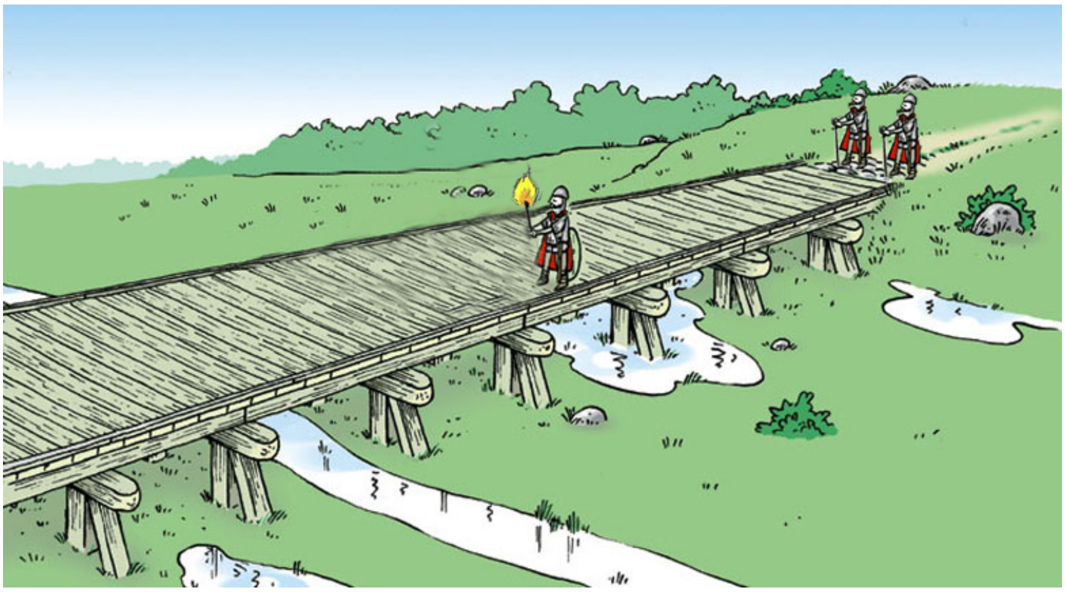
Distributed parameter sweep for UPPAAL *models*



DAT5, autumn 2010

Group d503a

Department of Computer Science

Aalborg University
January 11$^{\text{TH}}$, 2011

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
DK-9220 Aalborg East
Phone +45 9940 9940
Fax +45 9940 9798
http://www.cs.aau.dk

Title: Distributed parameter sweep for UPPAAL models
Theme: Distributed Systems
Project period: DAT5, autumn 2010
Project group: d503a
Participants:

Jimmy Merrild Krag

Peter Schmidt Freiberg

Brian Villumsen

Supervisor: Brian Nielsen
Copies: 5
Page count: 59
Appendices: 2
Finished: January 11th, 2011

Synopsis:

This report introduces UPPAAL PARMOS — a parameter sweep application for distributing the task of verifying UPPAAL models on a grid. The solution has been archieved, by examining exisiting parameter sweep frameworks and other work within this area. UPPAAL PARMOS provides a web service front end to the parameter sweep application service. This service provides a complex set of useful operations for submitting, accessing and mutating information. We provide a basic web front end mainly for the purpose of demonstration. A parameter sweep is to be specified using a custom developed syntax that allows both goal, condition and parameter interval to be specified. Each part of the entire system is developed using a module based approach which gives a great deal of flexibility in optimization and customization.

Preface

This project was carried out in the period from September 5th, 2010 to January 11th, 2011. It is the result of a study conducted by two DAT5 students and one F9S student at the Department of Computer Science at Aalborg University. The initial idea originates from a project proposal concerning the distribution of the verification of parametrized UPPAAL models on grids. The primary source of inspiration was given by our supervisor Brian Nielsen, yet as the bibliography of this project indicates, there were multiple sources used in attaining the needed knowledge.

Aalborg, January 11th, 2011.

Contents

1 Introduction

As computational tasks have advanced from solving of small puzzles to large scale complex problems, the workload for more computational power has increased. The explanation for this demand is primarily found in the rapid increase of time for a computational task to finish, yet also the available amount of resources is crucial. Depending on the individual tasks, solutions to manage these large scale problems exists and they mostly consist of distributing the workload on a number of computers, either coupled together as a cluster or more loosely over an ad-hock network.

Because of the evolution of the hardware in modern computers, the types of problems that can be modeled in such a way that computers can understand them have increased. This evolution, combined with workload distribution, has allowed new opportunities for scientific insight, in terms of acquiring knowledge from simulations rather than experiments or mathematically proven theories. Depending on the scientific area, models can vary in size and complexity, hereby also varying the amount of data that need to be processed.

This area of modeling, simulating and distributing the workload using grid computing is known as eScience [eScience Center, 2010]. Using tools and techniques from this area, scientists can change the borders of their experiment from economical, hazardous, ethical or simply impossible with current technology, to what that is computationally possible within budget. This means new opportunities and research areas that previously were limited by non-science reasons can be discovered.

When focusing on the computer science area, several possibilities of utilizing eScience beneficially can be spotted. One of these possibilities is in the area of model checking. Model checking as concept is all about the validation of a model, in terms of checking certain properties given certain conditions. Different model checking tools exists to model different scenarios, this report focuses on the model checking tool called UPPAAL. UPPAAL is a tool for modeling, simulation and verification of real-time systems, especially systems where timing aspects are critical.

The manual process of verifying a single model using the UPPAAL verifier is for the user simple and straightforward once the model has been created. However, when a model needs to be verified multiple times with only minor changes it quickly becomes both a uniform, tedious and time consuming task. This task could be automated in order to speed up the process and avoid flaws. Such an automated process is known as a parameter sweep.

The purpose of this report is to — if possible — design and implement a Parameter Sweep Application (PSA) for UPPAAL that distributes the workload through a grid.

1.1 Related work

This project stems from an idea of applying distributed computing to ease the task of model checking. Previous work within this field has been conducted at our department by Romain Sertelon [Sertelon, 2010]. We have studied this work and found that he suggest some interesting ideas. However, even though we share a related problem statement, our goals, design and implementation have taken us in another direction.

Reading guide

The content of this study is divided into chapters, sections and sub-sections. The content chapters are identified by numbers and the appendix chapters by letters. Sections are referred to by numbers e.g. 1.2.3 refers to chapter 1, section 2, sub-Section 3.

This report also makes use of acronyms for better readability. Acronyms are introduced the first time they are used with the full expression, followed by the acronym enclosed in parenthesis, and thereafter only the acronym is used, with a few exceptions for titles, quotes, etc. For example here IP is introduced, and hereafter only IP is shown. If the acronym had an s appended, it means plural form. In addition, a list of all acronyms is provided on page 55 for reference.

2 Distributed parameterized model checking

In this chapter we investigates the parameter sweep methodology and its possibilities in relation to model checking. We examine whether it is possible to gain advantages by distributing the workload. We study use-cases from currently and previously existing PSAs. Both how they are conducted and what — if any — framework is being used. This chapter is primarily based on Wibisono et al. [2008] and Casanova and Berman [2003].

Throughout the rest of the report, a task is referred to as a description of a PSA, describing the experiment that should be conducted, and with witch parameters. Each instance of the experiment with a specified set of parameter values, is referred to as a job.

2.1 Methodology

It is a two-phase procedure when using a scientific model-based approach for abstracting a computational problem. The first phase is developing a model that represents a given system and the second phase is validating the model.

The process of validation often requires adapting model value constants such that the model satisfies certain criterias. The number of allowed adjustable values in the model is called the parameter space. This parameter space grows exponentially as the number of parameters is increased, see equation 2.1, and each valid parameter combination could theoretically be a valid solution thus they should all be tested.

$$\mathbb{P} = \{M_0 \times M_1 \times \cdots \times M_n\} \tag{2.1}$$

However, as these tests are both time consuming and a uniform workload, an optimization scheme that minimizes the number of test and automates the process would be useful. The automation of testing different sets of parameter combinations is known as a parameter sweep.

A PSA is an application that has a loosely coupled relation to other instances of the same application. This characteristic allows for a parallel execution of the application in order to minimize the total runtime of the task. Furthermore, in order to reduce the number of jobs executed, a process of gradual refinement should be started, where an optimization schema is applied on the parameter space, using the data gathered from previously executed jobs. This process is depicted in Figure 2.1 on the following page. As illustrated in Figure 2.1 it is a continuing process that should proceed until a desired solution have been reached or all possible combinations in the parameter space has been tested.

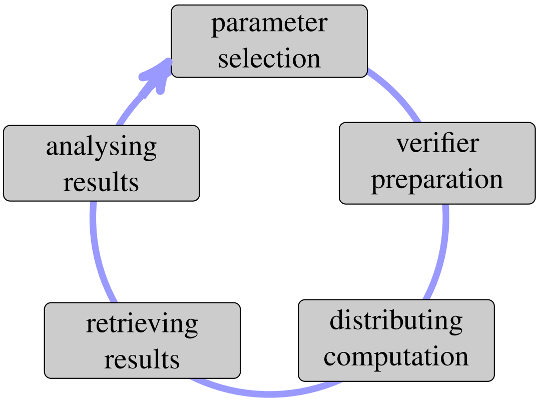


Figure 2.1: Flow of a Parameter Sweep Application

2.2 The UPPAAL tool

UPPAAL [UPPAAL Team, 2009] is a tool for modeling, simulation and verification of real-time systems. The most frequent use of UPPAAL is in areas where timing aspects are critical, such as real-time controllers or communication protocols. UPPAAL is usable for systems that can be modelled using timed automata, which are finite state machines that have been extended with clocks. This allows for simulation and verification of system properties. The idea of model checking is to do an exhaustive search that processes all possible dynamic behaviors of a system with the purpose of verifying specific system properties i.e. reachability properties.

A system is viewed as a network of processes composed of locations and connections between locations. These connections are called transitions and define how the system behaves. Constraints, called guards, are expressions that can be added to the transitions allowing for dynamically enabling or disabling the corresponding transition. UPPAAL also supports the concept of continuous time, allowing for both progress conditions on when transitions needs to be taken, and conditioning transitions on clocks. In order to substantially limit the state space when veritying, UPPAAL is limited to bounded integers and boolean data types in the models. However, as boolean values easily can be expressed as an integer it is possible to reduce the parameter space for models as in equation 2.2.

$$\mathbb{P} = \{M_0 \subset \mathbb{Z} \times M_1 \subset \mathbb{Z} \times \cdots \times M_n \subset \mathbb{Z}\} \tag{2.2}$$

The UPPAAL toolbox includes two different user interfaces

- a graphical interface for editing, simulating and viewing the verification of models
- and a console based model checker engine.

Although the graphical interface is using the model checker engine when verifying models, the model cheker engine is independent of the graphical interface and can be run with command line arguments.

The model created using the graphical interface is stored in a file using an XML based encoding. The system properties for the model checker to verify are expressed using a query language. They can be specified in the graphical interface which will store them in a text file with custom encoding. There exists a full grammar for the query language yet to summarize and provide examples, the queries in list 2.3 are all valid queries.

$$\begin{aligned} \exists\langle\rangle\ p &: there\ exists\ a\ path\ where\ p\ eventually\ holds. \\ \forall[]\ p &: for\ all\ paths\ p\ always\ holds. \\ \exists[]\ p &: there\ exists\ a\ path\ where\ p\ always\ holds. \\ \forall\langle\rangle\ p &: for\ all\ paths\ p\ will\ eventually\ hold. \\ p \longrightarrow q &: whenever\ p\ holds\ q\ will\ eventually\ hold. \\ & \quad p,\ q\ are\ state\ formulas. \\ \forall[]\ not\ deadlock &: a\ special\ case\ that\ checks\ for\ deadlocks. \end{aligned} \tag{2.3}$$

As the model checking engine is independent of the graphical interface and implemented as a console, it is possible to start the process of verification by using a command line interface, providing options and directions to the model and query file, as parameters to the model checker executable. The model checker executable is called *verifyta* and is implemented in C++ and available for both Linux and Windows operating systems.

2.3 UPPAAL models

For illustrating the UPPAAL models this section covers two models, a demonstration developed model and a model developed for industry use.

