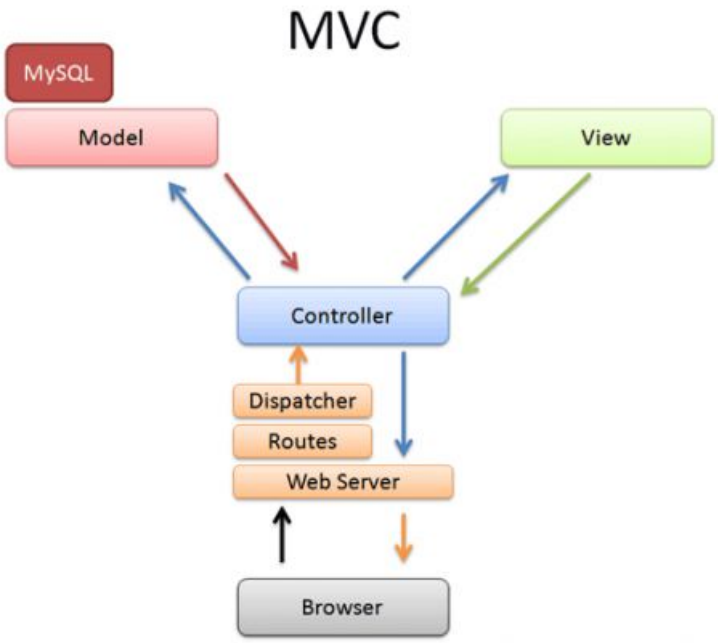

Figure 3.6: Model View Controller Design Pattern

Benefits of a MVC Framework First of all, separating the functionalities of the application in layers is a good way to improve the code and the debugging of the code. If there's an error while inserting an object into the database, it is easy to know where to look.

Frameworks enforce good coding standards that help to code faster, and to share the code easily as every developer familiar with the framework will quickly catch what is the purpose of the written code.

The maintenance of the code is easier while using a framework as parts of the application can be updated without breaking it and not knowing where the bug comes from.

Adding other development principles to a MVC framework can improve it a lot, for example, with the Don't Reapeat Yourself (DRY) principle, less code is written, thus leading to a cleaner code and even more understandable.

3.2.2 A Framework for Uppaal PM

Which MVC framework will be used to develop Uppaal PM? That choice is tricky, there are lots of frameworks, and it is hard to know which one is better than the others. However, we will present here several frameworks that are well-known by the web developer community and proven to be efficient and stable.

Overview of web frameworks Frameworks are created for one computer language; here, we will focus on scripting languages, as they are often already installed on web servers and generally more suited for a web application. PHP and Ruby are also two languages that are standards when it comes to the development of web applications.

For these languages, there is a lot of frameworks, specially for PHP as its community is very important, we can mention *Zend Framework*, *Cake PHP*, *CodeIgniter*, *Prado* and *Symfony* [13] for PHP and *Ramaze*, *Wuby* and *Ruby on Rails* [14] for Ruby.

Choosing the framework Among the mentioned frameworks, *Ruby on Rails* seems to be the most popular for Ruby and offers many interesting features such as code generation, code templates, template for the presentation layer, database abstraction, command line tool, test procedures and a possibility to add more features with plug-ins; *Symfony* is the equivalent for PHP language, it has approximatively the same features.

Even if these two frameworks look really similar, they have some differences that will allow us to make a choice, some of them are shown on figure 3.7. Moreover, there is the language, Ruby is a very intuitive full object oriented language whereas PHP is an intuitive procedural language that is slowly integrating object oriented paradigm; this difference makes Ruby more robust for an object oriented application such as Uppaal PM.

Downwards compatibility. There is one important point with frameworks: whether an application built with them is broken or not if the framework is updated to a newer version. *Ruby on Rails* tries to have all its versions downwards compatible, so does *Symfony* but as the framework is young, between two minor versions it happened that some basic functionalities were changed. At the time of the writing of this report, *Symfony* is

	Symfony	Ruby on Rails
Ajax	Prototype / Script.aculo.us	Prototype / jQuery Script.aculo.us
Model View Controller	✓	✓
Internationalization / Localization	✓	✓
Object Relational Mapping	Doctrine / Propel	Active Record
Testing Framework	✓	Unit, Functional and Integration Testing
Database Migrations	⊘	✓
Template Framework	✓	✓
Caching Framework	✓	✓
Form Validation	✓	✓

Figure 3.7: Symfony compared to Ruby on Rails (ref Wikipedia)

going to brand new 2.0 version with major changes whereas *Ruby on Rails* is going to 3.0 version with new features and some changes as well, but less.

For these principal reasons, *Ruby on Rails* will be used to build UPPAAL PM.

3.3 Verification Feedback

UPPAAL PM has to present the results to the user. As there will be a lot of results, we must find what is the best way to show these results.

3.3.1 Text view

A basic approach to show the results of the verification is to print a list of all intermediate results to the user on the web interface. However, this list can be very long, making the analysis of the results difficult.

To solve this problem, the interface can provide filters applied on the results' values, the values of the parameters and so on to help the user to see what he is interested in. Another solution is to compute a summary of the intermediate results and show it to the user so he can have a quick

overview of the results.

These solutions are useful, and can even be used for logging or debugging purpose, but they are not practical, it will still be hard to have an overview of the problem and its results with text results.

3.3.2 Graphical view

Graph of the results A most sophisticated approach to display the results is to show a graph of the results on the web interface. Using this method, the tool can provide an easy way to interpret the results of the verification.

To represent the highest number of parameters while offering an easy-to-understand graph, we may limit the graph to three dimensions. This view will allow the user to see the variation of the results for two parameters at once. If there are more than two parameters, the tool will ask which parameters to show on the axes of the graph and the values of the other parameters.

	Java Applet	Flash Application	Silverlight Application	SVG + Javascript	Pictures
Browser Native	✗	✗	✗	✓	✓
Cossplateform	✓	✓	✗	✓	✓
Open Source	✓	✗	✗	✓	✓
Smooth Animations	✓	✓	✓	Depends on the client hardware	✗
Easiness of Implementation	✓	✓	✓	✗	✗

Figure 3.8: Comparison of different technologies that can display 3D graphs

Displaying the graph To display a three-dimensional graph inside a web page, there are several ways. Indeed, we can use a *Java applet*, a *Flash Application*, a *Silverlight Application*, a combination of SVG and Javascript or generate *pictures* of the graph to insert them in the page. Figure 3.8 shows the differences between these technologies.

Jzy3D Library The best technology to display a three-dimensional graph according to the criteria of figure 3.8 is the Java Applet. Displaying 3D objects with *Java* requires the use of a library: JOGL; Jzy3D [15] is a library[1] specialized in the display of three-dimensional graphics built upon JOGL. This library will be used to display the results of the verification done by UPPAAL PM.

[1]There is another library named JMathTools, but it seems less stable.

Chapter 4

Design of the tool

4.1 Architecture of Uppaal PM

4.1.1 Inputs and Outputs

Uppaal PM will have inputs and outputs, they are shown on figure 4.1.

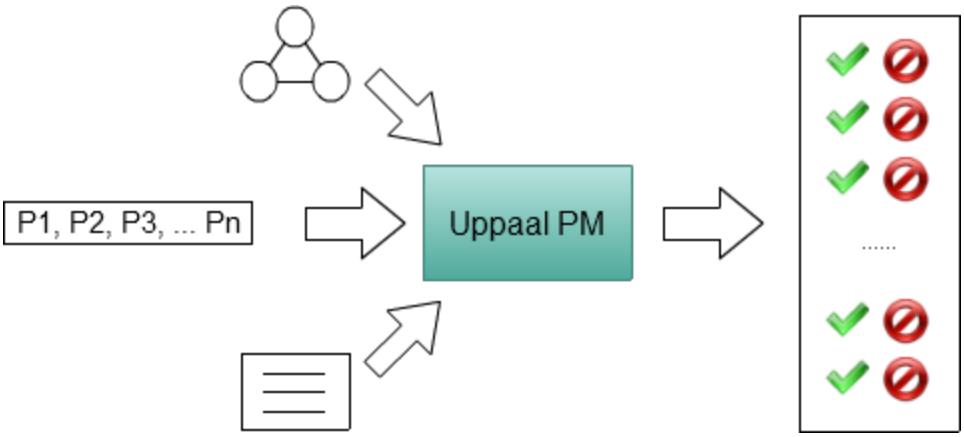

Figure 4.1: The inputs and outputs of Uppaal PM

The inputs consist of a model, some parameters and properties to be verified; on the other hand, the outputs will be the results of the verification.

The Model It represents the timed automata model to be checked in a .xml Uppaal format.

Parameters They have a name which can be anything allowed by Uppaal syntax, a minimum and a maximum value.

Properties They are standard Uppaal properties in a .q file.

Results They can be either a text list of results, or a three-dimensional graph as discussed in section 4.5 on page 30.

4.1.2 Tasks

To go from the inputs to the outputs, there are different operations that have to be accomplished. Figure 4.2 shows how the main task *Verify Parametric Model* can be split into more specific tasks.

