Distributed parameter sweep for UPPAAL *models*

MASTER'S THESIS, SPRING 2011
PETER SCHMIDT FREIBERG
JIMMY MERRILD KRAG
BRIAN VILLUMSEN
DEPARTMENT OF COMPUTER SCIENCE
AALBORG UNIVERSITY
JUNE 1ST, 2011

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

Title: Distributed parameter sweep for UPPAAL models
Project area: Distributed Systems
Project period: spring 2011
Project group: f11d603a
Participants:

Peter Schmidt Freiberg

Jimmy Merrild Krag

Brian Villumsen

Supervisor: Brian Nielsen
Copies: 5
Page count: 106
Appendices: 1
Finished: June 1st, 2011

Synopsis:

The use of a tool when conducting a parameter sweep is essential, since the amount of possible combination is often very large. This thesis presents UPPAAL PARMOS, a Parameter Sweep Application that distributes a parameter sweep of verifications of a UPPAAL model to multiple resources, while employing an optimization scheme to prioritize individual model configurations, thus allowing for faster access to desired results. The thesis includes both a fully described design of UPPAAL PARMOS as well as an implementation and test. Additional, a study of heuristic optimization algorithms has been conducted in order to utilize three of these, specifically Hill-Climbing, Simulated Annealing, and Pareto Archived Evolution Strategy, in UPPAAL PARMOS. Based on experiments on our implementation we conclude that there are grounds for utilizing UPPAAL PARMOS to acquire faster access to desired results.

Preface

This report serves as our master's thesis, which documents our work through the period of February 1st to June 1st 2011. This thesis, which marks the completion of our specialisation year, is produced while based at the Distributed and Embedded Systems research unit at the Department of Computer Science at Aalborg University. The work done in this thesis build upon our preliminary work titled *Distributed parameter sweep for* UPPAAL *models* [19], which was conducted in the autumn of 2010. No sections or paragraphs of our previous work directly exists in this thesis, yet the ideas obtained and experience gained has been continued in this thesis.

While [19] includes more background and studies of existing parameter sweep systems, Chapter 2 of this thesis includes a only basic knowledge. The knowledge gain from reading Chapter 2, should be sufficient for a reader familiar with the areas of distributed systems and model checking.

During the course of our work, Aalborg University provided us with access to the cluster *Fyrkat*, which has 124 nodes, with a total of 243 processors, providing 1020 processing cores at 2.33-2.93 GHz. however, only 19 nodes were at a disposal, of which 14 were only available during the final weeks of the project period, providing us with a total of 132 processing cores. Further specifications on the available nodes is provided in Chapter 6

The implementation of UPPAAL PARMOS, together with results from our experiments and our previous work [19] is found at the following web page: `http://cs.aau.dk/~bv1645`

Contents

1 Introduction **1**

2 Prerequisites **3**
2.1 The UPPAAL tool . . . 3
2.2 Parameter Sweep Application . . . 9
2.3 Clusters . . . 11

3 Design of UPPAAL PARMOS **13**
3.1 Design principles . . . 13
3.2 Parameter Sweep Application . . . 19
3.3 Task specification . . . 26
3.4 Storage . . . 27
3.5 Web service . . . 30
3.6 Graphical User Interface . . . 31

4 Algorithms **33**
4.1 Notation and definitions . . . 34
4.2 Algorithm classes . . . 36
4.3 Hill-Climbing . . . 41
4.4 Simulated Annealing . . . 45
4.5 Pareto Archived Evolution Strategy . . . 52
4.6 Parallelization . . . 55

5 Implementation of UPPAAL PARMOS **57**
5.1 Parameter Sweep Application . . . 59
5.2 Storage . . . 69
5.3 Front-end . . . 70
5.4 Algorithms . . . 73
5.5 Resource proxies . . . 75
5.6 Software platform . . . 78

6 Test **81**
6.1 Test environment . . . 81
6.2 Settings . . . 82
6.3 Overhead . . . 83
6.4 Scalability . . . 89
6.5 Algorithms . . . 91
6.6 Performance . . . 93