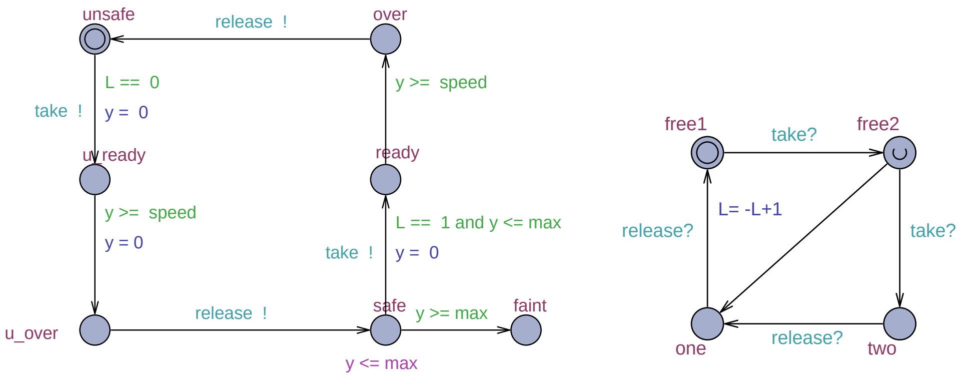
2.3.1 Bridge

The Bridge model models a situation with four Vikings that are about to cross a bridge, from a unsafe to a safe side, in the middle of the night. The bridge only supports two Vikings at all time and in order to walk over the bridge one Viking need to bring a torch, which they only have one of. The Vikings takes 5, 10, 20 and 25 minutes each way in order to cross the bridge and there is a swamp which emits toxic gasses on the safe side of the bridge so they need to hurry up. Given this scenario, different questions can be asked e.g.

Does a schedule exist where all four Vikings crosses the bridge?
What is the longest amount of time a Vinking can wait on the safe side - close to the swamp - before dying of toxic swamp gasses?

Table 2.1: Questions

This scenario can be modelled using the UPPAAL toolbox. This is done by creating two automata templates called *Viking* and *Torch* as depicted in Figure 2.2 on the next page, providing global declarations and a system definition. In UPPAAL templates can include declarations and this is also the case in the Bridge model. While the Torch template does not include declarations the Viking template contains a clock *y*, which represent the amount of time the Viking stays close to the toxic swamp, and a parameter *speed* representing the bridge-crossing time. This template technique allows for rapid duplicating of automatas which is needed in the Bridge model where four Vikings exists.



(a) (b)

Figure 2.2: Automata of a Viking (a) and a Torch (b) for the Bridge model.

Listing 2.1 contains the declarations that are needed. There are need for two communication channels *take* and *release*, a variable *L* containing information of the torch use and a clock called *time* for keeping track of the total walking time. Additionally, five constants are declared globally in order to ease fitting of their values. The *slow*, *slowest*, *fast* and *fastest* constants represents the walking time for each of the Vikings, respectively. The *max* constant is the parameter by which the effect of the toxic gasses affects and kills the Vikings.

```
chan take, release;
int L;
clock time;

const int max = 30;
const int fastest = 5;
const int fast = 10;
const int slow = 20;
const int slowest = 25;
```

Listing 2.1: Global declarations for the Bridge model

Lastly the system definition for the Bridge model is written as in Listing 2.2. Four Vikings and a single light torch are instantiated and the system is declared as containing both the Vikings and the single torch.

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

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

Listing 2.2: System definition for the Bridge model

This system description is stored using an XML based encoding, and the XML content is included in this report and can be found in Appendix A.1 on page 49. In Equation 2.4 and 2.5 follows the queries for answering the questions of table 2.1.

$$\exists\langle\rangle Viking1.safe \wedge Viking2.safe \wedge Viking3.safe \wedge Viking4.safe \tag{2.4}$$

$$inf\{Viking1.safe \wedge Viking2.safe \wedge Viking3.safe \wedge Viking4.safe\} : time \tag{2.5}$$

Given that property 2.4 holds, property 2.5 can be used to find the time for the Vikings to cross the birdge.

However, given the individual walking times of the Vikings there is a limit for how powerful the toxic can be before it becomes impossible for the Vikings to cross the bridge at all. This value can be found by utilizing an automated parameter sweep on the model with the constant max as single parameter.

2.3.2 Danfoss EKC 204A temperaure controller

The UPPAAL tool is not only applicable within the science community; industrial companies can also take advantages of UPPAAL in their Research and Development department. This section describes the use of UPPAAL applied on a Danfoss EKC 204A temperature controller [Larsen et al., 2003].

The origin of this model is a case from a model-based test generation where UPPAAL is used to generate a test sequence. This model contains two parameters called P_1 and P_2 defined in equation 2.6.

$$(0,\, 0) \leq (P_1,\, P_2) \leq (40000,\, 40000)\ where\ P_1 \geq P_2 \tag{2.6}$$

These parameters are used to specify a time interval $[P_2,\, P_1]$ where an action should occur. The objective is to find all pairs of P_1 and P_2 where $((2.7) \wedge \neg\, (2.8))$ holds.

$$\exists\langle\rangle\ test.Done3\ \wedge (DripOff2Off\ \wedge Off2On) \tag{2.7}$$

$$\exists\langle\rangle\ test.Done3\ \wedge (\neg DripOff2Off\ \vee \neg Off2On) \tag{2.8}$$

The goal is to find the most optimal pair, where the time difference is minimal, and not just a success criterion, which would be a relatively easy task. However, to find the precise time bounds when an action should be exercised is a more difficult task. If a brute force technique were used, the total number of jobs can be calculated using equation 2.9.

$$\sum_{P_2=0}^{40000} \sum_{P_1=P_2}^{40000} 1 = 800.060.001 \tag{2.9}$$

When evaluating the results of equation 2.9 it can be concluded, that it would require monumental computing power to finish the verification of all possibilities of the model within a reasonable amount of time. Therefor a smarter approach that includes an optimization schema, in order to minimize the number of runs, is necessary.

2.4 General about Parameter Sweep Applications

This section provides use cases from real world scenarios where parameter sweeps has been applied. The use cases describes systems that all could benefit from distributing the workload.

2.4.1 Use-case: Functional Magnetic Resonance Imaging

Functional Magnetic Resonance Imaging (FMRI) is a method for observation of brain activity while processing physical or cognitive simulation [Smith et al., 2004]. Before brain activation maps can be generated, datasets containing series of the 3-dimensional MRI scan must be processed. The processing of the FMRI data is conducted using parametric mapping tools and software libraries, developed at Oxford Centre for Functional Magnetic Resonance Imaging of the Brain. These packages hides the complexity of the performed analysis, however the choice of selecting the right parameters is still an important factor in FMRI analysis. Default values are provided for the parameters, yet they are not necessarily appropriate in all cases.

The process of searching for optimal parameters in the parameter space is both a uniform and time consuming task. A sweep experiment that varies the delay parameters of a hemodynamic response function with ranges from 4 second to 8 seconds with a step of 0.25 second is applied to 22 different datasets whould produces 364 different jobs. These 364 jobs are part of a single experiment and therefore the experiment needs to be repeated. Having the experiment repeated 6 times with job running times varying from 30 to 140 minutes takes a total of 416.7 hours; the equivalent of 17 days.

These numbers provides convincing reasoning for automating the process, and distributing the workloads on clusters thereby reducing the search time for finding optimal parameters.

2.4.2 Use-case: Spatial Stochastic Birth-Death Process

The results from a Spatial Stochastic Birth-Death Process, which simulates the dynamics of the bacterium *mycoplasma gallisepticum* found in House Finch birds, is of great importance to ecologist in order to study it [Wang et al., 2009]. The simulation needs to be run over a parameter space which includes,

- b, the birth rate of finches
- d, the death rate of finches
- E, the temporal length of the simulation and
- X and Y, as the breadth and width of the simulated spatial domain.

These five parameters includes both integer and rational numbers. The execution time of a single run is proportional to $X \times Y \times E$. This means that simulations of longer temporal solutions and of larger simulated spatial domains takes longer time to run. For instance, running a single simulation with parameters $X = 32,\ Y = 32,\ E = 30$ can take five hours.

The runtime of these simulations has been reduced using a PSA and distibuted computing. A solution containing a Master-Slave architecture has been implemented to utilize ad-hoc network computers. The reseachers are planing to extend the solution to enable the use of clusters.

2.5 Existing Frameworks

2.5.1 AppLeS Parameter Sweep Template

The AppLeS Parameter Sweep Template (APST) is an application execution environment developed by Casanova and Berman [2003] that handles scheduling and deployment of large scale parameter sweep application on grid platforms. The architecture of APST allows for a range of grid accessible resources, such as storage and compute, to be accessed through deployed middleware services as depicted in figure 2.3.

For launching applications APST includes existing technologies such as the remote execution manager Globus GRAM [Alliance, 2009] and supports the security and authentication mechanisms used in SSH. Additionally the APST can start jobs using the existing full-featured batch systems like Portable Batch System (PBS), IBM Tivoli Workload Scheduler LoadLeveler [IBM] or Condor [The Condor Team], which is a specialized workload management system.

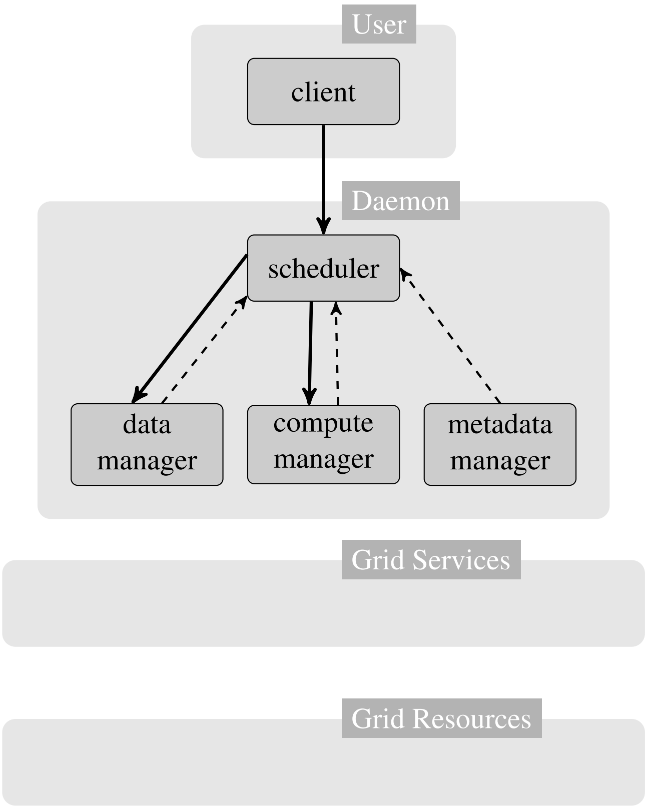


Figure 2.3: The architecture of AppLeS Parameter Sweep Template.

APST supports mechanism for transfer and storage of application data among storage resources. Depending on source and destination APST can use a range of different existing mechanisms, these includes:

SCP
SCP provides file transfer capabilities using SSH as a tunnel for providing encryption and authentication throughout the file transfer.

FTP
FTP is a network protocol for file transfer. It supports authentication yet users need to send their credentials using non-encrypted messages.

GASS [Bester et al., 1999]
Is a data movement and access service that, using URLs, defines a global namespace which allows applications to access remote files.

GridFTP
GridFTP has been developed as an extension of FTP to be used for grid based computing, in an effort to provide a more reliable and high performance file transfer.

SRB
The Storage Resource Broker (SRB) is a distributed file system that support the management of distributed data collections which includes transfer.

APST has implemented a client-server based architecture where the server, which acts as a service(*daemon*), is responsible for deploying and monitoring applications. This allows the user, through a client, to interact with the *daemon* to e.g. submit computation request or check application progress. APST uses an XML based encoding, which allows users to relatively easily control and direct grid resources to run desired applications. Resources can be added at any time during execution by a user sending XML based encoded descriptions to the *deamon*. An example of such a control sequence is written in listing 2.3.

```
<apst>
  <storage>
    <disk id="DISK1" datadir="/home/data">
      <gridftp server="storage.site1.edu" />
    </disk>
    <disk id="DISK2" datadir="/usr/home/data">
      <scp server="storage.site2.edu" />
    </disk>
  </storage>
  <compute>
    <host id="HOST1" disk="DISK1">
      <globus server="host.site1.edu" />
    </host>
    <host id="HOST2" disk="DISK2">
      <ssh server="host.site2.edu" processors="40" />
    </host>
  </compute>
</apst>
```

Listing 2.3: Example of APST control sequence

The listing 2.3 contains an example where the two storage- and two computing resources are attached to the *daemon*. This XML schema provides an intuitive and understandable syntax for both manually reading and writing the XML code. Once resources have been added, the users can create tasks that are going to be scheduled on the attached computing resources. A similar intuitive syntax for launching applications exists, and an example is shown in listing 2.4.

```
<tasks>
  <task executable="app" arguments="f1 g1" input="f1" output="g1"
      cost="1" />
  <task executable="app" arguments="f2 g2" input="f2" output="g2"
      cost="2" />
</tasks>
```

Listing 2.4: Example of APST control sequence

As shown in listing 2.4 the syntax for specifying executables, and their arguments, dependent input files, output files and the computational complexity in relation to other tasks is both easy and most intuitive.

2.5.2 Nimrod/G

This section is based on Buyya et al. [2000] and Abramson et al. [2002].

Nimrod is an application designed for easily creating and distributing parameter sweeps of existing applications, with as little programming involvement from the user as possible.

In order to create a parameter sweep with Nimrod, the user only needs to write a *plan* file, such as the one shown in Listing 2.5 on page 13, which describes the following

- the parameters and their associated values
- pre- and postprocessing for each individual job, such as file stage-in and result retrieval
- pre- and postprocessing for the entire parameter sweep
- the actual job excecution

Nimrod is then able to process the plan file, and automatically generate the jobs from the parameters, which can be executed on statically assigned remote hosts, or, with the /G edition, on more dynamic grid resources, by use of the Globus toolkit.