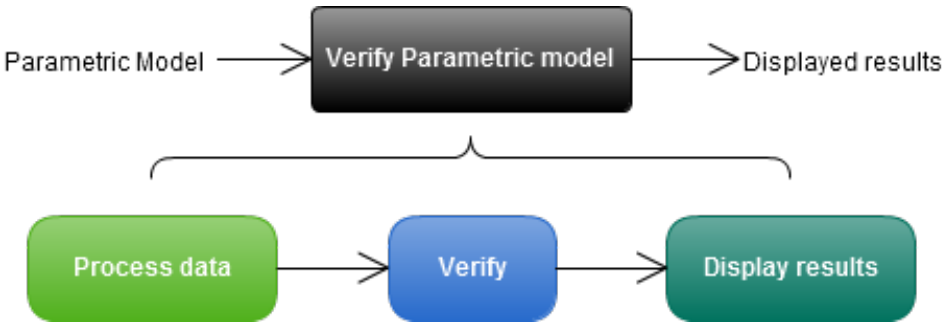

Figure 4.2: Decomposition of the main task of UPPAAL PM

Process Data This task consists in preparing the data for the verification of the model.

Verify This is where the models are "instantiated"[1] and verified with the cluster.

Display Results Here, the results of the verification are shown to the user.

4.1.3 Components

To facilitate the development, the application is divided into three components, each of them will have the responsibility to perform one task. Figure 4.3 represents these components and the links between them.

Web Interface This component is in charge of the *Process Data* task.

Feedback (Java Applet) This one is a part of the web interface specialized in the display of the results, thus it handles the *Display Results* task.

Verification Engine This component is responsible for the *Verify* task.

Database The Database is at the center of all three components, it allows the exchange of data.

[1] See 2.2.3 for an explanation of the instantiation of UPPAAL PM

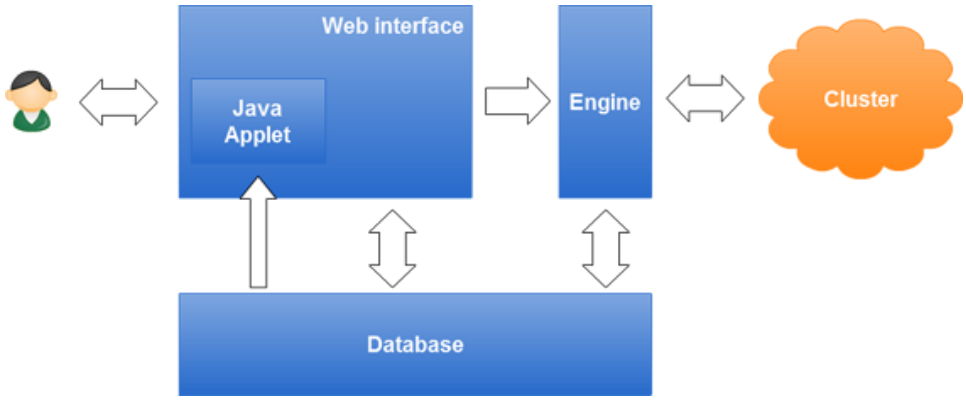

Figure 4.3: Architecture of UPPAAL PM

Data flow All the data that a user inputs into UPPAAL PM will follow the same path. This path is very similar from the task diagram shown on figure 4.2. Indeed, the user sends the model, the parameters and the properties through the web interface that will store them into the database. Then, the web interface commands the verification engine to verify this model. The engine sends the verifications to the cluster and get the results back to store them into the database. Finally, the Java Applet reads the results from the database and shows them to the user.

4.2 Database

4.2.1 Role

Each component needs to get or store information to realize its tasks; the web interface needs to store user information, the engine will have to get the properties of the job it has to check and so on.

The role of the database is to act as a central storage from where every component is able to get data or store it. This is the core of the application in terms of data flow, all the data request comes from the database or goes into it.

4.2.2 Data Model of the application

As the database is the center of the application, because of its role and because of the MVC pattern, the data model has to be created first and every aspects of the application data has to be represented. Figure 4.4 represents the Entity Relationship Model of the database.

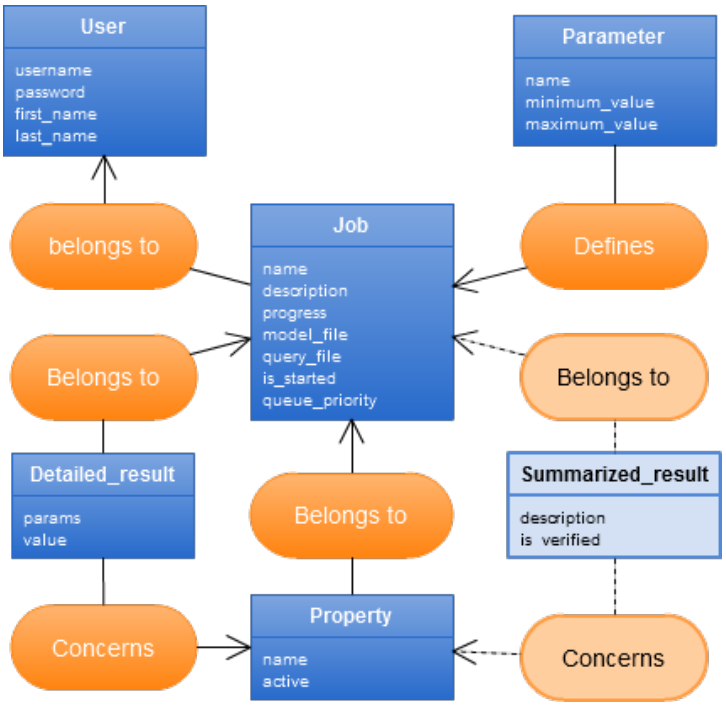

Figure 4.4: Data Model of UPPAAL PM

4.2.3 Entities

The entities shown on figure 4.4 are all characterized differently, they are described next.

User

The *User* entity represents a user of the application. It is characterized by a user name and a password both used to log in the application, and a full name (decomposed into first and last name).

Job

The *Job* entity represents a parametric model to be checked; as it can be viewed or edited by one user only, a job belongs to a user. It is characterized by:

- A name, so it can be easily identified by the user;
- A description;
- A progress value, between 0 and 100;

- The fact that it has been started or not, to know whether it can be edited or not;
- A priority in the queue (not implemented yet)
- A model file, which corresponds to the .xml file representing the model to be checked;
- A property file, which corresponds to the .q file listing the properties to be verified.

Parameter

The *Parameter* entity represents a parameter of a model to be checked, therefore, it belongs to a job. It is characterized by a name, a minimum value and a maximum value.

Property

The *Property* entity represents a property of a model to be checked, as the *Parameter* it belongs to a job. It is characterized by a name and its state: either active or not, as a property extracted from the properties file can be stored but not used for verification.

Detailed Result

The *Detailed Result* entity represents the result of an intermediate verification, it is linked to a job and a property. it is characterized by the parameters values of this verification and the value of the result: either 0 (false) or 1 (true) or any other numeric value.

Summarized Result

This entity is shown for information, it may not be used or even deleted in the future. It should represent a summarized result, that is to say a compilation of detailed results to show an easier-to-read feedback to the user.

4.3 Web Interface

4.3.1 Component tasks and auxiliary features

Component tasks

As we saw earlier, each component will perform one specific task. In the case of the web interface, this task can be split into more specific tasks as shown on figure 4.5.

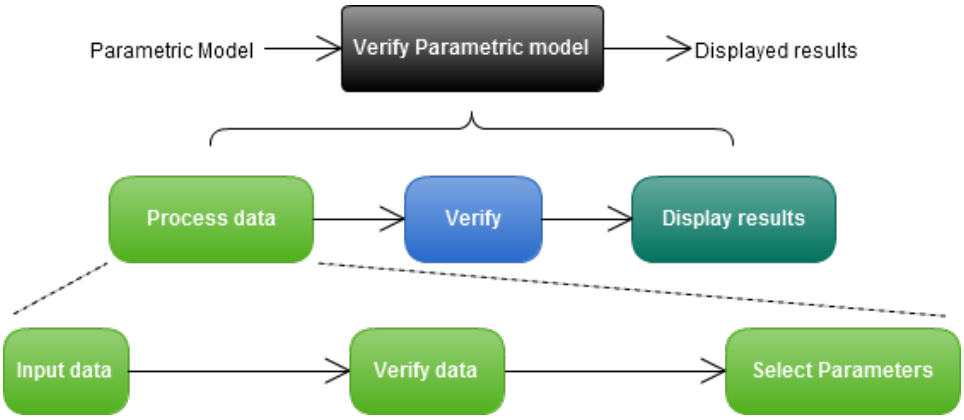

Figure 4.5: The tasks of the web interface

1. **Input Data.** Model file (.xml) and properties file (.q) are asked to the user and imported into the tool during the creation of a job.
2. **Verify Data.** The DTD (Document Type Definition) is used to check that the XML format of the model is well-formed. If not, the model file is rejected and the user has to provide a another one.
3. **Select Parameters.** Once the verification passes, the web interface extracts the parameters from the model file, and the properties from the properties file. The user chooses the parameters and the properties that will actually be used for the verification, and the boundary values of these parameters.

The *Display Results* is performed by a component that is included in the web interface: The feedback component; it will be discussed later in this document.

Administrative tasks

The web interface offers other functionalities than the ones required for the verification itself.

Authentication The web interface will provide authentication to allow users to log in the application.