7 Conclusion **95**
7.1 Results 95
7.2 Future work 96

8 Appendix **97**

Acronyms **101**

Bibliography **103**

1 Introduction

The continuous development in the areas of information technology and electronics have reached a state, where the developed products includes less electronics, in terms of hardware, and more information technology, in terms of software. This has given rise to an interdisciplinary area of research, concerning the topic of reactive systems [21, pp. 2-3]. Here, formalisms from theories of automata are used to describe and model a type of systems, known as real-time systems. A real-time system is a concurrent software system, made up by a number of processes, each marked with a deadline, specifying when the execution of the processes must be completed [3, pp. 2-3]. These systems includes operating systems, communication protocols, control programs and embedded software running in e.g. mobile phones, network routers, and the airbag system of a car.

Once a formal model of a real-time system has been developed e.g. using a network of timed automata, it can be used to extract information about the behaviour of the modelled system. However, as this can be an exhaustive manoeuvre, an automation tool called a model checker is utilized to conduct the information extraction. This thesis focus on the UPPAAL model checker, developed in a collaboration between Uppsala University and Aalborg University, hence the name UPPAAL.

In order to extract information about the behaviour of the modelled system, the user together with a model specifies certain properties that need to be examined. These properties can include questions such as; is a state visited more than once, or is it in any way possible for the system to deadlock during execution. The model checker will then do an exhaustive search in the state-space, in order to check whether each of the specified properties are satisfied.

In the previously described scenario, it is assumed that the model utilized is of such a quality, that it perfectly matches the modelled system. Yet, this is not always the case, and sometimes certain value settings in the model needs to be fitted. While a single value setting with a limited range produces only a small amount of required model checking runs, multiple value settings with a wider range requires exponentially many. In order to cope with this tremendous increase in workload, we suggested in [19] an automated scheme known as a Parameter Sweep Application (PSA). The PSA systematically changes the value settings in the model and subsequently distributes the run of the model checking application to compute resources.

In this thesis, we take the experiences gained from our previous work, and design a new version of the PSA. This new PSA is not only capable of conduction a parameter sweep, but also of minimizing the amount of required sweeps needed before the user is presented with a desireable result. This will minimize the waiting time for the user, and thereby also the development- and testing time of a real-time system.

Reading guide

The content of this thesis 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 thesis 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 Internet Protocol (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 101 for reference.

2 Prerequisites

The purpose of this chapter is to provide the reader with better understanding of various concepts, used throughout the rest of this thesis. The chapter includes three sections; this first is a presentation of the UPPAAL tool, the second is a description of PSAs and the third describes computer clusters.

2.1 The UPPAAL tool

UPPAAL is a tool for verification of real time systems, modelled as a network of finite state machines with clocks, known as Network of Timed Automata (NTA). Given a set of queries, expressed as reachability properties, UPPAAL performs an exhaustive search on the dynamic state of a model in order to verify its properties [36]. This concept is illustrated in Figure 2.1

Figure 2.1: UPPAAL verification

2.1.1 Model

The UPPAAL model consists of a number of Timed Automatons (TAs), where each TA consists of a number of locations connected by edges. Each location can have invariants, which limits when the state can be entered, or when it must be exited, practically disabling the location whenever the invariant is invalidated. Likewise, edges can be decorated with guards, enabling the edge when the guard is satisfied, disabling it otherwise. Furthermore, edges can also be decorated using update expressions, which modify the state of the TA, or synchronizations, which performs a two-way handshake synchronization between TAs, as well as selections, which non-deterministically assigns a value within a given range to a named variable. Below, a number of UPPAAL features are described.

Data types and containers Besides bounded integers and booleans, both arrays, structs and scalars are supported in UPPAAL, with struct and scalar being keywords, used to denote variables of these types, and arrays being denoted using the [size] notation.

Clocks clocks are a way of measuring time in a UPPAAL system, progresses evenly on over the entire system. However, updates to clocks are restricted to be simple non-negative integers, no complex expressions.

Constants Integer and boolean variables as well as arrays and structs of integers and booleans can be declared constants using the `const` keyword.