As can be seen in Figure 2.4 on the following page, Nimrod/G consists of several parts. Most visible to the user is the client, of which multiple instances can be run, on several machines, monitoring the same or different jobs.

At the heart of the system is the Parametric engine, which is responsible for the creation of jobs based on the parameters in the plan file, and, using information from the schedule advisor, directing the parameter sweep.

The Parametric engine is also responsible for storing the state of the experiment in the Persistent information database, thus ensuring that the parameter sweep will not have to start all over, should the system fail.

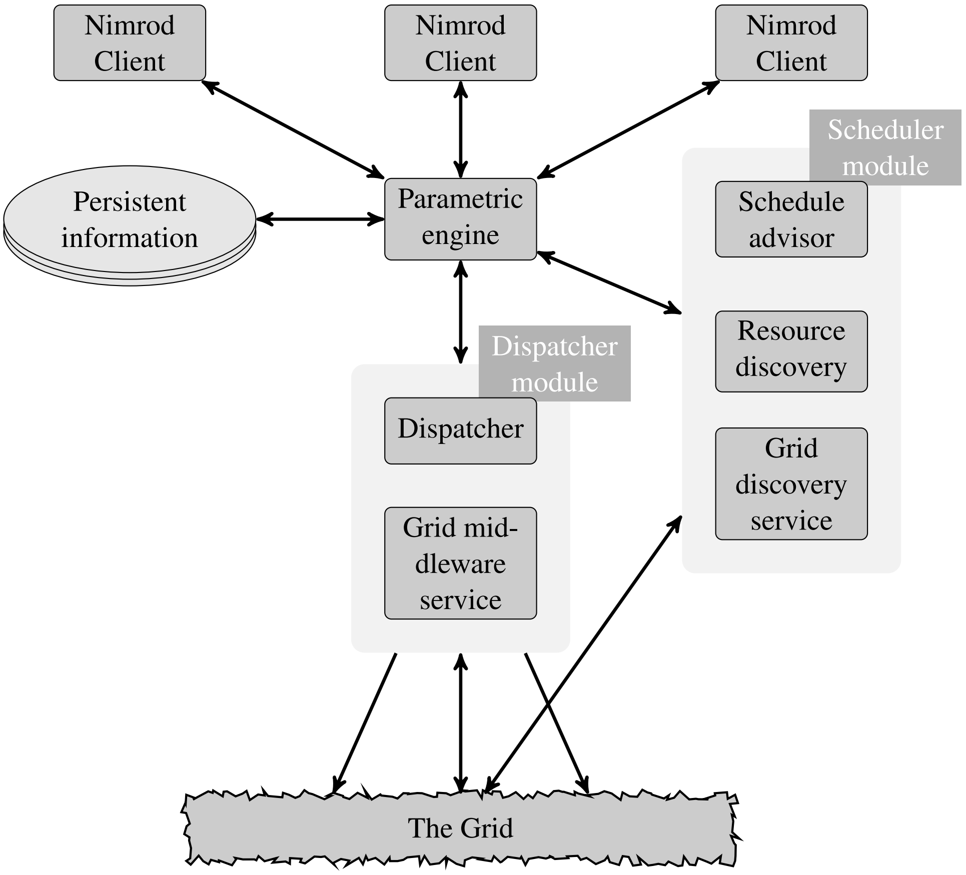


Figure 2.4: Nimrod system architecture

The Dispatcher will execute the jobs specified by the Parametric engine, on the resource specified. It does this by starting a job-wrapper on the remote machine, which handles the staging of data, and all other tasks done at the remote machine.

To support the Parametric engine is the scheduler module, which selects resources to use, and assigns jobs. These selections are reported back to the Parametric engine.

Nimrod/G is built for scheduling on the grid, and as such, the Scheduler module is able to schedule tasks according to deadlines, and select resources based on cost specifications (i.e. how much the user is willing to spend) Scheduling according to a deadline is necessary, since it is not as easy to predict the running time of the parameter sweep when executed on a grid, compared to running the parameter sweep on a cluster with a limited number of users, and static resources.

As can be seen in the Listing 2.5, the creation of parameter sweeps is quite simple and straightforward, and the Nimrod toolkit even comes with a GUI for creating the plan file.

2.5.3 Nimrod/O

Nimrod/O is a variation of Nimrod/G with the ability to apply parameter constraints, and optimize the scheduling based on the results of previous results. This section is based on [Peachey, 2005].

```
parameter P1 integer range from  0 to 40000 step 1;
parameter P2 integer range from  0 to 40000 step 1;

task nodestart
  copy ekc2params.sub node:./model.sub
  copy ekc2params.q   node:./query.sub
  copy $OS/verifyta    node:./verifyta
endtask

task main
  node:substitute model.sub $jobname.xml
  node:execute ./verifyta $jobname.xml query.q
endtask

task nodefinish
  copy node:output ./output.$jobname
endtask
```

Listing 2.5: A Nimrod plan file for the EKC 204A Temperature Controller

However, the optimization mechanism is quite limited, as it will only use the first value in the output file of a job, and as such, optimizing on several variables of both boolean and integer types can be quite difficult. An example of the fixes necessary to optimize on the three boolean results of the EKC 204A temperature controller model can be seen in listing 2.6 on the next page. As can be seen in line 16, an extra step is necessary, in order to map the output of verifyta to a single variable. In this case, all three booleans are represented such that true=0 and false=1, as the simplex algorithm minimizes, and all are then concatenated to a single value.

2.6 Summary

The use of model checkers to validate criteria, is an essential idea which has found use both in academia and industry wide. The task of model checking is however still time consuming for major tasks, and when models are parametrized to be run thousands or even millions of times, a new schema that is able to handle the automated process of parallel execution of these tasks is needed. There exists solutions to cope with this need of an automated process and we have studied some of the more serious of them.

In the study of different frameworks we have found that the Nimrod application provides parameter sweep functionality that could be usable in distributing model checking. We have also found that the /O edition of Nimrod provides functionality in terms of an optimization scheme that would limit the number of runs needed. However, we have discovered that in order to calculate optimized input parameters, a mapping as in Equation 2.10, of the multiple output values to a single one is required [Peachey, 2005] [Abramson et al., 2006].

$$m : y_0 \times y_1 \times \cdots \times y_n \mapsto y_{single} \tag{2.10}$$

This is because the build-in optimizer in Nimrod/O utilizes a single value to advise the

```
parameter P1 integer range from  0 to 40000 step 1;
parameter P2 integer range from  0 to 40000 step 1;

hard constraint {P1 >= P2}

task nodestart
  copy ekc2params.sub node:./model.sub
  copy ekc2params.q   node:./query.sub
  copy $OS/verifyta  node:./verifyta
  copy $OS/map         node:./map
endtask

task main
  node:substitute model.sub $jobname.xml
  node:execute ./verifyta $jobname.xml query.q > out.txt
  node:execute ./map out.txt
endtask

task nodefinish
  copy node:output ./output.$jobname
endtask

method simplex
  starts 2 named "Simplex"
    starting points random
    tolerance 0.000
  endstarts
endmethod
```

Listing 2.6: A Nimrod plan file for the EKC 204A Temperature Controller

build-in optimization processes. This prevents a direct link between multiple model checker output values and the corresponding input parameters.

APST is another candidate for running PSAs, however it fares even worse than Nimrod, as it does not have any optimization at all, it merely schedules according to available resources. As such, to use it effectively for PSAs, a complete optimization system must be hooked into the existing APST platform.

Should any of the above frameworks be used as a platform for launching PSAs, it would be the Nimrod/O framework. However, much work is required, both to define the mapping of values to one value, and to simplify the task of creating the PSAs for the user, and as such, it would likely be both easier and more effective to create a new application specifically designed for executing PSAs on the grid.

3 Problem statement

Based on the descriptions and discussions in the previous chapter, it seems obvious to examine how to develop a framework for running UPPAAL PSAs. This motivation leads to the following problem statement:

How can a PSA service, that distributes the workload of verifying a parametrized UPPAAL *model on multiple resources, be realized?*

We have chosen to name our project UPPAAL PARMOS (UPPAAL PARametric MOdelchecking Service). UPPAAL PARMOS should provide an easy way of running distributed UPPAAL PSAs.

3.1 Design goals

Based on our analysis, we state the following goals for the development of UPPAAL PARMOS.

Scalability: As the user should be able to utilize multiple resources, it is important that UPPAAL PARMOS scales well, and ensure a fair and balanced workload distribution throughout all available resources.

Performance: Overhead produced by data traffic should be minimized. UPPAAL PARMOS should be able to handle millions of jobs, and any overhead that each job produces will therefore quickly sum up.

Fault tolerance: Given that PSAs usually consist of a large number of jobs to be run, it is important that execution should be resumable after a crash or a planned outage. Otherwise a user might end up after a crash, being forced to restart a very long running task, with millions of jobs, even though the vast majority of the jobs were completed before the crash.

Usability: Given that users not necessarily are familiar with PSAs, the user interface should be intuitively designed, in order to minimize the prerequisite for using the system.

Connectivity: In order to establish communication between UPPAAL PARMOS, users and distributed resources a common standardized communication channel should be utilized.

Extensibility: UPPAAL PARMOS should provide programming interfaces for easy integration of new functionality, e.g. algorithms.

4 Basic concepts in distributed systems

As a foundation for designing our own solution, this chapter provides descriptions of basic concepts and examine some of the problems one might face.

4.1 Resource management: TORQUE

As the cluster put at out disposal for testing, is running the Terascale Open-Source Resource and QUEue Manager (TORQUE) middleware, we here presents an overview of it.

TORQUE is a resource manager based on PBS, and as such has many similarities with PBS. Being a resource manager, it does not perform the actual scheduling, instead utilizing a number of different schedulers, such as the Maui/Moab schedulers or the simple FIFO scheduler shipped with TORQUE, which it provide with information about jobs and available resources.

In order to execute jobs via TORQUE, the executable(s) must be available on all nodes where the job is to be executed. The command for executing jobs is `qsub [options] job` where the `job` is the name of an binary executable or an executable script file.

```
1   #!/bin/bash
2   #PBS -q dque
3   #PBS -l nodes=4
4   #PBS -o file.out
5   #PBS -e file.err
6   #PBS -W stagein=./executable@mother:$HOME/execdir/executable
7   #PBS -W stageout=./file.out@mother:$HOME/outdir/file.out,
        ./file.err@mother:$HOME/outdir/file.err
8   ### once a non-PBS-argument/non-comment line is encountered,
        PBS reads no further
9   ./executable | grep -e 'result:'
10  #PBS -l nodes=128 this will be ignored
```

Listing 4.1: Example TORQUE job execution script

If the job is a script file, it is possible to have the options inside it, embedded as comments. Listings 4.1 shows an example of this, where line 2 defines the queue to use (in this case *dque*), line 3 defines the resources required (here 4 nodes of unspecified power), and line 4 and 5 defines the files where to stdout and stderr should be written out. The lines 6 and 7 stage in the files needed for execution and stages out the redirected stdout and stderr. Finally in 9, the executable and run.

In this case, the *file.out* will not contain the output of *executable*, rather it would contain the output of *grep*, as this is what would have been printed to stdout, had it not been run via TORQUE.

4.2 Web Services

Web Services (WSs) are commonly used as an internet based API through which client computers can interact with a service. Often the interaction is in the form of a client using Remote Procedure Calls (RPCs) to invoke operations on a service.

The World Wide Web Consortium (W3C) defines a WS as following:

> A WS is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Services Description Language (WSDL)). Other systems interact with the WS in a manner prescribed by its description using Simple Object Access Protocol (SOAP)-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. [W3C, 2004]

In order to communicate with a web service some predefined communicate scheme is needed. The most common implementation of such a communication scheme is an XML-based SOAP message, yet also an alternative Representational State Transfer (REST) approach is widely used [Coulouris et al., 2005]. SOAP implements both asynchronous interaction and a client-server communication, using the HTTP POST method as request, and the response as reply.

HTTP, XML, SOAP and WSDL are however not the only technologies used for WSs. A number of protocols exists, which eases the implementation and utilization of WSs. For example SMTP, FTP, Java Message Service (JMS) and Internet Inter-Orb Protocol (IIOP) are alternatives to HTTP for transporting messages.

4.2.1 Examples of use

WSs find various uses, both public on the Internet, and as private WSs within and between corporations. Many companies provide web services as an interface to their resources, for others to access in their own programs. Examples WSs include SMS services, where customers may use the providers WS in their own systems, e.g. for bulk sending[Clickatell, 2010], online payment providers [PayPal, 2010], upload and download services to access file resources [Flickr, 2010], and also social media providers, like Facebook and Twitter, use web services to enable third party developers to create applications using their resources [Facebook, 2010] [Twitter, 2010].