Authorization Authorization will be implemented so that only registered users will be able to use UPPAAL PM, this will prevent unauthorized persons to run jobs on the cluster. Moreover, this allows each user of the interface to see only his jobs and to allow some users to be administrators.

Management of jobs The interface will allow a user to create, edit, list and delete many jobs. This will improve the easiness of use, and enables a user to create jobs and save them for future use.

Management of users Accessible only by administrators, this functionality will help them create, edit, list and delete the users of the tool.

4.3.2 Site Navigation

To achieve all the functionalities explained above, the web interface will be composed of several pages each containing forms or data shown to the user. To be able to create an optimized interface, we have to know how many pages will be mandatory and what links we must provide between these pages.

Typical Use Case To have a better understanding of the site navigation, we can describe what will be the typical use case of the application:

1. Log in the application;
2. Click “create a new job”;
3. Fill the form: name, description, model file, query file;
4. After validation of the fields, user is taken to the configuration of the job;
5. Delete unused parameters, choose boundary values for parameters and select properties to be checked;
6. Run the job;
7. Wait for results offline or online, watching the progress of the computing on the interface.

Figure 4.6 shows the site map of the web interface with two kinds of pages. The orange ones represent pages or actions that modify the database records whereas the light blue ones represent pages that only show information from the database. On the same figure, the locks indicate secured accesses, from the login page, only registered members can access the rest of the interface, and only authorized users can access the administration.

4.3.3 Look of the interface

The look of the interface itself is not the most important element, but it can help to make the interface easier to use by placing important elements in a place where they will be easily seen, or using a color code for the errors and

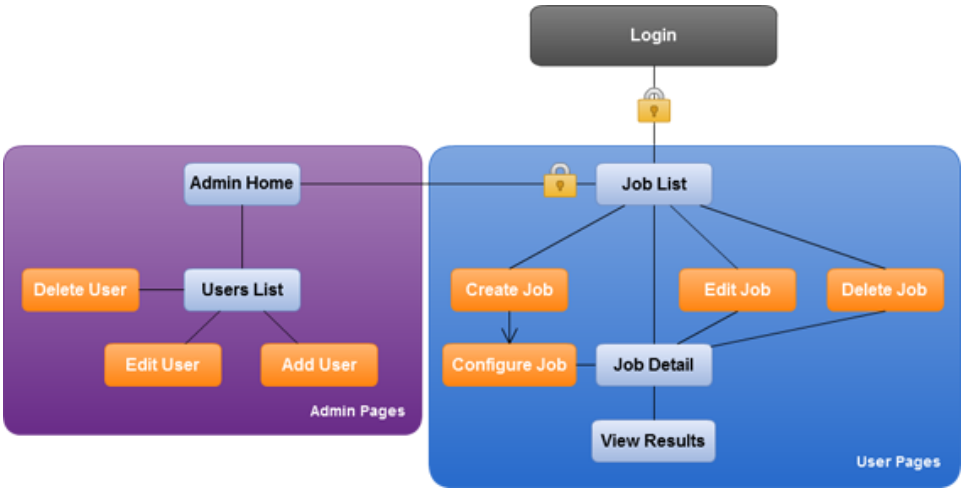

Figure 4.6: Site map of the web interface

so on. Figure 4.7 shows a sketch of the home page of the interface. These sketches are drafts, thus there might be differences with what can be seen on the current interface.

Home Page The left menu will be shown only for the administrators, as it will not be useful for normal users. The *create job* button on the top right is on every page so that a user can create a new job anytime.

As for the home page specifically (fig. 4.7), it will show a list of all user's jobs and quick actions to perform on them such as editing, deleting or aborting/running a job. The view will be paginated to keep the visualization of the list simple.

Job Page Figure 4.8 shows the job page, it shows relevant details about the job: its description, the files used for this job, a progress bar for the execution of the verification, the properties verified and the parameters for the verification.

The *Edit* button is shown only if the job is not started, preventing edition of the job if it has been launched. This button is replaced by a *Show Results* button when the job has been started, allowing the user to see the results of the verification in real-time or after the verification.

4.4 Engine

4.4.1 Component tasks

The engine will perform the *Verify* task, as for the web interface, this task can be split into more specific tasks as shown on figure 4.9.

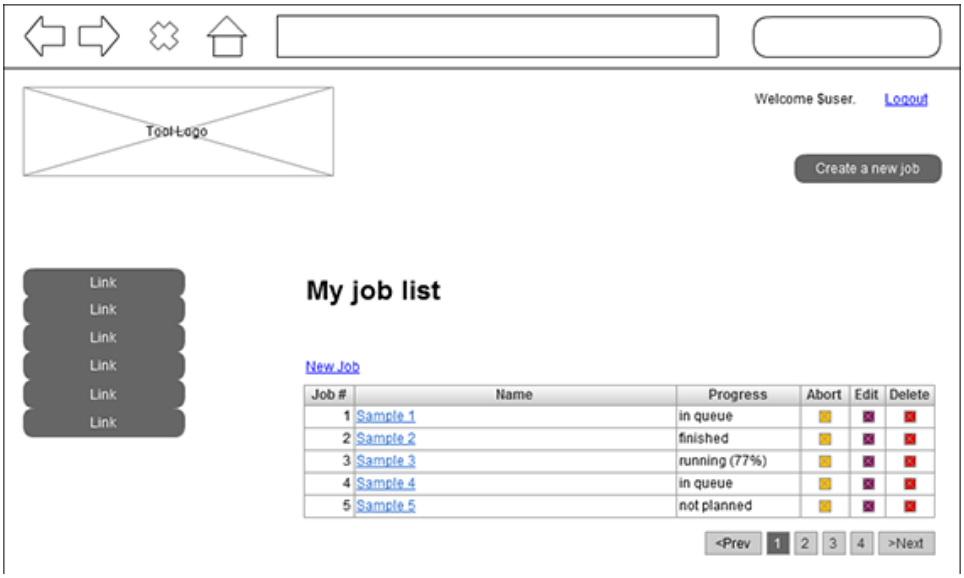

Figure 4.7: Sketch of the web interface - Home page

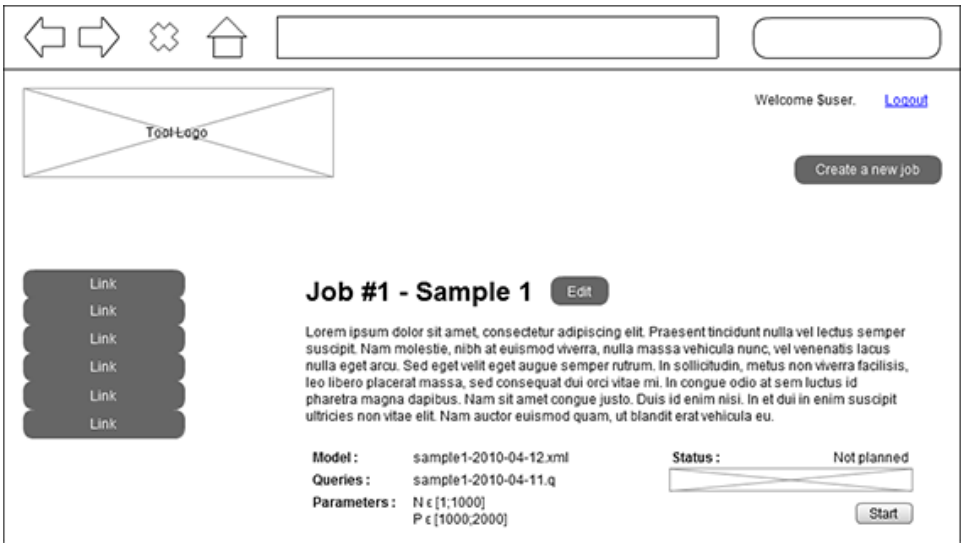

Figure 4.8: Sketch of the web interface - Job page

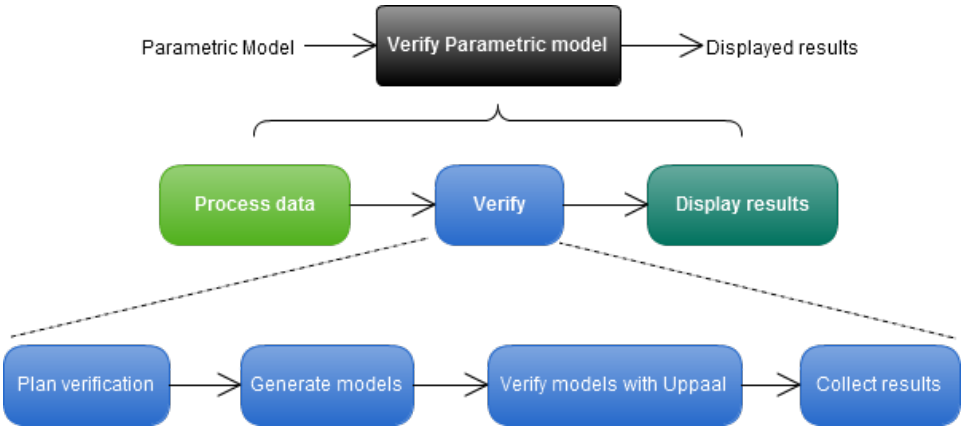

Figure 4.9: The tasks of the engine

1. **Plan verification.** The first task of the engine is to plan the verification using a search algorithm to orientate the search in the parameter space. This task will be repeated with algorithms like *Hill Climbing* or *Simulated Annealing*.