Synchronization channel Channels can be defined to provide binary synchronization between TAs.

Urgent and Committed locations These locations freeze time, in order to proceed immediately, such as when several synchronization channels needs to be synchronized. Committed locations also require that the next transition exits the location.

Global/local declarations UPPAAL variables can be defined locally to the TA, or it can be declared in global scope, thus being available to all TAs.

Templates a TA is modelled as a template, and can be instantiated with different parameters.

Functions Functions can be declared alongside variable declarations, both local and global in scope. Returning values from functions is possible, however, only structs and integers are allowed.

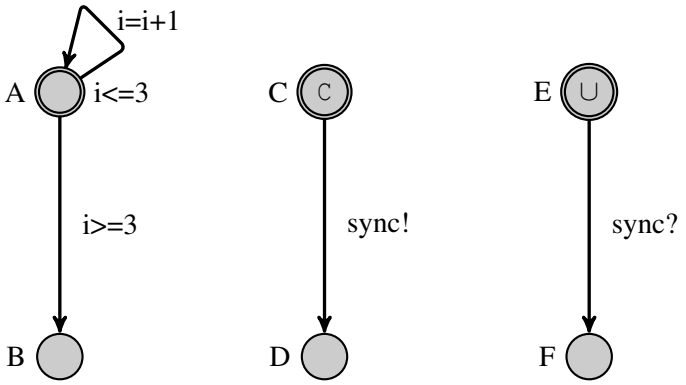


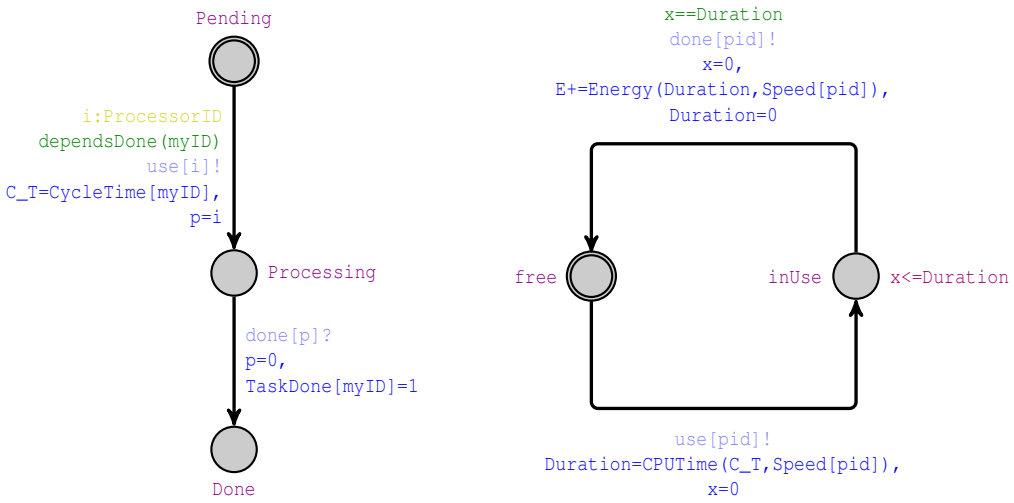
Figure 2.2: UPPAAL model examples

Figure 2.2 shows examples of UPPAAL TAs, in which *A, C, E* are initial locations, which is where the TA start, and *C* is a committed location, and *E* is an urgent location, meaning that it must leave immediately. In order to leave *C*, it has to synchronize on the channel named *sync*, which the $E \rightarrow F$ edge is able to. The *A* location has an invariant, saying that the variable *i* must be less than or equal to three, meaning that it must leave this location before *i* is greater than three. Furthermore, it has an edge leading back to *A* with an update on *i*, and another leading to *B*, with a guard that *i* must be no greater than three.

UPPAAL model example: Task Graph Scheduler

The Task Graph Scheduler (TGS) models a scheduler, whoose goal it is to map *T* tasks onto *P* processors, such that all tasks are completed within *D* time units. Furthermore, in order to preserve energy, tasks can scheduled on slower, more energy-effective processors, whenever *D* permits it.