4.2.2 The elements that build a Web Service

Figure 4.1 on the next page shows the WS Protocol Stack. The four layers represent how a WS suite is built with elements from each layer, each building upon the foundation of the layers below it. For example one might create a WS by using HTTP as transportation for SOAP messages, having the service interface defined by WSDL, and announced by an Universal Description, Discovery and Integration (UDDI) service.

Layer	**Technologies**
4. Service Discovery	UDDI
3. Service Description	WSDL
2. Messaging	SOAP (+ Extensions), REST
1. Transport	HTTP, SMTP, FTP, JMS, IIOP

Figure 4.1: Web Service Protocol Stack

WSs usually communicate using standard transport layers, also used by other internet media such as the ones shown in the Transport layer of Figure 4.1. Common for the remaining three layers is however, that they are WS specific, and utilize some form of XML to represent and serialize information.

Many programming languages have APIs for easily creating and using WSs. PHP for example has extensions that provides the means to both create a server and a client application using SOAP as well as generating WSDL documents [The PHP Group, 2010], and ASP.NET provides framework for the same [Microsoft .NET Framework SDK QuickStart Tutorials, 2010].

When creating a WS, discovery will usually not receive great attention, as often it is enough to distribute a description document internally in a corporation, or to announce it on a website for public access. A directory service can however be very useful if a large corporation has a lot of internal web services, or for gathering descriptions for similar web services in a specific line of business. The latter could for example be a directory for travel agencies who are to access booking information for a number of hotels, possibly belonging to different chains.

The following section will describe the structure of SOAP messages, also touching the issue of security in WSs explaining how to use SOAP Extensions to implement encryption. Later the structure of WSDL documents will be described, and lastly we will briefly describe REST as an alternative to SOAP.

SOAP

As of version 1.2, the Simple Object Access Protocol (SOAP) defines how to represent the contents of a message in XML, how to use messages to conduct request-reply patterns, how receivers should process the XML messages, and how to use HTTP or SMTP for communication [Coulouris et al., 2005] [W3C, 2010].

The structure of a SOAP message XML document, is depicted in Figure 4.2 on the following page. A SOAP message consists of a <*Header*> and a <*Body*> element, contained in an <*Envelope*> element at the root of the document. The <*Header*> is however optional, and can be left out. Each element also specify addresses to the XML namespaces they use. The

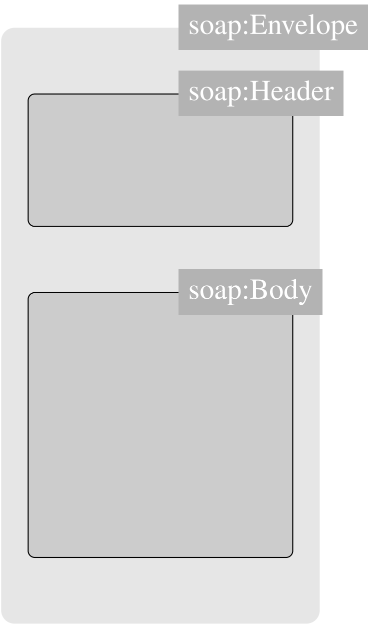


Figure 4.2: Structure of a SOAP message

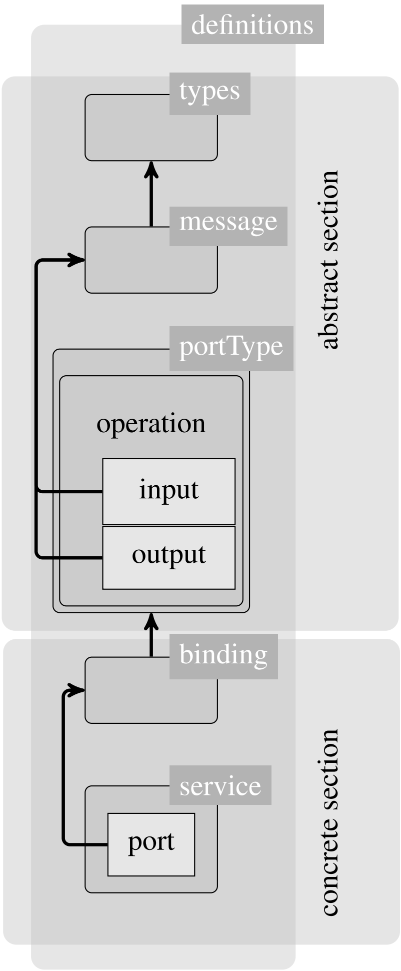


Figure 4.3: Structure of a WSDL document

Envelope usually defines `http://schemas.xmlsoap.org/soap/envelope/`, which is a standard location for soap SOAP XML namespaces.

Sending SOAP messages in plain XML poses a security risk for messages containing sensitive data. Therefore appropriate security measurements should be implemented for relevant applications, which is done via SOAP Extensions. SOAP Extensions allows one to define operations that should be applied at four stages in the life of a SOAP message, namely before serializing and after serializing the message at the sender side, and before de-serializing and after de-serializing the message at the receivers side. The extensions that should be applied for a given message is noted in the header of the message, to tell the receiver application which extensions to apply. Control over these points allows the programmer to implement various kinds of extensions, such as logging of messages, addition of meta information, and security in the form of encryption and/or key signatures.

WSDL

A Web Services Description Language (WSDL) document is a XML document type, which is used for describing a WS with regards to locating and functionality. A WSDL document is usually used to define a SOAP WS, but may as well describe other types WSs with regards to the format of messages and employed network protocols. This section is based on [Christensen et al., 2001].

A WSDL 1.1 document consists of a <*definitions*> element at the root, containing five main element types, namely <*service*>, <*binding*>, <*portType*>, <*message*> and <*types*>. The first two elements, <*service*> and <*binding*>, define concrete service and protocol specific properties, while the later three, <*portType*>, <*message*> and <*types*>, define abstract properties in a uniform way. Figure 4.3 on the preceding page shows an overwiew of the structure a WSDL document.

<service> Each <*service*> defines a set of <*port*>s which make up a WS. Every <*port*> each define an endpoint address and points to a <*binding*> for that address. The <*port*>s may bind to different addresses, and/or protocols, but all <*port*>s belonging to the same <*service*> must provide equivalent semantic behaviour.

<binding> Maps abstract content into a concrete format, by defines with witch protocol and in which data format, the <*operation*>s and <*message*>s of a specific <*portType*> is to be conducted. Only one protocol is defined per <*binding*>, but there may be several <*binding*>s for any **portType**.

<portType> Defines a set of <*operation*>s that can be performed by a WS. Each <*operation*> refers to which <*input*> and/or <*output*> <*message*>s it makes use of.

<message> Defines which logical data elements are used by <*operation*>s. All the data elements must be of a defined <*type*>.

<types> Defines the data types used in the exchanged messages, using XML Schema Document (XSD). This ensures that different computing platforms agree on how to read the data, and thus avoids errors that could otherwise occur from systems not agreeing on type specifications.

REST

Representational State Transfer (REST) is an alternative to SOAP. The concept is not specific to WSs, but more a concept for software architecture over distributed media in general. In REST, systems communicate using XML (sometimes other serialisations are used), via the HTTP protocol, as with SOAP and WSDL, but with REST the definitions and XML wrapping used with SOAP and WSDL is unnecessary. REST sees each URL as a different object, and uses existing HTTP methods to retrieve info about, and manipulate the object.

5 Design

This section provides an overview of the chosen design for UPPAAL PARMOS. These goals arise from the goals described in Section 3.1 on page 15.

5.1 Decisions

This section describes design decisions regarding UPPAAL PARMOS.

5.1.1 Web service

In order to comply with our goal of connectivity, we have decided that the primary way of interacting with our software system, should be through a WS. By using a WS as an API towards our software system, we exclude no one from utilizing it, no matter what OS or programming language they may prefer, as all communication is conducted over a standard type network connection. The WS will prepare submitted PSA tasks for processing, and leave the processing of the tasks to a service that takes care of verifying the task. The status and results of the tasks can then be accessed through the WS.

5.1.2 Data storage

In order to make our system fault tolerant, we have decided that all data should be stored in a database using reliable transactions (the properties of Atomicity, Consistency, Isolation, Durability (ACID)). By letting this database be responsible for most of the systems idea of state, the process of recovering after a crash will be much easier. This is due to the nature of ACID-compliant databases, ensuring transaction-safety. A recovering system will therefore find a consistent state in the database, and thus the system can continue from the state it was in before the crash, given of course that the database has not been corrupted or lost, e.g. from the break down of a storage device.

5.1.3 PSA service

We have decided to build our PSA service using a scheduler, and a number of dispatcher types, all having a shared interface through which they can be controlled by the scheduler, but each able to utilize a different kind of resource. There are two reasons for making a scheduler. First, the scheduler must provide scalability, as it should be able to balance the workload distribution throughout all available resources. Second, for performance reasons, the scheduler should be extensible with scheduling algorithms that strive to decrease the time the user has to wait for usable results from a submitted task. This performance reason is elaborated further in Section 5.2.1. The PSA service should also be extensible, by adding new dispatcher types. The dispatchers should be made so that reusing common interface functionality is easy. The extensibility of the scheduler and the dispatcher is elaborated further on in Section 5.2.2.

The decision for a design with a scheduler and a number of dispatchers has been greatly inspired by Casanova and Berman [2003] and Buyya et al. [2000].

5.2 Issues

Following the design goals of Section 3.1 on page 15, some goals are (in principle) straightforward to resolve while others require further discussions. This section discusses the open issues, and elaborates on/relates to what is written about them in Section 5.1 on the preceding page.

5.2.1 Performance

When managing PSA tasks, where the jobs are distributed among a number of resources, it is crucial to minimize the amount of data that needs to be transferred for each task. When running a large scale PSAs with hundreds of thousands, or even millions, of jobs, each transferred byte for each job matters, in order to minimize the total run-time overhead.

Also there should be no unnecessary queries to the database. Network traffic may not be an issue if the database is set up on the same host as the rest of the system, since disk IO will then be the tightest bottleneck. Still, the hard disk is usually the slowest form of IO on modern computer systems, and may pose a serious bottle neck for tasks with a very large amount of jobs with a relatively low running time. However, if the database is hosted on another machine, queries will also add to the amount of traffic that needs to go through the network interface. These will however not be as much of an issue on tasks with long running jobs, as the verification itself will become the bottleneck

As stated in 5.1, the scheduler should be able to provide usable results as quickly as possible. Therefore a variety of algorithms should be available for the user to choose between when submitting a task. Ideally all the jobs of a task should be run for complete results, but in order to provide fast results, an algorithm could speed up the process, by looking at the results of already completed jobs, in order to choose which jobs to prioritize higher.

5.2.2 Extensibility

UPPAAL PARMOS must be easy to extend with new dispatcher types, providing the ability to utilize new resource types, and new scheduling algorithms to reduce the time for producing usable results. The task of extending UPPAAL PARMOS should be focused on defining specifics, rather than ensuring the right interface. Therefore a common base functionality should be provided and shared between scheduling algorithms and dispatcher types respectably. Shared functionality, e.g. for accessing data in the database, should be delegated to shared functions, easily reused in new instances. An extra gain of this is, that changing shared behaviour is easier, as it can be done for all scheduling algorithms or dispatcher types at once.

5.3 Architecture

In light of the previous Sections, we have designed UPPAAL PARMOS to contain the components portal, web service, database, scheduler and dispatchers. These are categorized into the three categories Interface, Database and PSA Service, the database component being a category in it self. Figure 5.1 gives an overview of the architecture of UPPAAL PARMOS.

The purpose of the Interface category is to provide access to the system from the outside world. The purpose of the Database is to provide reliable storage for the systems state as described in 5.1. The purpose of the PSA Service category is to process tasks using scheduling algorithms to producing usable results as fast as possible. The following sections will describe the role and purpose of each category and component further.

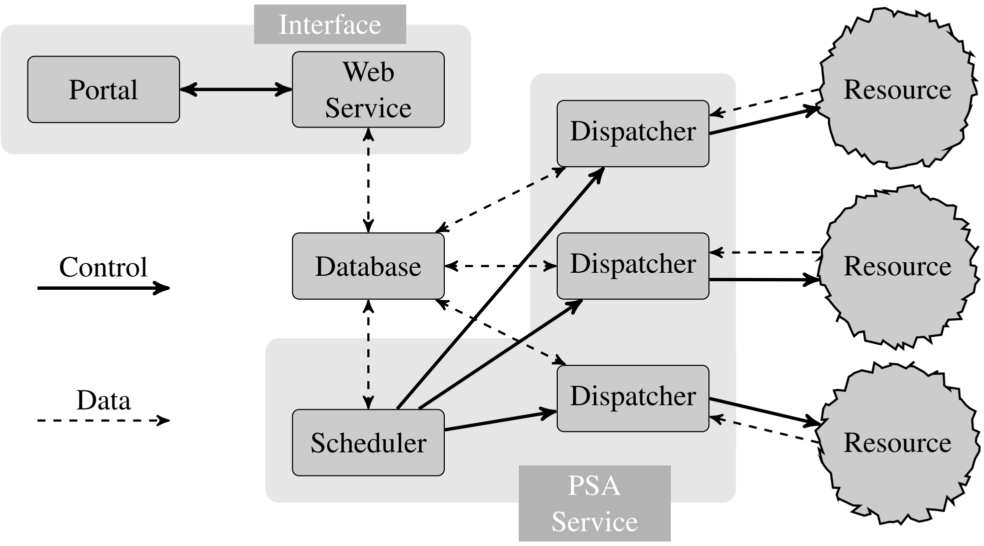


Figure 5.1: UPPAAL PARMOS architecture

5.3.1 Interface

The Interface category contains the components, that provides communication between the rest of the system and the outside world.