2. **Generate Models.** The next step consists in "instantiating" the models with the specified parameters values.

3. **Verify models with Uppaal .** When a model is generated, it has to be checked with UPPAAL . This task consists in sending the verification to the cluster.

4. **Collect results.** Once the verification is finished, the engine will get the results and store them into the database.

4.4.2 Architecture

To perform these subtasks, the engine is also divided into several components as shown on figure 4.10

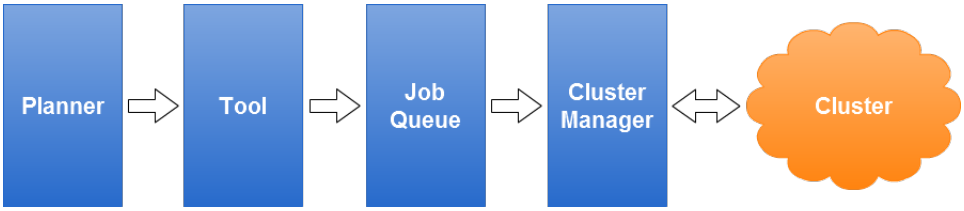

Figure 4.10: Components of UPPAAL PM's verification engine

The planner will first decide for which values of the parameters a model should be instantiated, then, the model files are generated by the tool. Next,

a *Cluster Job* is created and placed into the job queue. When the job queue allows the execution of the Cluster Job, it passes through the cluster manager which transmits the job to the cluster so it can be executed.

Planner

This component is in charge of the planning of the search inside the parameters space. It is modular, to allow different planners to be used. Its role is to decide what values of the parameters should be chosen depending on one of the parametric search algorithms presented in 3.1 (p.10).

Tool

It handles every operation involving the model checking tools, for example the generation of a model, the extraction of the parameters from a model, etc. This component is used by the web interface and the engine. Moreover, it is modular, like the planner component, to allow the use of different model checkers. Figure 4.11 shows the tool for UPPAAL model checker, it is composed of one mandatory module (UPPAAL tool) and additional modules that handle specific tasks.

Figure 4.11: Tool for UPPAAL

Cluster Job

This entity represents an instantiated job. It will contain all the information of the job: the model with the parameters values replaced by those chosen by the Planner, the properties file generated with only the properties to be verified and the .xrsl file needed by the cluster to execute this job.

Job Queue

This component manages all the intermediate jobs created that are sent to the cluster. Its role is to provide a queue for these jobs, so that they will be executed in the order specified by the users - if a job A is run before a job B, the user expects job A to start first. Moreover, as the cluster may not be

used just by UPPAAL PM, using a queue allows to set a limit to the number of jobs sent to the cluster at the same time.

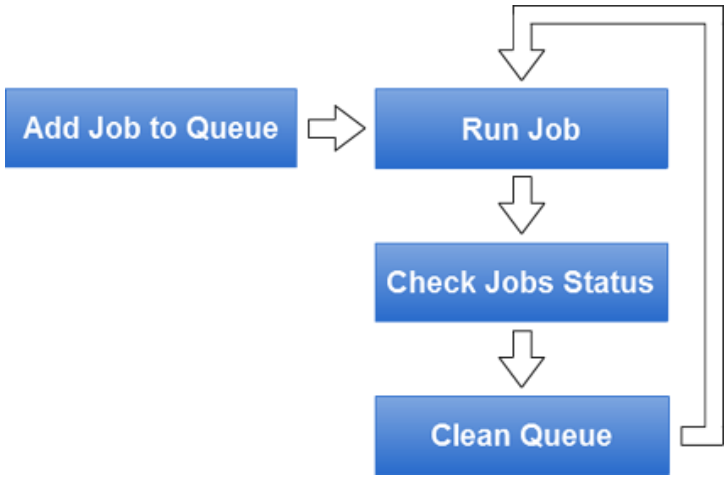

Figure 4.12: The job queue operation

The job queue operation is showed on figure 4.12. It consists of a loop that will run jobs that are not run until the maximum of running jobs is reached, then it checks for the status of the running jobs, if there are finished jobs, the queue is cleaned and it does it again and again until the application closes.

Cluster Manager

This component is responsible for the communication with the cluster; it is where all jobs are sent, where their status is checked, and where the results are retrieved from the cluster back to the application.

Configuration

The engine includes a configuration module to let the administrator of UPPAAL PM set some settings and to allow performance tuning.

4.5 Feedback

4.5.1 Textual representation

The results of the verification will be shown as a list of detailed results. To help the user interpreting them, some improvements have to be brought to this list; they are shown on figure 4.13. The first improvement is to provide the users a filter for the properties, if several properties have been checked, then, the user may want to see the results one property at a time. If a

property is selected, then, the *property* column will be hidden as it is not relevant to show it.

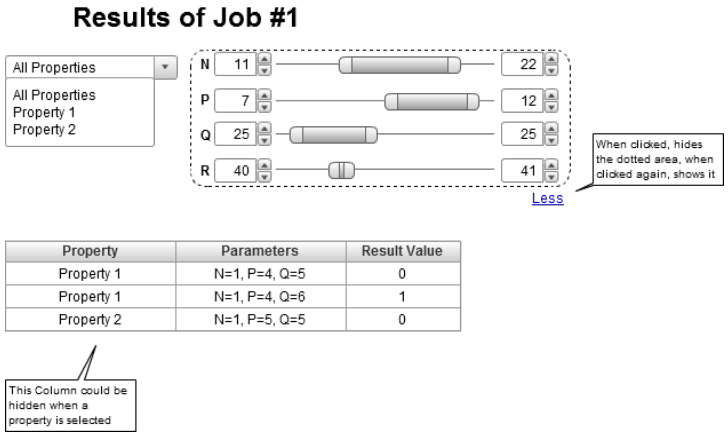

Figure 4.13: Textual results sketch

As shown on figure 4.13, UPPAAL PM will provide the user with optional filters on parameters values that will appear only if the user clicks on **more**. The use of these filters is easy to understand: the user can either specify the values by hand, typing them into the text boxes, or use the graphical controls. These controls can be extended or shortened moving their extremities and moved by drag and drop on their line. Once the values have been chosen, the user clicks on the **Apply** button to update the list.

4.5.2 Graphical representation

As seen earlier, the results will also be shown on a 3D graph as shown on figure 4.14.

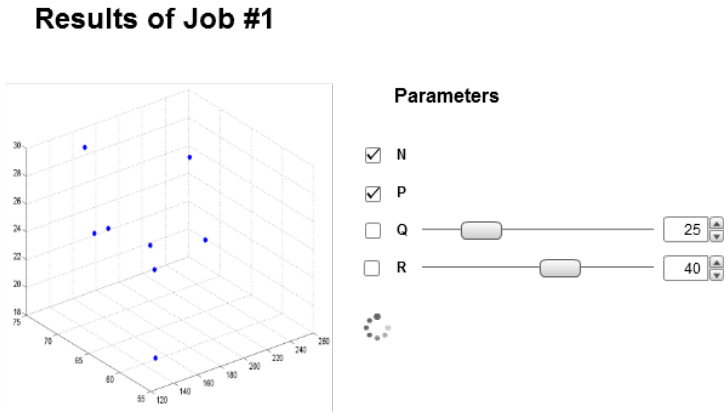

Figure 4.14: Graphical results sketch

As a 3D graph cannot show more than two parameters at the same time on its axes, we have to chose which parameters will be on the axes and specific values for the other parameters. Figure 4.14 shows a sketch of this graphical view: a Java Applet displays the 3D graph and controls that will allow the user to choose which part of the parameter space he wants to see.

To update the graph, the webpage will not be refreshed as it will induce another loading of the applet and it can be really annoying when we want to use it. To perform such an update, we can use either Ajax to update the results, or there might be a way with the Applet itself. One of the possible architectures for this update mechanism is shown on figure 4.15.

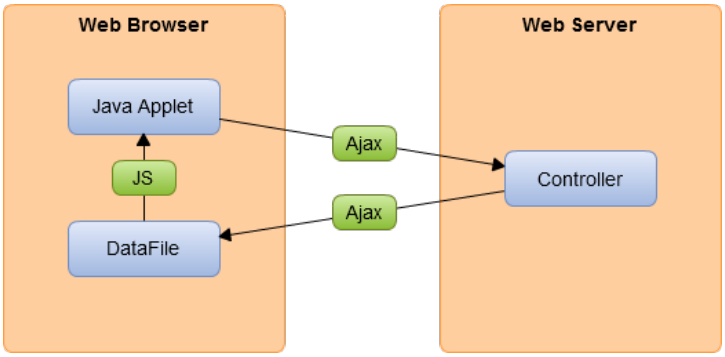

Figure 4.15: Architecture of the graphical view of the results

Chapter 5

Implementation

5.1 *Ruby on Rails* Application

5.1.1 Presentation

Ruby is an interpreted language, thus an interpretor is needed to run scripts or applications such as *Ruby on Rails* . Under linux OS, nearly every distribution provides a package to install this interpretor. To add functionalities to a Ruby script, we can use gems [16]; they are packages created for Ruby which are easily installable with a simple command such as **gems gemname**.

Ruby on Rails can be downloaded and installed with several methods, for example, the basic installation consists in downloading a compressed file ,unzip it and use it. A best and easier solution is to use the *Ruby on Rails* gem.