The tasks may have dependencies between them, such that one task must be completed before another can begin. The specific dependencies for the tasks in the TGS are shown in



(a) Task (b) Processor

Figure 2.3: The TAs of the Task Graph Scheduler

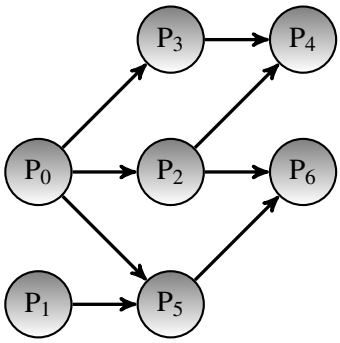


Figure 2.4: Task dependencies.

Figure 2.4, where it can be seen that P_0 must be executed before P_3 and P_2, whereas for P_5 to start, both P_0 and P_1 must be finished.

In order to model this behaviour, two processes are created, one to model a processor, and one to model a task, which are tied together in the system declaration in Listing 2.4

The task is modeled as shown in Figure 2.3(a), with only three states. A *Pending* state, in which it awaits until a processor is ready to execute it, at which point it will transition to the *Processing* state, which it will stay in, until the task is completed, and it transitions to the *Done* state.

To make sure that a task does not transit to the *Processing* state before their dependencies are met, a guard is inserted on the transition from *Pending* to *Processing*, which checks whether the dependencies are met, using a globally defined function called `dependsDone` defined in lines 22 to 28 in Listing 2.1

Once all dependencies are satisfied, a *selection* statement will nondeterministically select a processor to synchronize with, utilizing one of the channels in the *use* channel array.

For the task to transit to the *done* state, it must once more synchronize with the processor

once more, using a *done* channel, at which point it will update its status to reflect that it is now in the *Done* state.

The processor, shown in figure 2.3(b), consists of two states, namely *free* and *inUse*, which reflects the processor being idle and working.

When transitioning from *free* to *inUse*, the processor synchronizes with a task, using a *use* channel, thus allowing a task to transition to its *Processing* state.

The *inUse* state has an invariant, ensuring that the processor exits when sufficient time has passed, for the task to be completed. The elapsed time is measured using a clock x, defined in the processors local declarations as shown in Listing 2.2, which is compared to a variable named *Duration*, whose value is calculated using the function *CPUTime*, defined in lines 22 to 28 in Listing 2.1, which gives the duration of the task, at the speed of the specific processor.

The guard on the transition back to the *free* state requires that the clock x exactly equals that of the tasks duration, thus preventing the processor to spend too much or too little time in the *inUse* state. Upon transitioning to the *free* state, the energy expended on processing the task is added to the total energy expenditure of all tasks processed so far.

```
1  const int NP=4; //number of Processors
2  const int NT=7; //number of Tasks
3  const int MAXC=200; //Max cycle duration of task
4
5  typedef int [0,NP-1] ProcessorID;{
6  typedef int [0,NT-1] TaskID;{
7  typedef int[0,MAXC] Duration_t;
8
9  const int [0,1] Pred [TaskID][TaskID]={
10 {0,0,0,0,0,0,0}, //p0 depends on none
11 {0,0,0,0,0,0,0}, //p1 depends on none
12 {1,0,0,0,0,0,0}, //p2 depends on p0
13 {1,0,0,0,0,0,0}, //p3 depends on p0
14 {0,0,1,1,0,0,0}, //p4 depends on p2, p3
15 {1,1,0,0,0,0,0}, //p5 depends on p0,p1
16 {0,0,1,0,0,1,0}  //p6 depends on p2,p5
17 };
18
19 const Duration_t CycleTime[TaskID]={9,30,18,48,6,9,6};
20 const int Speed[NP]={1,2,3,4}; //cycles per time units
21 int [0,1] TaskDone[NT];
22 bool dependsDone(TaskID i){{
23     for(j:TaskID)
24        if(Pred[i][j])
25         if(! TaskDone[j])
26             return false;
27     return true;
28 }{
29
30 Duration_t CPUTime(int Cycles,int Speed){
31     int dur = Cycles/Speed;
```

```
32      if(Cycles%Speed>0) dur+=1; //ceil
33      return dur;
34  }
35
36  int Energy(int Duration, int Speed){
37  return (Duration *Speed*Speed*Speed);
38  }
39
40  urgent chan use[ProcessorID];
41   chan done[ProcessorID];
42
43  int [0,MAXC] C_T;
44  int E;
45  clock now;
```