Portal

The portal is the GUI component of our software. It communicates with the PSA Service solely through the WS, and it is intended as a standard way of allowing the user to submit tasks in form of a model, a query and a parameter file to the system, cancel tasks, and retrieve information on the progress and results of already submitted tasks. The rest of the system is independent of the portal, and therefore, once the task has been submitted, the portal can be closed without harm to the running task.

Users may create their own custom interface component, as long as this communicates with the back end through the WS.

Web service

The WS is the external interface of UPPAAL PARMOS, and provides an API for controlling the PSA Service, and managing resources available to the PSA Service.

A user may submit tasks in the form of a model, a query and a parameter file, to the PSA Service. When a task is submitted, the WS parse and verify the structure of these files, responding with an error back to the user if they do not pass the check. If the files pass, the parameter file will be dissected, and all data is stored appropriately in the database (see Section 5.3.3 on the next page for more details on where the data is placed). After this, the scheduler will find the task data, and start working on the task.

5.3.2 PSA Service

The PSA Service is a service that consists of a scheduler and a number of dispatchers.

Dispatchers

A dispatcher is a job processing slave of the scheduler. If no dispatchers has been initiated successfully, the system will not be able to compute any jobs. In Figure 5.1, three dispatchers is shown to be controlled by the scheduler, but there may be any number of dispatchers initiated by the scheduler, depending on which computing resources are available to the system. All dispatchers have an identical interface, that enables the scheduler to start, stop and assign jobs to them, as well as retrieve information on the dispatchers current status. Several types of dispatchers can be written, in order to enable the scheduler to utilize more kinds of resources, as long as they provide the correct interface to the scheduler. The same type of dispatcher may be initiated for different resources, using the same middleware.

Figure 5.2 on the facing page shows the flow of a dispatcher. Dispatchers are initiated by the scheduler, either when the scheduler it self initiates, or at some point after it has been added to the database, and the running scheduler checks for new dispatchers. A running dispatcher is assigned jobs by the scheduler, periodically checks whether any of the jobs it has dispatched have finished. When a job has finished, the dispatcher retrieves the result, and updates the jobs record in the database with the result.

Scheduler

The scheduler is the key component in UPPAAL PARMOS. The flow of the scheduler is shown in Figure 5.3 on page 28. When the scheduler initiates, it will also create dispatchers for the resources found in the database, keeping a handle to each of them in order to be able to start, stop and assign jobs to them.

While running, the scheduler is responsible for starting, stopping and assigning jobs to the

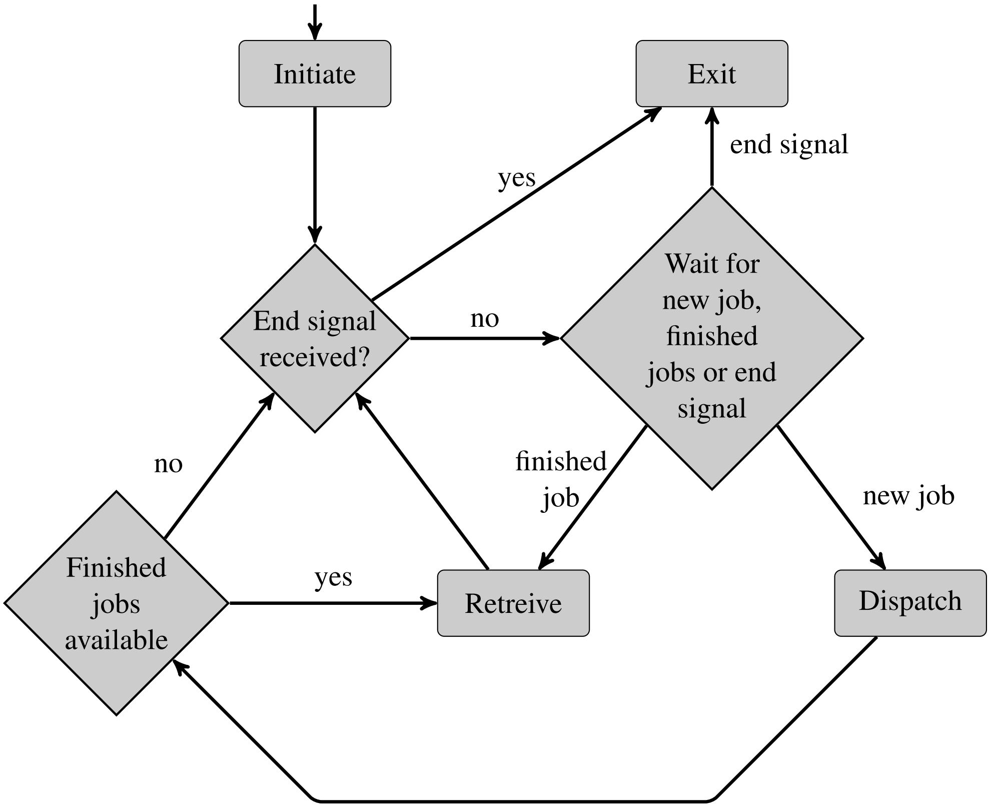


Figure 5.2: Dispatcher flow

dispatchers when appropriate, and for scheduling the jobs of each task according to a per task, user chosen algorithm. The main goal of the scheduling is to minimize the time for getting usable results of a task.

When the scheduler decides to run a job, it first creates the job and saves it in the database. Next it chooses a dispatcher, and tell this dispatcher to process that specific job. Periodically, depending on the chosen algorithm, the scheduler will check the database for finished jobs, and extract their results and use them to calculate the next jobs to be processed.

5.3.3 Database

All the other components, except for the portal, integrate through the database. For the reasons explained in Section 5.1, all data storage, and most communication happen through getting data from and updating data in the database.

The database consists of six tables storing various information about the dispatchers and the tasks and jobs submitted via the web service. The names of the tables are *dispatchers*, *tasks*, *taskparameters*, *conditions*, *goals* and *jobs*, and the following paragraphs will describe the usage of each table.

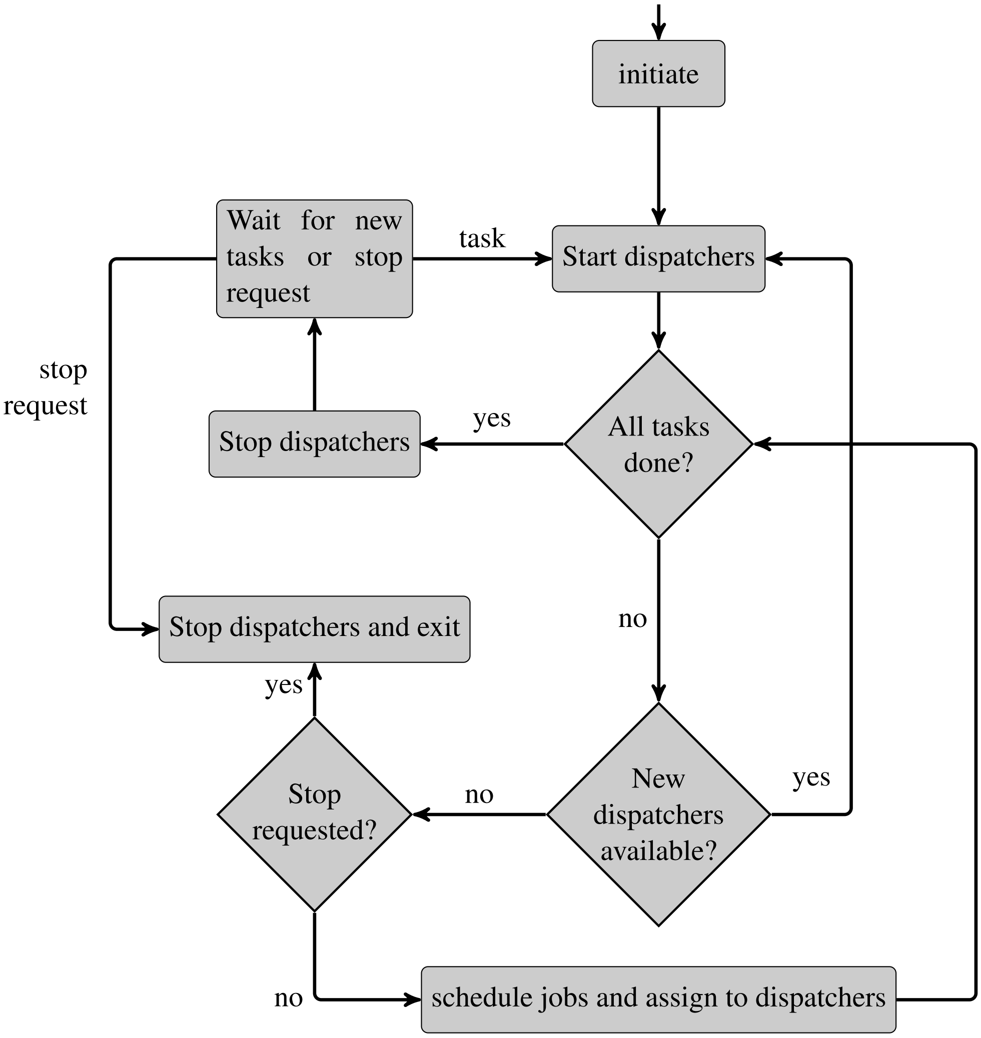

initiate
Wait for new tasks or stop request
task
Start dispatchers
stop request
Stop dispatchers
yes
All tasks done?
no
Stop dispatchers and exit
yes
Stop requested?
no
New dispatchers available?
yes
no
schedule jobs and assign to dispatchers

Figure 5.3: Scheduler flow

dispatchers Contains records that describes the resources available to create dispatchers for. Each record store the type, endpoint address and login credentials for a dispatcher, as well as any additional options needed to use the dispatchers resource.

tasks Contains a record for each task the scheduler should work on. Each record store the tasks model and query files along with task configuration parameters, and timestamps for start and finish of the task.

taskparameters Contains a record for each parameter in every task. Each record store the upper and lower bound for the parameter, and how much it should increment with each iteration in a brute force search. These are defined by the user in the parameter file, and placed here by the WS after parsing the parameter file. The increment value serves as a hint from the user to the algorithm doing searches over the parameter space.

conditions Contains records that define preconditions. These precondition records are created by the WS parsing the parameter file, and must be fulfilled by the parameters of each job, generated in relation with a given task, in order to allow it to run at all. Each record stores a reference to which task it belongs, along with a specified precondition that decide whether to run a job or not.

goals Contains records that define success criteria for jobs, depending on which task they belong to. Again these come from the WS parsing the parameter file, and each record stores a reference to which task it belongs, and along with this a criterion that must be fulfilled to declare the job result successful.

jobs Contains record for the jobs generated by the scheduler. Each record starts out with just storing a reference to the task which it belongs to, and the parameters it is to be run with in the model of the task. Later when a dispatcher is assigned the job, a reference to that dispatcher is added. lastly, when the job has finished, the result of the processed job is added.

Implementation 6

In order to test our design ideas, a prototype has been developed. The implementation of UPPAAL PARMOS is accomplished in four different modules: The portal, the WS, the database and the PSA Service. In the rest of this Chapter, we will first, in Section 6.1, describe implementations decisions we have made, and thereafter we will dedicate a section to describe each module, except for the database, as we use a standard solution. Details for this will be described in Section 6.1.2.

6.1 Decisions

6.1.1 Platform

In order to minimize the development time and produce a widely compatible prototype we have chosen to use the Mono software platform [Novell editors]. Mono is a free and open source project that supports the implementation of an Ecma international standard compliant .NET-compatible set of tools. These include a compiler for the C# programming language and Common Language Runtime for compilation and execution of applications and services. The mono project also comes with a Base Class Library and a Mono Class Library that provides extensive sets of classes for rapid development of applications and services. The Mono platform is cross-platform and can be run on the most used operating systems including Linux, BSD, Mac OS X, Windows, Solaris, Unix and even game console systems like the PlayStation 3, Nintendo Wii and Microsoft Xbox 360.

Both the portal and the web service are hosted on a Internet connected server that runs a Linux distribution. The web server software is Apache, and the module *Mod_Mono*[mod] has been installed in order to provide ASP.NET support through Mono, which the web service and our default portal software requires.