Even if *Ruby on Rails* is a great and quite complete framework [17], it can lack features needed for one's application, that's why its creators used the gems to extend it. UPPAAL PM uses four gems presented later in this document.

Ruby on Rails uses another great Ruby program named Rake. This program allows to run predefined tasks with the command line, for example creating databases, creating a *Ruby on Rails* application, a new model for the application and so on. These tasks can be customized, allowing the developer to create tasks for its own needs, or to rewrite the default ones to improve them.

5.1.2 Creating Uppaal PM

As a *Ruby on Rails* application, UPPAAL PM was created by the command line with the rails command shown on listing 5.1.

```
1 cd /home/user/ && rails new uppaal_pm
```

Listing 5.1: Creating the *Ruby on Rails* application

This command generates the application directory structure shown on figure 5.1

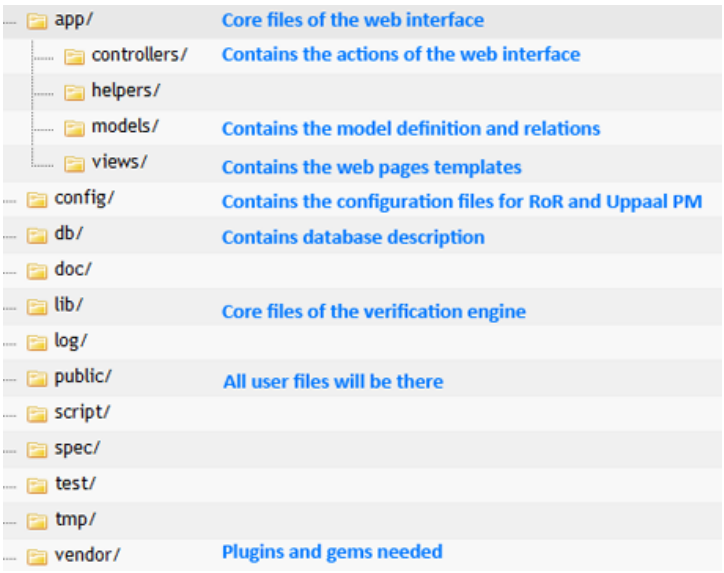

Figure 5.1: Uppaal PM Directory Structure

5.2 Database Component

5.2.1 Creation of the model

The first step while developing a *Ruby on Rails* application is to create the model of this application, this one is created thanks to a command line shown on listing 5.2.

```
1 cd /home/user/uppaal_pm && rails generate model Property
    name:string active:boolean
```

Listing 5.2: Generating a model with Rails

This command will generate a database migration file and a model file; the first will allow *Ruby on Rails* to create the table associated to the entity, and the second will allow the developer to add methods to the object representing the entity and to specify the associations of this one.

5.2.2 Models associations

As shown on figure 4.4 in section 4.2, the entities are linked to each other - A *Job* belongs to a *User* for example. To implement these links, Rails uses a very intuitive syntax inside the model file; listing 5.3 shows the Job model with its relations to other models.

```
1 class Job < ActiveRecord::Base
2   ## Relations with other models
3   belongs_to :user
4   has_many :parameters, :dependent => :destroy
5   has_many :properties, :dependent => :destroy
6   has_many :detailed_results
7   has_many :summarized_results
8
9   ## Code
10 end
```

Listing 5.3: Associations of the Job model

5.2.3 The Job Model

The central model of the application is the *Job* model. As the center of the application, this model is very complete compared to the others, there is more in it than the associations.

Model and Properties Files

The *Job* model is already associated to other models, i.e. information stored into the database, but there is other information to associate to the *Job* model: the UPPAAL model file and the properties file. To associate these files to the job, we used a gem for *Ruby on Rails* named *Paperclip* [18].

Paperclip allows to attach a file to a model, and use it as if it was also a classic model. Listing 5.4 shows how to add attachments to the *Job* model. These files will be stored on the web server, under the public directory of the application.

```
1 class Job < ActiveRecord::Base
2   ## Associations
3
4   ## Declaration of attachments
5   has_attached_file :model, :path =>
        ':rails_root/public/job_files/:id/:basename.:extension'
6   has_attached_file :query, :path =>
        ':rails_root/public/job_files/:id/:basename.:extension'
7
8   ## Code
9 end
```

Listing 5.4: Adding Paperclip's attachments to the *Job* model

Validation

Each time a job is created or edited, we must ensure that the values or files given by the user are correct, or at least not inconsistent. For example, if the user enters "one" instead of "1" for a parameter value, we have to deny the creation or modification and ask the user for a good value.

Ruby on Rails provides a validation framework built-in all models, every model created as default validation for all its fields. This validation is basic, it will just check if the type of the input is the same as the expected one. To add other built-in validations, we can modify the database migration files to add constraints on the fields, for example, **not null** or **a default value** as shown in listing 5.5.

```
1  class CreateJobs < ActiveRecord::Migration
2    def self.up
3      create_table :jobs do |t|
4        ## code
5
6        t.boolean :is_planned, :default => false, :null => false
7        t.boolean :is_queued, :default => false, :null => false
8        t.integer :queue_priority, :default => 0, :null => false
9
10       ## Code
11     end
12   end
13   ## Code
14 end
```

Listing 5.5: Adding constraints on the fields

Even if this basic framework is very useful, it is not enough for sophisticated validation. Fortunately, *Ruby on Rails* provides a generic **validate** method on the models; the argument of this method is the name of a method that will process to the validation. We use this method to validate the model file against its DTD, to eliminate malformed XML files.

To be able to validate an XML document against its DTD, we need to use an XML Library. For this purpose, we use the gem named *Libxml-ruby* [19] which is, actually, a ruby wrapper for the well-known LibXML library for C language; with this library and the validate method, we can now validate the model file when it is uploaded as shown in listing 5.6.

```
1  class Job < ActiveRecord::Base
2    ## Code
3
4    ## Validation of Job while created or edited
5    validate :must_be_valid_xml ## Validates XML with its DTD
6
7    ## Code
8
9    protected
10
11   def must_be_valid_xml
12     ## Code
13     begin
14       doc.validate(dtd_object)
15     rescue
16       errors.add(:model, "is not valid - check XML syntax")
17     end
```

```
18 end
```

Listing 5.6: Implementing our own validator with libXML library

5.3 Web Interface

5.3.1 Controllers

Presentation

A controller is created for each model of the application. A controller offers actions that will allow a user to access the resource or modify it. Every controller inherits from a super class of *Ruby on Rails* and contains public methods named actions. Every action is accessible through the web application, either directly from the URL, for example a **new** action that presents a form to the user, or from another action.

Splitting the application into models helps keeping the code clear and flexible, with a controller for each model, we can split the functions of the web application as well. The actions inside the controllers handle specific operations and this also increases the clarity of the code, and the flexibility of the application.

Routing

Ruby on Rails comes with a nice feature that prints understandable URLs, and can even allow the URL to be used to give commands to the application thanks to a structured writing of the URLs. *Ruby on Rails* follows the RESTful [20] design, which means that every operation on the application resources are linked to HTTP protocol operations, for example an update corresponds to a PUT operation, and so on.

By default, every resource controller (generally corresponding to a model of the application) has seven actions: index, new, create, show, destroy, update, edit. These actions are most of the time enough to create a simple web interface, but in our case, we had to provide several other actions, that's why we added actions to the job resource as shown in listing 5.7.

```
1 map.resources :jobs,
2                :member => {                    # Adding other
                     routes to this resource
3                    :configure => :get,         # Shows the
                         configuration of the job
4                    :updateconfig => :put,      # Allows to update
                         the configuration of the job
5                    :run => :get,               # Allows to run the
                         job
6                    :abort => :get,             # Allows to abort
                         the job
```

```
7                   :results => :get          # Shows the results
                        of the job
8               }
```

Listing 5.7: Job resource mapping in the routes file

5.3.2 Nested Forms

It often happens that two models are linked by a *belongs_to* or *has_many* association, in these cases, we would like to be able to edit the "possessions" of the "owner" model while editing this one. This is the purpose of nested forms.

By default, forms are created for one model only in *Ruby on Rails* , these nested forms allow forms to contain both the model fields and fields for its "possessions". For example, in UPPAAL PM, the **parameter** model is part of the **job** model, and we can edit the parameters inside a job form. To use these nested forms, we have to proceed in two steps. The first one consists in activating the nested forms option in the model that owns the other one, as shown in listing 5.8

```
1 class Job < ActiveRecord::Base
2   ## Code
3   accepts_nested_attributes_for :parameters, :allow_destroy =>
        true, :reject_if => lambda { |a| a[:name].blank? }
4   accepts_nested_attributes_for :properties
5   ## Code
6 end
```

Listing 5.8: First step, activating nested attributes

The second step is to create the nested forms, an example is shown on listing 5.9, the important part is the *fields_for* helper, this is the method that will allow the form builder to create fields for the other model.

```
1 <% form_for @job, :url => { :action => "updateconfig" } do |f|
      %>
2   <%= f.error_messages %>
3
4   <h2>Parameters</h2>
5
6     <% f.fields_for :parameters do |builder| %>
7       <%= render 'parameter_fields', :f => builder %>
8     <% end %>
9     <!-- CODE -->
```

Listing 5.9: Second step, creating the nested form

5.3.3 Views

The views of *Ruby on Rails* are linked to an action of a controller. For example, if we have a job#new action returning html content[1], it will need a view at views/jobs/new.html.erb.