Listing 2.1: TGS global declarations

```
1  clock x;
2  Duration_t Duration;
```

Listing 2.2: TGS processor declarations

```
1  ProcessorID p;
```

Listing 2.3: TGS task declarations

```
1  system Processor,Task;
```

Listing 2.4: TGS system declarations

2.1.2 Query

The queries are expressed using a query language as described by [37], the grammar of which can be seen in Listings 2.5, where `Expression` is a side effect free expression, further explained in [37].

```
1  Prop
2    = 'A[]' Expression | 'E<>' Expression | 'E[]' Expression
3      | 'A<>' Expression | Expression '-->' Expression
4      | 'sup' ':' List
5      | 'sup' '{' Expression '}' ':' List
6      | 'inf' ':' List
7      | 'inf' '{' Expression '}' ':' List
8    :
9
10 List
11   = Expression | Expression ',' List
12   :
```

Listing 2.5: the UPPAAL query language grammar

The *A* and *E* are equivalent to the mathematical $\forall$ and $\exists$ respectively, and the semantics of the different queries is shown in Equation 2.1.

$$\begin{aligned}
\exists\langle\rangle\ p &: \textit{there exists a path where p eventually holds.}\\
\forall[]\ p &: \textit{for all paths p always holds.}\\
\exists[]\ p &: \textit{there exists a path where p always holds.}\\
\forall\langle\rangle\ p &: \textit{for all paths p will eventually hold.}\\
sup\{p\} : list &: \textit{returns the supremum of the expressions}\\
&\quad \textit{in list when evaluated where p holds}\\
inf\{p\} : list &: \textit{returns the infenum of the expressions}\\
&\quad \textit{in list when evaluated where p holds}
\end{aligned} \tag{2.1}$$

UPPAAL query example: Task Graph Scheduler

The goal of the TGS is to schedule the tasks, such that they complete within *D* time units, thus UPPAAL must verify that such a configuration exists, which the first query in Listing 2.6 depicts. Furthermore, it should not be possible that the scheduler ever enters a deadlock, such that the tasks do not finish. This is depicted in the next query. In the final query, it is tested whether a schedule exists in which the tasks can be done at all, with no regard for *D*, which indicates that *D* may be too ambitious, if this last query is satisfied, but the first is not.

```
1 E<> forall (i:TaskID) Task(i).Done and now <= 50
2 A[] not (forall (i:TaskID) Task(i).Done) imply not deadlock
3 E<> forall (i:TaskID) Task(i).Done
```

Listing 2.6: uppaal query file

2.2 Parameter Sweep Application

A PSA is an application structured as a set of multiple "experiments", each of which is executed with a distinct set of parameters [6]. This allows for performing multiple experiments, without much, if any, interaction from the user, once it has been started. Furthermore, some PSA can direct the parameter sweep, based upon the results of previous experiments, thus possibly limiting the duration of the parameter sweep, if an exhaustive parameter sweep is not necessary, for example to optimize certain parameters, in which case only the best parameters are needed.

Using scripting languages such as *Bash*, it is fairly straightforward to create a simple PSA. However, while this may be adequate for a single parameter sweep, it may quickly become inadequate for multiple or complex parameter sweeps, in which case a dedicated PSA may be the solution needed. Furthermore, several PSAs are also capable of distributing the sweep onto several computer nodes, thus speeding up the parameter sweep, by utilizing the fact that individual parameter sweep experiments are often largely independent.

Below is a description of a few PSAs.