6.1.2 Database

The separation of the WS and the PSA Service requires a database, as described in 5.1.2 on page 23, and we have chosen to use MySQL [MySQL, a] for that. MySQL is an open source relational database that provides a fully-featured management system. MySQL complies with the ACID properties when using the InnoDB engine [MySQL, c].

The people behind MySQL, provides a connector called *Connector/Net*, which is a fully-managed ADO.NET driver for MySQL [MySQL, b]. It allows for connectivity to the database from any part of UPPAAL PARMOS.

Using a relational database that supports queries using the SQL language allows for easy storing and retrieving of data and allows for a flexible design where data redundancy can be minimized. This is important as we store information about each job, and we have in

Section 2.3.2 on page 7 shown that a full parameter sweeps could become very large with regard to the number of jobs, hence the data storage size for each job should be minimal. A complete description of the database schemas can be seen in Appendix A.2 on page 53.

6.1.3 Tenets

UPPAAL PARMOS has been designed and implemented with disjointment and extensibility in mind. To realize this in the PSA Service, we have decided to follow a few tenets in the implementation of this module. These tenets help us avoid certain kinds of problems as described below.

Database connections

For performance reasons, we wish to avoid dynamic creation of database connections whenever one is needed. Therefore each thread that needs to query the database should maintain its own database connection. Each thread is then responsible for providing this connection, whenever it is needed in that threads context. Thus a thread must also supply a reference to its own connection, when calling a method on an object, where a connection is needed.

Base extensible classes

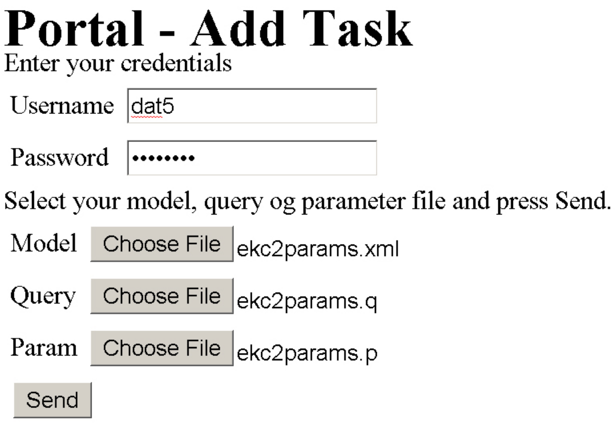
To comply with our extensibility goals (see Sections 3.1, 5.1.3 and 5.2.2), base classes for scheduling algorithms and dispatcher types have been implemented. These base classes must provide all the non-specific shared functionality between all scheduling algorithms and dispatcher types respectively.

Common for both is that they should provide methods that enable them to be controlled by the scheduler. This is especially relevant for dispatcher types, as they will run in their own thread, while scheduling algorithms are merely invoked by the scheduler.

6.2 Portal

The portal allows the user to submit parameter sweep tasks to the PSA and view information regarding submitted tasks. The portal retrieves and transmits all its information through a web service that acts as an API for the PSA. The communication is exchanged using SOAP and transmitted through the HTTP communication protocol. This decoupled structure allows users to create their own customized portal using the web service as an API.

ASP.NET, which is a web application development framework, and the C# programming language have been chosen as development tools for the portal. The choice of using e.g. ASP.NET for developing the portal is also made based on the large number of libraries that are available. A library that is applicable for the portal is the System.Web.Services. It consists of classes that allow the exchange of messages through standardized protocols and automation tools for generation of web service proxies using WSDL.



Portal - Add Task

Enter your credentials

Username: dat5

Password: ••••••••

Select your model, query og parameter file and press Send.

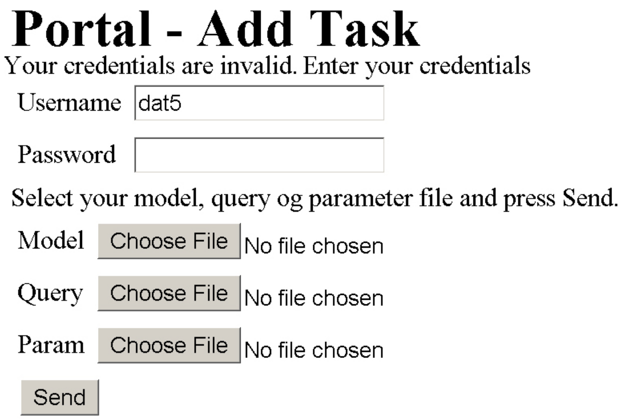
Model: ekc2params.xml

Query: ekc2params.q

Param: ekc2params.p

Figure 6.1: Graphical layout of the portals add-task-interface

Figure 6.1 depicts the graphical layout of the portals add-task-interface. This interface provides the basic yet sufficient functionality to add tasks to a PSA service. The user is required to enter valid credentials before a task ticket is issued i.e. without valid credentials the user is not able to add a task as depicted in Figure 6.2.



Portal - Add Task

Your credentials are invalid. Enter your credentials

Username: dat5

Password:

Select your model, query og parameter file and press Send.

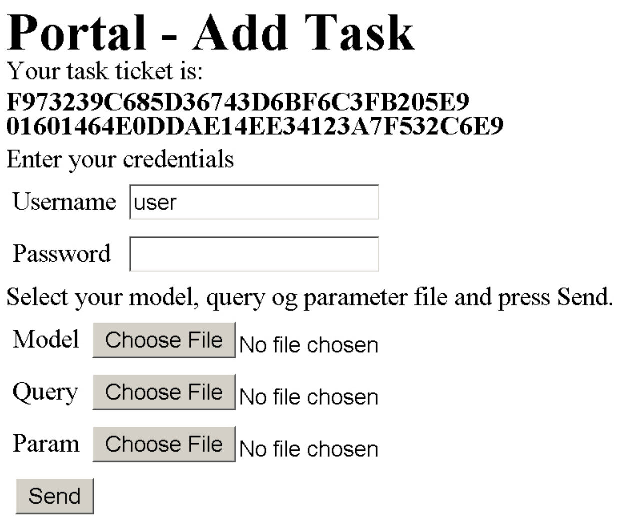
Model: No file chosen

Query: No file chosen

Param: No file chosen

Figure 6.2: Graphical layout of the portals add-task-interface, with error message.

Once the user has provided both valid credentials and the three required files a task ticket ID, as in Figure 6.3, is displayed. This ticket ID is used as a unique identifier that can be shared with other users. This allows the user to share task and appurtenant job information without disclosing their credentials.



Portal - Add Task

Your task ticket is:

F973239C685D36743D6BF6C3FB205E9 01601464E0DDAE14EE34123A7F532C6E9

Enter your credentials

Username: user

Password:

Select your model, query og parameter file and press Send.

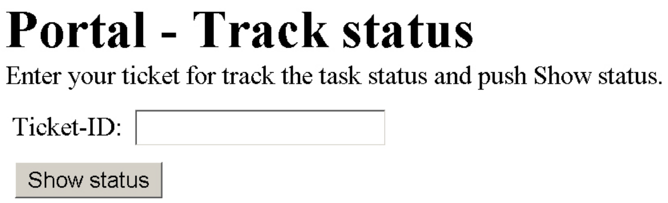
Model: No file chosen

Query: No file chosen

Param: No file chosen

Figure 6.3: Graphical layout of the portals add-task-interface, with success message.

After a task have been added to the PSA service, the user can utilize the information retrieving functions provided by the web service. To demonstrate this, a basic feature, depicted in Figure 6.4, of displaying the different states of jobs belonging to a task has been implemented.



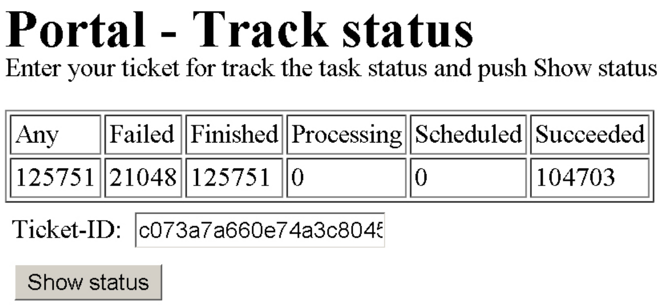
Portal - Track status

Enter your ticket for track the task status and push Show status.

Ticket-ID:

Figure 6.4: Graphical layout of the portals data-retrieving-interface

Once a valid task ticket ID has been entered the informaiton is retrieved as depicted in Figure 6.5.



Portal - Track status

Enter your ticket for track the task status and push Show status.

Any	Failed	Finished	Processing	Scheduled	Succeeded
125751	21048	125751	0	0	104703

Ticket-ID: c073a7a660e74a3c804

Figure 6.5: Graphical layout of the portals data-retrieving-interface, with information.

6.3 Web service

As mentioned in 6.2 on page 32 the web service serves as an API for front ends to the PSA. This allows the users to develop their own custom front end implementations fitting their particular needs. The web service is developed using the Mono software platform and coded in C#, which allows for the automatic generation of all SOAP and WSDL messages, hence focus could be placed on providing functionality.

To ease the process of specifying the task parameter space, we have chosen to implement a syntax which allows the user to specify the parameter space settings in a single file, just like the query and model file. Listing 6.1 illustrates some of the available commands.

```
1  p1 = {5:30, 1};
2  p2 = {0:100, 10};
3
4  # {p1 <= p2};
5  # {p1 != 20};
6
7  ? q1;
8  ? !q2;
```

Listing 6.1: Parameter specification in the parameter file

Line 1 in Listing 6.1 illustrates the syntax for specifying parameter p1 having an interval from 5 to 30 stepping 1, and line 2 similarily illustrates the syntax for specifying parameter p2 having an interval from 0 to 100 stepping 10.

It is also possible to specify that the scheduler must skip a value when certain conditions is not satisfied. An example of this is illustrated in line 4 and 5 in Listing 6.1.

In order for the scheduler to know which jobs that are to be considered successful another statement must be specified. An example of this is given in line 7 and 8 in Listing 6.1. In this example the first query in the queryfile should be true and the second should be false.

Listing 6.2 contains the methods available through the web service API. Focus in the implementation has been on providing few yet competent methods for allowing maximum customization on the user-side. The description of each method is found below Listing 6.2.

```
public string GetNewTicket(Credentials c);

public Task[] GetUserTickets(Credentials c);

public Status AddTask(string ticket, byte[] modelContent, byte[]
    queryContent, byte[] paramContent);

public Job[] GetJobs(string ticket, JobFilter jf, int start, int
    amount);

public Job GetJob(string ticket, int id);

public int GetJobCount(string ticket, JobFilter jf)

public Status UpdateJob(string ticket, Job j);

public bool IsValidTicket(string ticket)
```

Listing 6.2: Methods available through the web service API

Data containers

The current interface utilizes four different data containers. These are created using C# classes and they contain only fields which type are serializable. This allows the run-time to generate the needed SOAP messages. These four containers are:

Credentials For storing the username and associated password of authorized users

Job For storing information related to a single job e.g. parameters and output.

Status A common structure for wrapping both error messages and harmless informations.

Task For storing task related information e.g. creation and finish time.

Enumerated types

An enumeration called *JobFilter* is used to enable filter support on the job list. Given that the enumerations in C# consist of a native type, no serializability problem arises when encoded into SOAP messages. The user is able to apply the following filters

Any	Any job
Failed	Only failed jobs
Finished	Only finished jobs i.e. both failed and succeeded
Processing	Only currently running jobs
Scheduled	Only scheduled jobs i.e. ready to run and not yet executed
Succeeded	Only succeeded jobs

GetNewTicket

Given that users are able to provide valid credentials, a ticket is issued. This ticket can be used once to add a parameter sweep task, and thereafter multiple times to request task information. The ticket has the C# data type *string* and its content is the result of a the cryptographic hash function *SHA-256* applied to a unique task identification number concatenated with a private constant PSA service key.

GetUserTickets

Given credentials a list of appertaining tickets are returned.

AddTask

Given the following four input paramters

- a valid ticket,
- the *base*$_{64}$ encoding of a UPPAAL model file,
- the *base*$_{64}$ encoding of the models query file,
- the *base*$_{64}$ encoding of the UPPAAL PARMOS parameter file

a new task is added to the UPPAAL PARMOS service. As a result of the method call, a status message is returned containing information about the new task.

GetJobs

Given the following four input paramters

- a valid ticket,
- a filter option e.g. only finished jobs,
- a desired starting position in the job list,
- and a desired maximum amount of returned jobs

the `GetJobs` method returns a list of jobs related to a given task identified by the ticket matching the filter.

GetJob

Given a ticket and a Job ID, `GetJob` returns a specific job.

GetJobCount

Given the following two input paramters

- a valid ticket,
- and a filter option e.g. only finished jobs

the `GetJobCount` method returns the number of jobs related to a given task identified by the ticket matching the filter.

UpdateJob

Given a valid ticket and a job container, the job information is updated at the UPPAAL PARMOS service. This allows the user to change e.g. the computed parameters. However, this must happen before the job is executed. This methods is also used if the user want to cancel the job.

IsValidTicket

Given a ticked, returns true or false, depending on wether the ticket is valid or not.