Every view can access the instance variables of the controller action to show their values of bind them to forms. In UPPAAL PM there is nothing special concerning these views as they are quite simple.

5.3.4 Gems

Authlogic

Authentication and Authorization are two features often seen in web applications. It takes time to implement an efficient solution for them, that's why UPPAAL PM uses Authlogic [21], a gem that provides easy-to-use and ready-to-go authentication solution with a syntax similar to ActiveRecord, the ORM shipped with *Ruby on Rails* .

Will Paginate

As UPPAAL PM has to deal with lots of information, it has to present it in the best way possible. To do this, a simple solution is to use pagination to limit the number of information shown on a page. Implementing pagination can be complicated and time consuming; to avoid these problems, we used the gem **Will paginate** [22].

5.4 Verification Engine

5.4.1 Overview

The verification engine of the application is located inside the **lib/** directory. The initialization of the different components of the engine takes place inside the config/environment.rb file, which is the initializer of the Rails application also; this improves the integration of the engine inside the framework.

Architecture of the engine As seen in section 4.4.2, the engine is divided into several components, the implementation of this architecture is shown on figure 5.2.

On this diagram, we can recognize the job queue component as well as the cluster manager; each of these components is implemented with one class in the common Ruby module[2]. The two other components - the planner and

[1] Actually, if we have a route mapped to a GET method of HTML protocol

[2] A Ruby Module is the equivalent to a Java Package

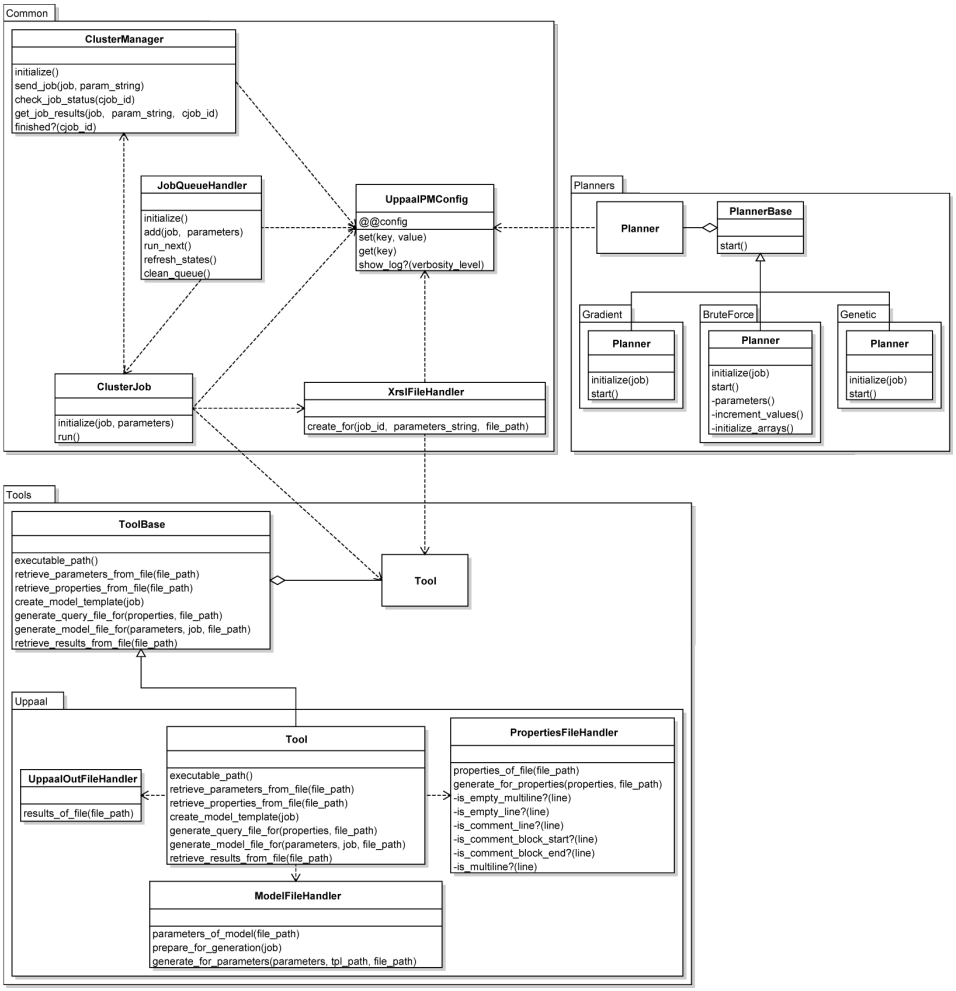

Figure 5.2: UML diagram of the engine

the tool - are actually implemented as several classes due to their modularity, this will be explained in section 5.4.2.

Multi-threading The engine is multi-threaded and it will be executed in its own threads to avoid hangs in the web interface; figure 5.3 shows how threads are created and the links between them.

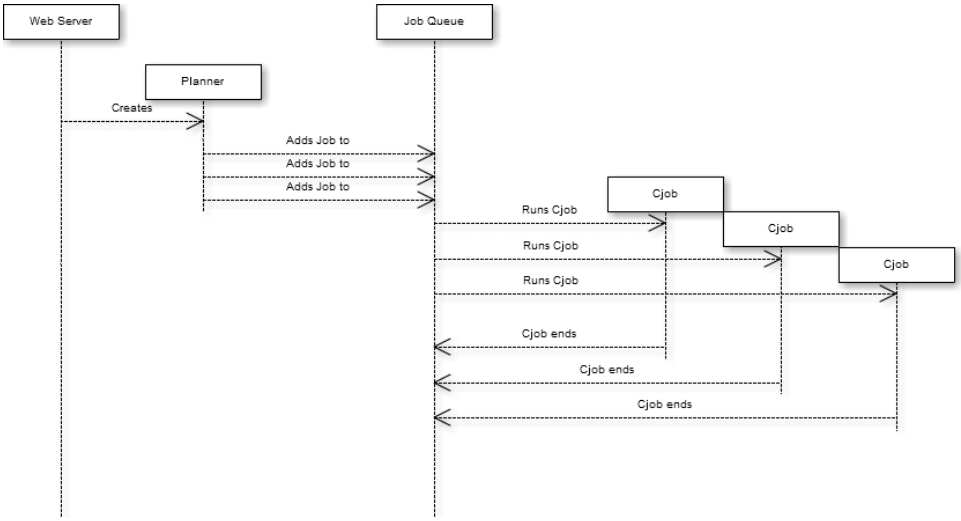

Figure 5.3: Threads of the verification engine

When the application starts, two threads are created; the first, the main one, is for the execution of the web server, the second is for the *Job Queue* which is destroyed when the application stops. During the execution of the application, when a user wants to start the verification of a job, a planner is instantiated and a thread is created for its execution; basically, the planner adds jobs to the queue and, either it terminates and the thread disappears - like the brute force planner - or it can wait for the results to add other jobs to the queue. Once the jobs are added to the queue, they are transformed into Cluster Jobs and they will eventually be run. When a Cluster Job is run, a new thread is created for its execution also, and, once it completes, the thread is destroyed.

5.4.2 Modularization

Factory Pattern To implement modularity in UPPAAL PM, we use the design pattern *Factory*. This design pattern, represented on figure 5.10, allows to instantiate an object whose class is known only at execution time.

```
1 class Car
2   def build(type)
3     type::new
```

```
4   end
5 end
6
7 class RaceCar; end
8 class CityCar; end
9
10 factory = Car.new
11
12 race_car = factory.build RaceCar
13 city_car = factory.build CityCar
```

Listing 5.10: Factory Pattern in Ruby

In this example, we have one class named *Car* and two other classes respectively named *RaceCar* and *CityCar*. The builder function will create a new instance of the type passed as argument and return it.

Interface with Ruby To create different modules for one component, there must be a common set of methods that the application can invoke when necessary. To achieve this, we could use an interface, but they do not exist in Ruby language. There's a little workaround to create the behavior of an interface using inheritance; it is shown on listing 5.11.

```
1 class Car
2   def move
3     raise NotImplementedError
4   end
5 end
6
7 class RaceCar < Car
8   def move; end
9 end
10
11 class CityCar < Car
12   def move; end
13 end
14
15 def move_car car
16   car.move
17 end
18
19 race = RaceCar.new
20 city = CityCar.new
21
22 move_car race
23 move_car city
```

Listing 5.11: Interface workaround for Ruby

Complete method To implement the modularization of the Tools and the Planners, we use a mix of these two solutions with a little trick that

avoids the instantiation of the factory class. This solution is shown in listing 5.12.

```
1  class Car
2    class << self
3      def new klass
4        klass::new
5      end
6    end
7    class CarBase
8      def move
9        raise NotImplementedError
10     end
11   end
12 end
13
14 class RaceCar < Car::CarBase
15   def move; end
16 end
17
18 race_car = Car.new RaceCar
```

Listing 5.12: Complete solution for modularization

This solution is used for both the Planners and the Tools. It allows to add new algorithms and new model checkers to the application easily and with only a small knowledge of the rest of the application.