***APST*:** The AppLes Parameter Sweep Template (APST) is an application for scheduling and deploying large scale parameter sweeps on grid platforms [5]. It works in client-server configuration, where a daemon is responsible for deploying the experiments, as well as monitoring running experiments. The daemon also acts as a server, to which the user, through a client application, is able to submit parameter sweeps, as well as inquire to the progress of already submitted jobs. Furthermore, APST is capable of utilizing third-party schedulers to schedule the individual experiments.

***Nimrod*:** Nimrod provides much the same facilities as APST, however it allows even more flexibility in the parameter sweep specification, as it provides a small scripting language, allowing the user to specify multiple actions to be performed at various stages of the parameter sweep. Furthermore, Nimrod also allow the user to direct the parameter sweep, by utilizing one of several built-in single objective optimization algorithms.

error may be necessary, to adjust all parameters, in order to satisfy all queries. Furthermore, in order to improve the performance of the model, more trial-and-error attempts may be necessary, before the optimal values of the parameters are found.

Both of the above scenarios can be automated, through utilization of a PSA, by letting the parameters be a subset of the variables in the model, thus allowing for the PSA to actually change the model.

In the TGS, it would be obvious to let the parameter sweep take place over the *NP* variable, as this will thus vary the number of processors available to schedule the tasks onto. Furthermore, the number of tasks may also be changed, or their duration, to see how this may affect the scheduling, and by creating a few additional queries, monitoring the performance of the scheduler, using the *sup* and *inf* queries, it can even be possible to find optimal parameter settings, for things such as the energy consumption of the system, or the execution time of all the tasks.

The Listing 2.7 shows a uppaal model, which has been prepared for parametrization, on the *NP*, *B* and *P* variables. This parametrization however requires that the *Speed* array is initialized before anything else. Therefore, a committed state is inserted in the Processor process, as the start state, with an edge leading to the former start state, on which the method *procInit* is called, such that each processor initializes its own place in the Speed array before starting any task execution.

```
1  const int NP=8; //number of Processors
2  const int NT=8; //number of Tasks
3  const int P=29;
4  const int B=18;
5
6  const int I=P;
7  const int MAXC=100; //Max cycle duration of task
8
9  const int MaxTaskSize=10;
10 typedef int [0,NP-1] ProcessorID;
11 typedef int [0,NT-1] TaskID;
12 typedef int[0,MAXC] Duration_t;
13
14 int [0,1] Pred [NT][NT]={
15 // dependencies based on the Berkeley MPEG-1 encoder
16 {0,0,0,0,0,0,0,0,0,0},//I1
17 {1,0,0,0,0,0,0,0,0,0},//P4
18 {1,1,0,0,0,0,0,0,0,0},//B2
19 {1,1,0,0,0,0,0,0,0,0},//B3
20 {0,1,0,0,0,0,0,0,0,0},//P7
21 {0,1,0,0,1,0,0,0,0,0},//B5
22 {0,1,0,0,1,0,0,0,0,0},//B6
23 {0,0,0,0,1,0,0,0,0,0},//P10
24 {0,0,0,0,1,0,1,0,0,0},//B8
25 {0,0,0,0,1,0,1,0,0,0} //B9
26 };
27
28 Duration_t CycleTime[NT];//={9,30,18,48,6,9,6};
29
30 Duration_t CycleTimeData[MaxTaskSize]={I,B,B,P,B,B,P,B,B,P};
31 void taskInit(int tid)
32 {
33     CycleTime[tid]=CycleTimeData[tid];
34     for (i:TaskID) Pred[tid][i]=PredData[tid][i];
35 }
36 int Speed[NP];
37 void procinit(int pid)
38 {
39     Speed[pid]=(pid+1);//+SpeedMult;
40 }
41
42 //const int Speed[NP]={1}; //cycles per time units
43 int [0,1] TaskDone[NT];
44 bool dependsDone(TaskID i){
```

```
45      for(j:TaskID)
46         if(Pred[i][j])
47          if(! TaskDone[j])
48              return false;
49      return true;
50  }
51
52  Duration_t CPUTime(int Cycles,int Speed){
53      int dur = Cycles/Speed;
54      if(Cycles%Speed>0) dur+=1; //ceil
55      return dur;
56  }
57
58  int Energy(int Duration, int Speed){
59  return (Duration *Speed*Speed*Speed);
60  }
61
62  urgent chan use[ProcessorID];
63   chan done[ProcessorID];
64
65  meta int [0,MAXC] C_T;
66  meta int E;
67  clock now;
```