6.4 PSA Service

The PSA Service is the most advanced component of UPPAAL PARMOS. As described in 5.1.3 on page 23, it consists of a scheduler that is able to utilize a number of scheduling algorithms and dispatcher types. Currently only a brute force scheduling algorithm, and a TORQUE dispatcher type has been implemented. However base classes, containing elements that should be shared between the scheduling algorithm and dispatcher types respectively, has also been implemented. The rest of this Section will describe further details of the dispatcher type base class, as the scheduling algorithm base class is used much the same way, and is trivially implemented.

The base class for dispatcher types is called *Dispatcher*, and every new dispatcher type must inherit this class. The new dispatcher type must then call the base class constructor, and override the base class Virtual classes with resource specific functionality (algorithm specific in the case of a scheduler algorithm).

Listing 6.4 shows the virtual methods, and Listing 6.3 shows the interface methods and properties in the base class. These will be described below Listing 6.4.

```
public void Start ();

public bool AddJob (ref MySqlConnection _connection, Job _job);

public DispatcherState GetStatus;

public int GetID
```

Listing 6.3: Interface methods and properties in the *Dispatcher* base class

```
protected virtual bool CheckConnection ();

protected virtual void TransferExecutable ();

protected virtual void TransferTaskFiles (int _taskId);

protected virtual void Open ();

protected virtual void Dispatch ();

protected virtual void Retrieve ();

protected virtual void Close ();
```

Listing 6.4: Virtual methods in the *Dispatcher* base class

AddJob

Is used by the scheduler via the scheduler algorithms to add a job to a dispatcher instance based on that jobs ID. It will only add a job that is not currently reserved by any dispatcher.

Start

Is the method invoked in a new thread, when the scheduler starts a *Dispatcher* instance. It contains a main loop for the dispatcher, making use of the overridden virtual methods when appropriate.

GetStatus and GetID

Used by the scheduler to retrieve information on a dispatcher instance.

CheckConnection

Checks the connection to the resource. Used in the base constructor.

TransferExecutable

Makes sure that the UPPAAL executable is present at the remote resource. Used in the base constructor.

TransferTaskFiles

Transfers files that are common for a task to the resource, if the dispatcher instance has not computed jobs from this task before.

Open

Performs actions needed before starting to loop in the main loop.

Dispatch

Sends jobs to the resource and starts them.

Retrieve

Retrieves finished jobs from the server and updates the database with the results.

Close

Performs closing actions after looping.

7 Test

During development of UPPAAL PARMOS, we have made a number of smaller test runs to test implemented features, and reveal bugs and performance problems. This chapter describes our test setup, and experiences we have made during a final test run of our software, processing all jobs of a task describing totally 125,751 jobs.

7.1 Test setup

We have been provided with access to a cluster at the AAU network, and relied on a server of our own for running the UPPAAL PARMOS application.

7.1.1 Our server

Our test setup for UPPAAL PARMOS was placed on a server outside the university network on a 20/2 mbit ADSL connection. The server it self is a HP Pavilion dv6000 laptop computer with the specifications shown in Table 7.1.

CPU	Mobile AMD Sempron(tm) Processor 3500+, 1.8GHz
L1	128KB
L2	512KB
Memory	1GB
OS	Ubuntu 10.04.1 LTS

Table 7.1: Server specifications

7.1.2 Resources

The only resource available for UPPAAL PARMOS during the test, was the *Benedict* cluster at AAU, which consists of 36 *brother* nodes, and 7 *sister* nodes, administered by the *mother* server. Specifications can be seen in Table 7.2 on the following page.

Of the available 43 worker nodes, we were only allowed to use 15 at a time, thus limiting the amount job that were able to run concurrently.

	Brother nodes	**Sister nodes**
CPU	Intel Pentium 4 CPU 2.80GHz (Northwood)	Xeon (Kentsfield) X3220@2.4GHz dual-die dual core (Quad core)
L1	Cache: 20KB (Instruction (12K) / data (8K))	Cache: 64KB (Instruction 32KB / data 32KB)
L2	Cache: 512KB	Cache: 4096KB
Memory	2GB	8GB
Disk	Seagate . . .	Seagate . . .
NIC	Intel e1000	Intel e1000
OS	Ubuntu Karmic Koala (9.10)	Ubuntu 6.06LTS

Table 7.2: Bendict worker nodes specifications (Taken from the wiki at `https://benedict.grid.aau.dk/wiki/`)

7.1.3 Model

For our test case, we used the Danfoss EKC 204A temperaure controller described in Section 2.3.2 on page 7. Unchanged, the task consist of roughly $8 \cdot 10^8$ distinct jobs. Unfortunately executing all these jobs on the available nodes on the *benedict* cluster would take far too long, which is why we have changed the value intervals for the parameters P_1 and P_2, so the task will only define 125,751 jobs. The parameter file used is shown in Listing 7.1, created from the specifications written in Section 2.3.2, and modified to have P_1 and P_2's intervals going from 0 through 500.

```
/* Parameters */
P1={0:500,1};
P2={0:500,1};

/* Conditions */
#{P1 >= P2};

/* Goals */
? q1;
? !q2;
```

Listing 7.1: Parameter specifications in the parameter file

7.2 Cluster shutdown

During the execution of the test, the cluster was shut down, which resulted in the PSA Service being unable to submit the jobs. Once this was discovered, we stopped the service to wait for the cluster to be available again.

Once the cluster was up and running again, we started the service back up, and it continued from the point where it had stopped, with only minimal cleanup of in databse necessary —

jobs that had been dispatched to the cluster, but had not yet returned any results had their status reset. However, due to software updates on the cluster, the UPPAAL verifier was no longer able to run properly, and aborted with error. However, the dispatching of the jobs continued without any problems, saving the error message to the database.

7.3 Results

Even though all successful verifications returned negative results, the test itself was successful. It showed that UPPAAL PARMOS behaves in the way it should, but also highlights two points of necessary future development:

- Automatic detection of and recovery after resource failure
- Minimizing jobs needed before useful results are obtained

Especially the latter is necessary if the system is going to be useful, as we only had time to run 125,751 jobs out of $8 \cdot 10^8$, and did not manage to find a single parameter combination yielding a positive result during the almost 4 days it took in total.

7.3.1 Performance

The test run took a total of three days, 22 hours and 27 minutes to execute 125,751 jobs, thus averaging about $2.7\frac{sec}{job}$, with 15 nodes, each running as many jobs concurrently as the cluster scheduler would allow.

8 Conclusion

In this report we presents UPPAAL PARMOS, a functioning PSA for UPPAAL. UPPAAL PARMOS has been created with the purpose of demonstrating our solution designed for solving the problem stated in Section 3.

Although our work is far from done, it shows the feasibility of the project, as we have already been able to distribute the verification of models onto multiple nodes of a cluster, resulting in decreased average verification time.

We have managed to create a simple and easy-to-learn specification of the parameter file, allowing the user to specify the parameter space, with certain simple constraints applied to it, along with the requirements for a job to be considered successful.

The PSA comes with a SOAP based web service for interacting with the PSA service. This frontend provides a compact set of operations that gives the functionality needed for submitting, accessing and mutating task and job information.

In order to test our application, we ran a PSA consisting of 125,751 jobs in total. The test was conducted at a single cluster and, while testing, we discovered that UPPAAL PARMOS is almost capable of handling failures at the resources on its own. We therefore predict that complete fault tolerance of resource failures will be relatively easy to implement in the future.

As emphasize in Chapter 9 on page 47 the study made in this project have resulted in a variety of useful ideas. Many of these are not implemened yet, however they will provide inspiration for future evolvement of this project. Especially the subject of optimizing the scheduling algorithms has been deemed of utmost importance in future development, as it is necessary in order to produce usable results within reasonable time, and as such, we can conclude that this should have the main focus in future projects.

9 Future Work

During the project, many ideas have been formed, and not all have been implemented yet. Most prominent among them is the need for advanced scheduling algorithms. UPPAAL PARMOS does support custom scheduling algorithms, however, for the time being, only brute force has been implemented, which has shown itself inefficient during testing, as expected. In order for the scheduling optimization to be possible, another thing is necessary; the detection of wehther a validation is successful or not. This has not been necessary for our brute force search algorithm, as it does not analyse on the results, but it is for any more advanced algorithm, and it is also necessary for the end user in order to view the result.

Another way of optimizing the search is to allow shortcircuiting of goals, i.e. one goal is only verified if another succeeds. If it fails however, the entire verification is considered failed and there is no need to verify the other goal.

Fault tolerance is also a design goal of ours, which we discovered during testing is almost fulfilled, and as such, the final bits and pieces should be made, in order to have complete fault tolerance against resource failures.

Furthermore, a number of practical optimizations/tweaks should be made, such as increasing the size of the job ID in the database, as it is currently only a 32 bit integer, which has a real propability of being exceeded. UPPAAL PARMOS should also be rewritten to be eventbased, instead of polling between sleeping. Further support for parameter constraints would also optimize the parameter sweep by decreasing the size of the parameter space.

Lastly, supporting other versions of UPPAAL or other verification engines would broaden the scope and usability of our tool, but for now we have chosen to focus only on the standard UPPAAL verification engine

Appendix A

A.1 UPPAAL Bridge Model XML

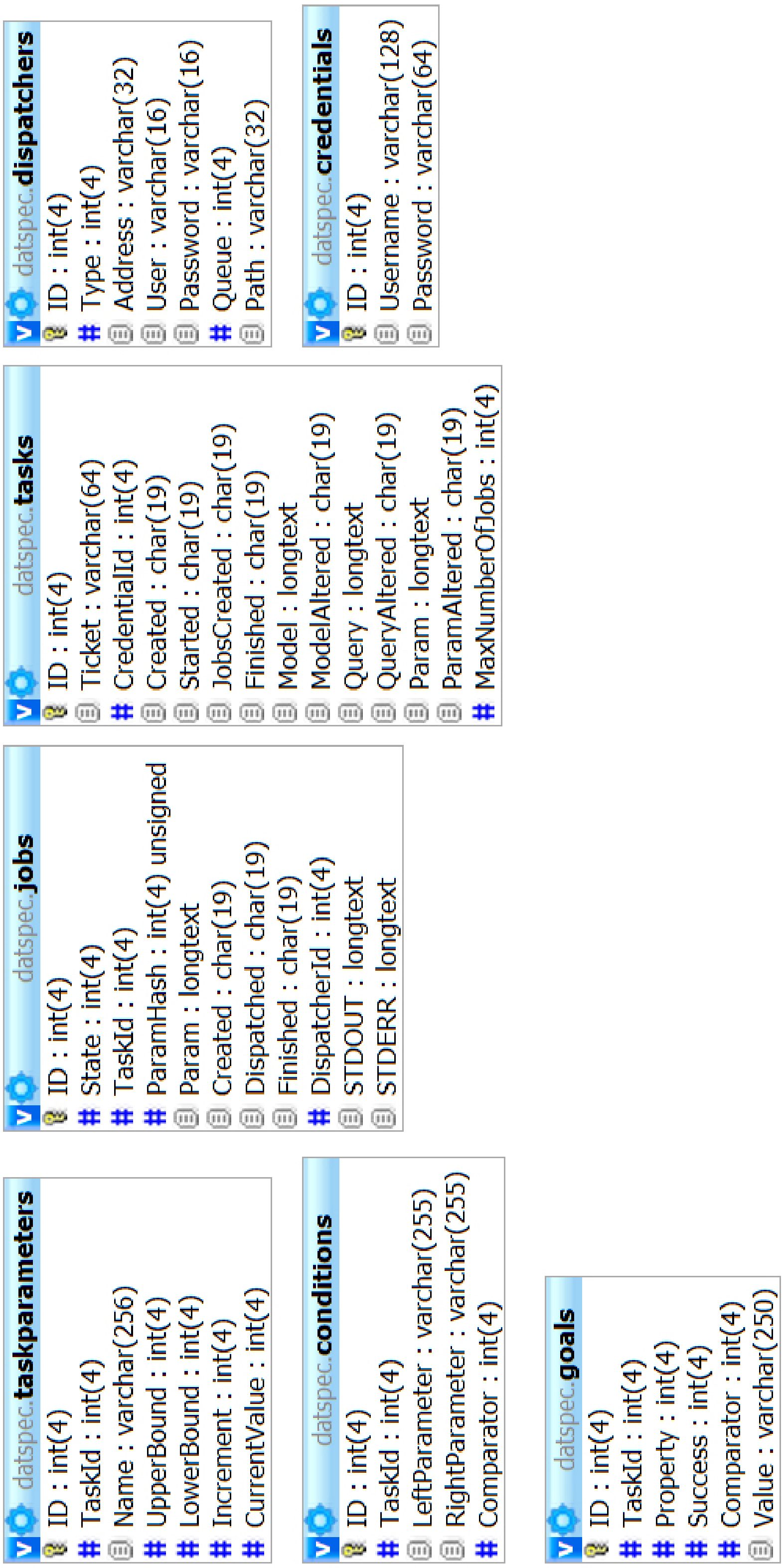
This section contains the XML content of the Bridge model described in Section 2.3.1 on page 5. The XML content is listed in Listing A.1.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System 1.1//EN' 'http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd'>
<nta>
    <declaration>
        chan take, release;
        int L;
        clock time;

        const int max = 30;
        const int fastest = 5;
        const int fast = 10;
        const int slow = 20;
        const int slowest = 25;
    </declaration>
    <template>
        <name>Viking</name>
        <parameter>const int speed</parameter>
        <declaration>clock  y;</declaration>
        <location id="id0" x="216" y="-32">
            <name x="184" y="-64">over</name>
        </location>
        <location id="id1" x="216" y="80">
            <name x="176" y="48">ready</name>
        </location>
        <location id="id2" x="216" y="200">
            <name x="184" y="168">safe</name>
            <label kind="invariant" x="184" y="216">y &lt;= max</label>
        </location>
        <location id="id3" x="-16" y="200">
            <name x="-64" y="168">u_over</name>
        </location>
        <location id="id4" x="-16" y="80">
            <name x="-72" y="48">u_ready</name>
        </location>
        <location id="id5" x="-16" y="-32">
            <name x="-64" y="-64">unsafe</name>
        </location>
        <location id="id6" x="328" y="200">
```