5.4.3 Components

Job Queue

The Job Queue has an important role in UPPAAL PM, it is responsible for the processing of the intermediate verification and the good behavior of the application. All jobs goes through the queue before being effectively computed, figure 5.4 shows the UML representation of the Job Queue.

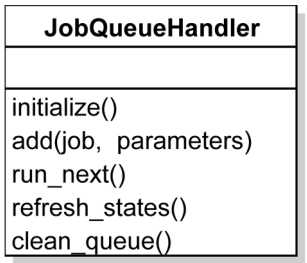

Figure 5.4: UML representation of the Job Queue

It is actually composed of only one class that is instantiated in its own thread when the application starts. The way it works is the following:

1. It receives jobs at any time and puts them in the queue as Cluster Jobs.
2. Every N seconds[3], the Job Queue is refreshed, all running jobs statuses are checked on the cluster and updated inside the queue.
3. Then, all finished jobs are deleted from the queue
4. Finally, if there are not started jobs and if the number of running jobs is under the maximum authorized by the configuration, the Job Queue runs new jobs.
5. These steps are repeated until the application terminates.

Cluster Manager

The Cluster Manager, whose UML representation is shown on figure 5.5, is responsible for the communication with the cluster. It is instantiated when the application starts and used by the Job Queue and the Cluster Jobs to perform several actions.

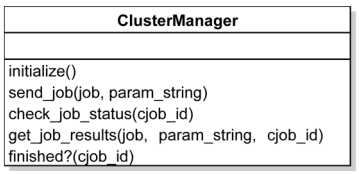

Figure 5.5: UML representation of the Cluster Manager

It can send a cluster job to the cluster, check the status of a cluster job on the cluster itself and once a cluster job is finished, it gets its results .

For development purpose and to extend the possibilities of Uppaal PM, the Cluster Manager can send commands through the cluster gateway even if Uppaal PM is not installed on the same server as the gateway. This is possible thanks to SSH host key authentication, indeed, commands are sent thanks to SSH between the web server and the cluster gateway.

Planner

The Planners are responsible for the planning of the verifications, they are the components that optimize the verification of the parametric models. They are modular as shown in figure 5.6.

To be compatible with Uppaal PM, a planner just has to inherit from PlannerBase class and implement the start() method which actually starts the planning of a verification. There are two kinds of planners: those which

[3]this value is specified in the configuration

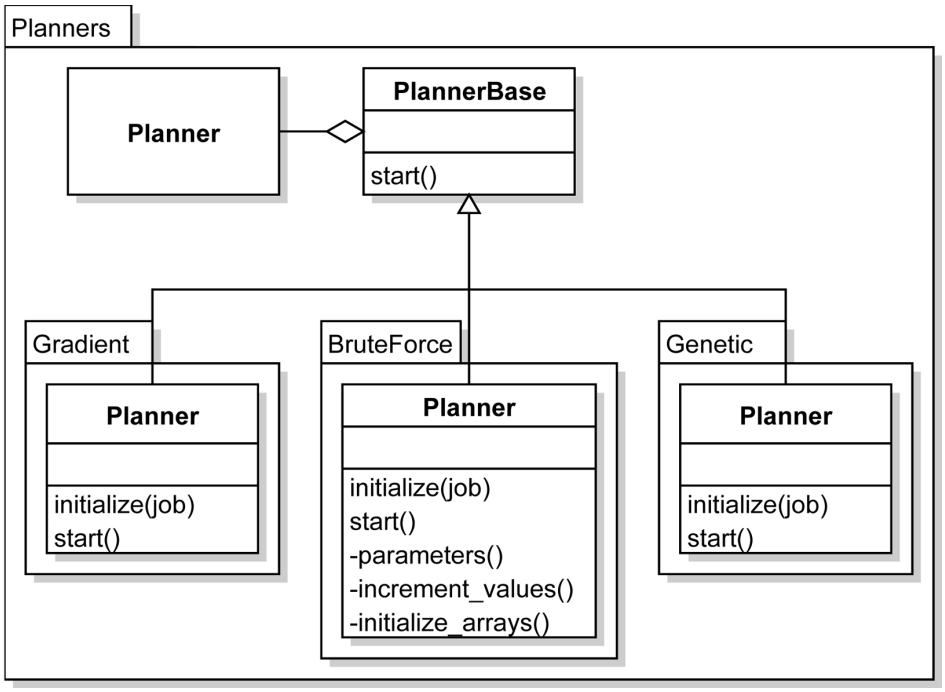

Figure 5.6: UML representation of the Planners

will send all intermediate jobs and terminate, and those which will send intermediate jobs by little number, wait for their results and continue like this until termination. Currently, there's only one planner implemented, it is of the first kind and based on the brute force algorithm.

Tool

The Tools encapsulate the model checkers used to verify the parametric models. They are also modular, figure 5.7 shows the only tool implemented currently : Uppaal model checker.

The tool for Uppaal has been developed around four classes: the main one which inherits from ToolBase (Tool), one to handle the model file (ModelFile), one to handle the properties file (PropertiesFile) and the last one to handle the output file produced by Uppaal (OutputFile). Each of these classes have a specific role, but they are specific to Uppaal . Each tool can have its own classes under one condition: implement at least the Tool class which has to inherit from ToolBase.

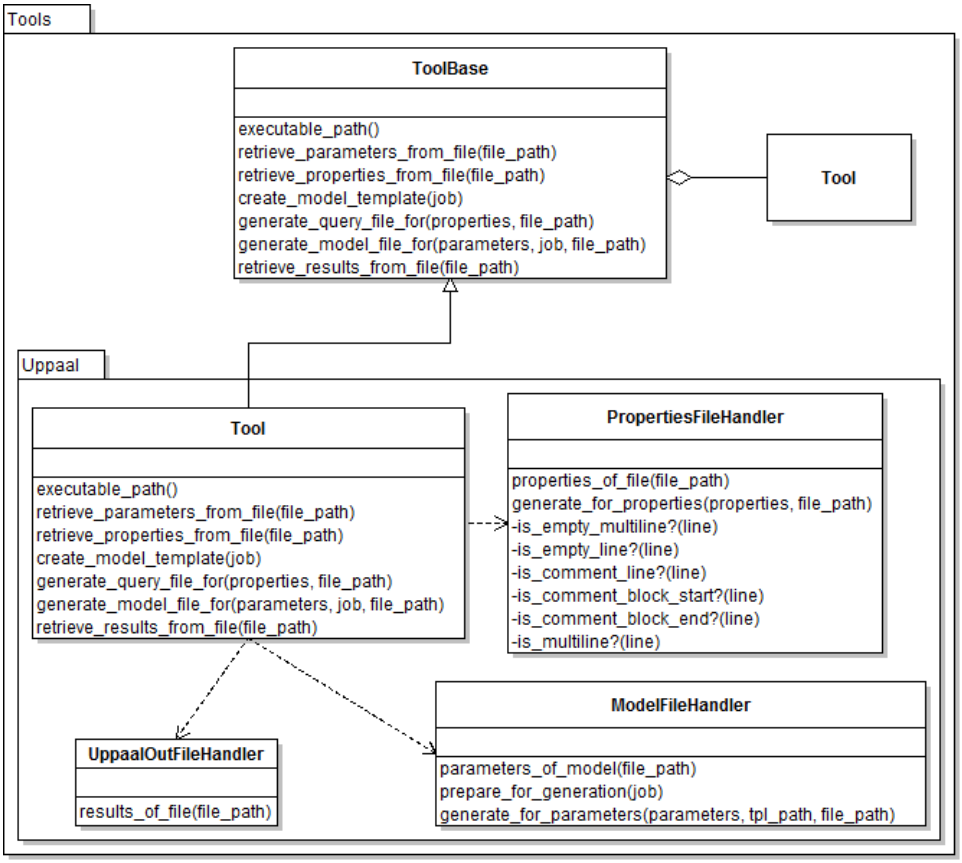

Figure 5.7: UML representation of the Tools

Conclusion and Future Work

Uppaal PM brings a new approach to parametric model checking. Its "instantiation" of the models allows it to work with parameters anywhere inside the model, it even provides a supremum search when the model checker used can find them in classic timed automata models.

The work done and showed in this report lays the foundations of Uppaal PM, the current version gives a basic parametric model checker, which can be assimilated to an automation of parametric model checking, as it is based on a brute force algorithm.

Moreover, Uppaal PM is easy-to-use, configurable, and easy-to-install. Indeed, it is a common *Ruby on Rails* application that has to be deployed and configured.

The modularity of Uppaal PM can bring new tools, new algorithms which could expand its features. It will be able to gain accuracy and optimized calculations while verifying parametric models.

There is still work to be done to implement the algorithms discussed in this report and create the feedback interface, an important part of this tool.

Appendices

Appendix A

Cluster

A.1 Presentation

A cluster is a group of computers designed to execute long and complex computations. It acts as a black box from the outside: a client sends a job to the cluster, then the cluster executes it when possible. The client asks the cluster whether the job sent is finished or not; if yes, the client asks the results to the cluster. A cluster provides asynchronous computing: the client sends a job, and the cluster is then completely managed by the cluster, while the client can do something else in the mean time.