```
            <name x="312" y="168">faint</name>
        </location>
        <init ref="id5"/>
        <transition>
            <source ref="id0"/>
            <target ref="id5"/>
            <label kind="synchronisation" x="72" y="-56">release!</label>
        </transition>
        <transition>
            <source ref="id1"/>
            <target ref="id0"/>
            <label kind="guard" x="224" y="-8">y &gt;= speed</label>
        </transition>
        <transition>
            <source ref="id2"/>
            <target ref="id1"/>
            <label kind="guard" x="224" y="104">L ==  1 and y &lt;= max</label>
            <label kind="synchronisation" x="176" y="128">take!</label>
            <label kind="assignment" x="224" y="128">y = 0</label>
        </transition>
        <transition>
            <source ref="id3"/>
            <target ref="id2"/>
            <label kind="synchronisation" x="64" y="176">release!</label>
        </transition>
        <transition>
            <source ref="id4"/>
            <target ref="id3"/>
            <label kind="guard" x="-8" y="96">y &gt;= speed</label>
            <label kind="assignment" x="-8" y="120">y = 0</label>
        </transition>
        <transition>
            <source ref="id5"/>
            <target ref="id4"/>
            <label kind="guard" x="-8" y="-8">L ==  0</label>
            <label kind="synchronisation" x="-64" y="15">take!</label>
            <label kind="assignment" x="-8" y="16">y =  0</label>
        </transition>
        <transition>
            <source ref="id2"/>
            <target ref="id6"/>
            <label kind="guard" x="240" y="176">y &gt;= max</label>
        </transition>
    </template>
    <template>
        <name>Torch</name>
        <location id="id7" x="40" y="80">
            <name x="8" y="48">free1</name>
        </location>
        <location id="id8" x="184" y="80">
            <name x="152" y="48">free2</name>
            <urgent/>
        </location>
        <location id="id9" x="40" y="208">
            <name x="16" y="216">one</name>
        </location>
        <location id="id10" x="184" y="208">
            <name x="160" y="216">two</name>
        </location>
        <init ref="id7"/>
        <transition>
            <source ref="id7"/>
            <target ref="id8"/>
            <label kind="synchronisation" x="88"
               y="56">take?</label>
        </transition>
        <transition>
            <source ref="id8"/>
            <target ref="id9"/>
        </transition>
        <transition>
            <source ref="id8"/>
            <target ref="id10"/>
            <label kind="synchronisation" x="192"
               y="128">take?</label>
        </transition>
        <transition>
            <source ref="id9"/>
            <target ref="id7"/>
            <label kind="synchronisation" x="-24"
               y="128">release?</label>
            <label kind="assignment" x="48" y="112">L=
               -L+1</label>
        </transition>
        <transition>
            <source ref="id10"/>
            <target ref="id9"/>
            <label kind="synchronisation" x="88"
               y="208">release?</label>
        </transition>
    </template>
    <system>
        Viking1 = Viking(fastest);
        Viking2 = Viking(fast);
        Viking3 = Viking(slow);
        Viking4 = Viking(slowest);

        system Viking1, Viking2, Viking3, Viking4, Torch;
    </system>
</nta>
```

Listing A.1: XML encoding of UPPAAL Bridge model

A.2 Database schemas



datspec.**dispatchers**
ID : int(4)
Type : int(4)
Address : varchar(32)
User : varchar(16)
Password : varchar(16)
Queue : int(4)
Path : varchar(32)

datspec.**credentials**
ID : int(4)
Username : varchar(128)
Password : varchar(64)

datspec.**tasks**
ID : int(4)
Ticket : varchar(64)
CredentialId : int(4)
Created : char(19)
Started : char(19)
JobsCreated : char(19)
Finished : char(19)
Model : longtext
ModelAltered : char(19)
Query : longtext
QueryAltered : char(19)
Param : longtext
ParamAltered : char(19)
MaxNumberOfJobs : int(4)

datspec.**jobs**
ID : int(4)
State : int(4)
TaskId : int(4)
ParamHash : int(4) unsigned
Param : longtext
Created : char(19)
Dispatched : char(19)
Finished : char(19)
DispatcherId : int(4)
STDOUT : longtext
STDERR : longtext

datspec.**taskparameters**
ID : int(4)
TaskId : int(4)
Name : varchar(256)
UpperBound : int(4)
LowerBound : int(4)
Increment : int(4)
CurrentValue : int(4)

datspec.**conditions**
ID : int(4)
TaskId : int(4)
LeftParameter : varchar(255)
RightParamater : varchar(255)
Comparator : int(4)

datspec.**goals**
ID : int(4)
TaskId : int(4)
Property : int(4)
Success : int(4)
Comparator : int(4)
Value : varchar(250)

Figure A.1: Database schemas

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability	23
APST	AppLeS Parameter Sweep Template	9
FMRI	Functional Magnetic Resonance Imaging	8
IIOP	Internet Inter-Orb Protocol	18
JMS	Java Message Service	18
PBS	Portable Batch System	9
PSA	Parameter Sweep Application	1
REST	Representational State Transfer	18
RPC	Remote Procedure Call	18
SOAP	Simple Object Access Protocol	18
SRB	Storage Resource Broker	10
TORQUE	Terascale Open-Source Resource and QUEue Manager	17
UDDI	Universal Description, Discovery and Integration	19
WS	Web Service	18
WSDL	Web Services Description Language	18
W3C	World Wide Web Consortium	18
XSD	XML Schema Document	21

Bibliography

Mod Mono. `http://www.mono-project.com/Mod_mono`. Accessed January 10, 2010.

D. Abramson, R. Sosic, J. Giddy, and B. Hall, **2002**. D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: a tool for performing parametised simulations using distributed workstations. In *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on*, pages 112–121. IEEE, 2002. ISBN 0818670886.

Abramson, Peachey, and Lewis, **2006**. D. Abramson, T. Peachey, and A. Lewis. *Model optimization and parameter estimation with Nimrod/O*. Computational Science–ICCS 2006, 3991/2006, 720–727, 2006.

Alliance, **July 2009**. The Globus Alliance. *GRAM*, July 2009. `http://dev.globus.org/wiki/GRAM`. Accessed: December 16, 2010.

Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke, **1999**. Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: a data movement and access service for wide area computing systems. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, IOPADS '99, pages 78–88, New York, NY, USA, 1999. ACM. ISBN 1-58113-123-2. doi: http://doi.acm.org/10.1145/301816.301839. `http://doi.acm.org/10.1145/301816.301839`.

R. Buyya, D. Abramson, and J. Giddy, **2000**. R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *hpc*, page 283. Published by the IEEE Computer Society, 2000.

Casanova and Berman, **2003**. H. Casanova and F. Berman. *Parameter Sweeps on the Grid with APST*. Wiley Online Library, 2003. ISBN 0470853190.

Christensen, Curbera, Meredith, and Weerawarana, **March 2001**. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*. W3C Note, March 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

Clickatell, **2010**. Clickatell. *Developer API*, 2010. `http://www.clickatell.com/developers/clickatell_api.php`. Accessed: December 1, 2010.

Couluris, Dollimore, and Kindberg, 2005. G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. Addison-Wesley Longman, 2005.

eScience Center, 2010. University of Copenhagen eScience Center. *What is eScience?*, 2010. `http://www.escience.ku.dk/what_is_escience/`. Accessed: December 14, 2010.

Facebook, 2010. Facebook. *Facebook Developers Documentation*, 2010. `http://developers.facebook.com/docs/`. Accessed: December 1, 2010.

Flickr, 2010. Flickr. *The App Garden - API Documentation*, 2010. `http://www.flickr.com/services/api/`. Accessed: December 1, 2010.

IBM. IBM. *Tivoli Workload Scheduler LoadLeveler*. `http://www-03.ibm.com/systems/software/loadleveler/`. Accessed: December 16, 2010.

Larsen, Larsen, Nielsen, Skou, and Wasowski, December 2003. Kim G. Larsen, Ulrik Larsen, Brian Nielsen, Arne Skou, and Andrzej Wasowski. *Danfoss EKC Trial Project Deliverables*, Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, 9220 Aalborg Ø, December 2003.

Microsoft .NET Framework SDK QuickStart Tutorials, 2010. Microsoft .NET Framework SDK QuickStart Tutorials. *ASP.NET Web Services QuickStart Tutorial*, 2010. `http://quickstarts.asp.net/QuickStartv20/webservices/`. Accessed: October 22, 2010.

MySQL, a. MySQL. *About MySQL*, a. `http://www.mysql.com/about/`. Accessed: January 10, 2011.

MySQL, b. MySQL. *MySQL 5.1 Manual: MySQL Connector/NET*, b. `http://dev.mysql.com/doc/refman/5.1/en/connector-net.html`. Accessed: January 10, 2011.

MySQL, c. MySQL. *MySQL 5.1 Manual: The InnoDB Storage Engine*, c. `http://dev.mysql.com/doc/refman/5.1/en/innodb-storage-engine.html`. Accessed: January 10, 2011.

Novell editors. Novell editors. *What is Mono?* `http://www.mono-project.com/What_is_Mono`. Accessed: December 17, 2010.

PayPal, 2010. PayPal. *Technical Documentation*, 2010. `https://cms.paypal.com/us/cgi-bin/?cmd=_render-content&content_ID=developer/library_documentation`. Accessed: December 1, 2010.

T. C. Peachey, 2005. T. C. Peachey. *The Nimrod/O Users' Manual v2.6*. Monash University, 2005. `http://messagelab.monash.edu.au/NimrodO/Documentation?action=download&upname=NimrodOUserManual_2.6.pdf`.

Sertelon, **October 2010**. Romain Sertelon. *Uppaal for Parametric Models*, CISS - University of Aalborg, October 2010.

Smith, Jenkinson, Woolrich, Beckmann, Behrens, Johansen-Berg, Bannister, De Luca, Drobnjak, Flitney, et al., **2004**. S.M. Smith, M. Jenkinson, M.W. Woolrich, C.F. Beckmann, T.E.J. Behrens, H. Johansen-Berg, P.R. Bannister, M. De Luca, I. Drobnjak, D.E. Flitney, et al. *Advances in functional and structural MR image analysis and implementation as FSL.* Neuroimage, 23, 208–219, 2004. ISSN 1053-8119.

The Condor Team. The Condor Team. *Condor.* `http://www.cs.wisc.edu/condor/description.html`. Accessed: December 16, 2010.

The PHP Group, **2010**. The PHP Group. *PHP Manual section: Webservices*, 2010. `http://www.php.net/manual/en/refs.webservice.php`. Accessed: October 22, 2010.

Twitter, **2010**. Twitter. *API Documentation*, 2010. `http://dev.twitter.com/doc`. Accessed: December 1, 2010.

UPPAAL Team, **November 2009**. UPPAAL Team. *UPPAAL 4.0 : Small Tutorial*, November 2009. `http://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf`. Accessed: December 10, 2010.

W3C, **2010**. W3C. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, 2010. `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`. Accessed: December 15, 2010.

W3C, **2004**. W3C. *W3C Web Services Glossary*, 2004. `http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/`. Accessed: October 18, 2010.

J. Wang, I. Altintas, P.R. Hosseini, D. Barseghian, D. Crawl, C. Berkley, and M.B. Jones, **2009**. J. Wang, I. Altintas, P.R. Hosseini, D. Barseghian, D. Crawl, C. Berkley, and M.B. Jones. Accelerating Parameter Sweep Workflows by Utilizing Ad-hoc Network Computing Resources: an Ecological Example. In *2009 Congress on Services-I*, pages 267–274. IEEE, 2009.

Wibisono, Zhao, Belloum, and Bubak, **2008**. A. Wibisono, Z. Zhao, A. Belloum, and M. Bubak. *A framework for interactive parameter sweep applications.* Computational Science–ICCS 2008, 5103, 481–490, 2008.