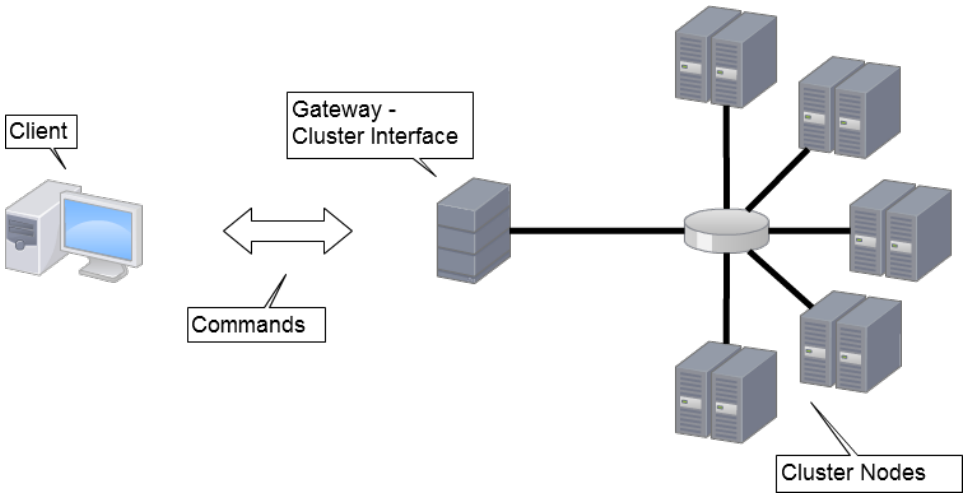

Figure A.1: Architecture of a Cluster

The computers inside the cluster are named cluster nodes and they are linked by a private network, not accessible from the outside. There is just one computer inside the cluster that is visible from the outside, it is the gateway to the cluster. This computer is responsible for the management of

the jobs sent to the cluster. It determines which jobs can be run depending on the cpu time required by the job and the order of submission of the jobs. Figure A.1 shows the architecture of a cluster.

A.2 ARC Interface

To be able to receive jobs, the gateway must provide commands to the client, this is why the ARC interface exists. Using several commands, a client can send jobs, verify their statuses and get the results of the job.

A.2.1 XRSL Files

To tell the cluster what job we want it to compute, we have to use an XRSL file. As shown in listing A.1, this file contains all the information about the job that has to be executed.

```
1 &(executable=/bin/sleep)
2   (arguments=30)
3   (stdout="out.txt")
4   (stderr="err.txt")
5   (cputime=10)
6   (outputfiles=("out.txt" "")("err.txt" ""))
```

Listing A.1: XRSL file example (hello.xrsl)

Here, the executable that will be called at execution is /bin/sleep (for example purpose of course), it takes 30 as argument, standard output will be logged into out.txt and error output in err.txt.

A.2.2 ngsub

To submit a job to the cluster, we have to use the *ngsub* command as shown in listing A.2.

```
1 ngsub -c benedict.grid.aau.dk -f hello.xrsl
```

Listing A.2: ngsub usage

The -c switch indicates which cluster has to be used, and the -f switch indicates which XRSL file used to describe the job.

This command returns a unique job identifier that looks like: gsiftp://benedict.grid.aau.dk:2811/jo

A.2.3 ngstat

To verify the status of a job sent to the cluster we use ngstat command as shown in listing A.3.

```
1 ngstat
    gsiftp://benedict.grid.aau.dk:2811/jobs/2463812857668701043587263
```

Listing A.3: ngstat usage

This command returns the status of the job in the cluster. There are several codes to represent the status of a job, but the most common are :

- SUBMITTING - When a job is being submitted to the cluster queue
- EXECUTING - When a job is running in the cluster
- FINISHED - When a job is finished
- FAILED - When a job has failed

A.2.4 ngget

To get the output files of a finished job, we use ngget command as shown in listing A.4.

```
1 ngget
    gsiftp://benedict.grid.aau.dk:2811/jobs/246381285766870104358726 3
```

Listing A.4: ngget usage

This command stores the output files in the current directory of the running shell, under a folder named by its id.

Appendix B

Development architecture

The application has been developed under Linux OS for compatibility issues with some libraries, and because Linux will be the host of UPPAAL PM in the end. A *Ruby on Rails* application also requires a Database Management System (DBMS), we have chosen MySQL because it is open source, and usually present on every Linux web server.

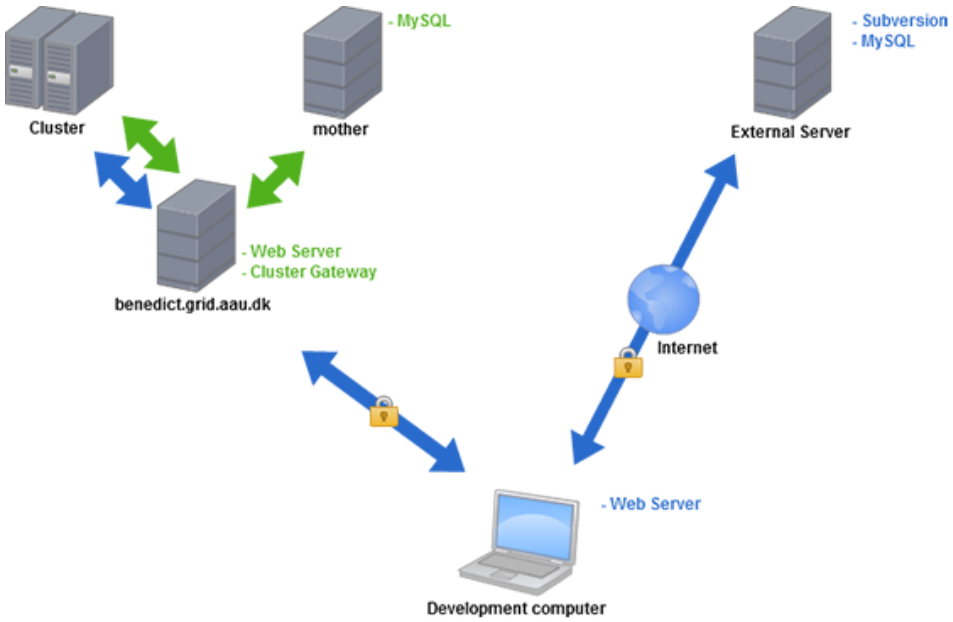

Figure B.1: Development organization

For development purpose we also had to set ssh host authentication between the computer where UPPAAL PM was executed and *benedict.grid.aau.dk* to allow automatic execution of commands on this server. During the development, we also used a subversion repository on an external server (the same where the database was located).

Figure B.1 shows the organization set for the development of Uppaal PM. All blue items are for development and testing purpose, whereas green ones are for production. For development purpose, an SSH tunnel is present between the development computer and the external server to access the database as if it was local. The development web server runs on the computer and sends commands to the cluster through *benedict.grid.aau.dk*. When the application will be deployed on the production web server, the database will be on *mother*. The production architecture is unaware of the development structure built around it, allowing development on one side and production use on the other.

Bibliography

[1] Uppsala University and Aalborg University. UPPAAL website. http://www.uppaal.com.

[2] Gerd Behrmann. UPPAAL CORA website. http://www.cs.aau.dk/ behrmann/cora/.

[3] Marius Mikucionis. UPPAAL TRON website. http://www.cs.aau.dk/ marius/tron/.

[4] Alexandre David. UPPAAL TIGA website. http://www.cs.aau.dk/ adavid/tiga/.

[5] Tom Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech : The hybrid technology tool. http://embedded.eecs.berkeley.edu/research/hytech/.

[6] Étienne André. Imitator ii. http://www.lsv.ens-cachan.fr/ andre/IMITATOR2/.

[7] Étienne André, Thomas Chatain, and Laurent Fribourg. An inverse method for parametric timed automata. *International Journal of Foundations of Computer Science*, June 2009.

[8] Étienne André. Behavioral cartography of timed automata, August 2010. http://www.lsv.ens-cachan.fr/ andre/documents/2010-08-28-Behavioral-Cartography-of-Timed-Automata.pdf.

[9] T. Hune, J.M.T. Romijn, M.I.A. Stoelinga, and F.W. Vaandrager. Linear parametric model checking of timed automata. *Journal of Logic and Algebraic Programming*, pages 52,53, 2002.

[10] Wikipedia Community. Genetic algorithm. http://en.wikipedia.org/wiki/Genetic_algorithms.

[11] Danikos Software. Intelligent search. http://www.learnartificialneuralnetworks.com/intelligentsearch.html#annealing.

[12] Wikipedia Community. Model view controller. http://en.wikipedia.org/wiki/Model-View-Controller.

[13] Fabien Potencier. Symfony website. http://www.symfony-project.org.

[14] David Heinemeier Hansson. Ruby on rails website. http://www.rubyonrails.org.

[15] Martin Pernollet. Jzy3d website. http://code.google.com/p/jzy3d/.

[16] Nick Quaranto. Ruby gems website. http://rubygems.org/.

[17] Ruby On Rails Community. Guides for ruby on rails. http://guides.rubyonrails.org/.

[18] Nick Sieger. Paperclip git webpage. http://github.com/thoughtbot/paperclip.

[19] Dan Janowski. Libxml ruby project website. http://libxml.rubyforge.org/.

[20] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. 2000. http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm.

[21] Ben Johnson. Authlogic git webpage. http://github.com/binarylogic/authlogic.

[22] Mislav Marohnic. Will paginate git webpage. http://github.com/mislav/will_paginate.