Listing 2.7: TGS parametrized

```
1  E<> forall (i:TaskID) Task(i).Done and now <=50
2  inf{forall (i:TaskID) Task(i).Done and now <=50}:now,E
3  A[] not (forall (i:TaskID) Task(i).Done) imply not deadlock
4  E<> forall (i:TaskID) Task(i).Done
5  inf {forall (i:TaskID) Task(i).Done}:E
```

Listing 2.8: TGS parametrized query file

2.3 Clusters

Clusters consist of a number of computers networked together to provide some service that a single computer cannot efficiently deliver, such as high throughput and density storage, high performance computing or high availability services. This section will focus on clusters for high performance computing, and how to use them.

2.3.1 Cluster management

Several tools exist for managing cluster resources, such as the Terascale Open-Source Resource and QUEue Manager (TORQUE), Simple Linux Utility for Resource Management (SLURM), Oracle Grid Engine (OGE) and others. Their purpose is to manage jobs submitted by multiple users, and execute them on multiple resources, while allowing administrators to control the resources available to each user.

Despite these different systems, the focus of this section will be on the SLURM, as the Fyrkat cluster at our disposal utilizes it. Information in this section stems from [2, 20] as well as the man pages related to the SLURM commands.

SLURM is an open source resource manager, which enables users to execute jobs on a cluster. A SLURM job is essentially a reservation of resources for use by a specific user, at a specific time, accompanied by a number of *job steps*,where each job step describes a single parallel task, as well as any dependencies it may have to other job steps. A job step essentially contains the work to be done, by the nodes it is allocated to.

SLURM functions in a master-slave architecture [4, pp. 243-260], where one machine is designated the SLURM master, and the rest of the machines in the cluster are slaves. The master runs a daemon, i.e. a background process, named *slurmctld*, which monitors resources and jobs. Each slave runs an instance of *slurmd*, which executes jobs on the slave.

A number of auxiliary commands exists, which instructs the *slurmctld* daemon, or extracts information from it, and in some cases also the *slurmd* daemon. Some of these commands can be used to initiate jobs, such as the *srun* and the *sbatch*. While the *srun* command will interactively run a job or job step, *sbatch* will schedule a script for execution.

Both commands can take a number of arguments, such as specifying the number of cores or nodes needed, using the `--nodes=x-y`, where *x* describes the minimum number of nodes required for the job, and *y* describes the maximum number. Furthermore, environment variables, e.g. *PATH*, may also be set up by SLURM, before executing the job, by use of the `--export=list`, where *list* is a comma separated list of key-value pairs, specifying the variable names, and their appertaining values.

Once a job has been submitted via the *sbatch* command, the scheduler running in *slurmctld* will attempt to schedule the job as requested by the user. This means that it must wait for the necessary resources to be available, before running the job, which implies that if a user requests more resources that are available, the job can never be scheduled.

Furthermore, in order to handle multiple users and multiple jobs simultaneously, SLURM implements several queues, from which jobs are executed, if resources permit. The order of these queues depends on the scheduler loaded for SLURM, which can be either a built-in one, such as a First-In First-Out (FIFO), or use commercial schedulers such as the MOAB Cluster Suite.

2.3.2 Cluster storage

Storage on clusters can be managed in any number of ways. The most simple is to let each cluster node have its own storage, and let the jobs be responsible for moving the data around. This, however, is quite impractical, as each individual job must reimplement the data transfer. Therefore a common solution is preferred, in which only minimal concern for data placement is necessary in the job. SLURM provides an option for this, using the `sbcast` command, which ensures that all nodes allocated to the current job have access to the data. However, another solution could be to use a network file system, where the user is completely free from considering the location of the data, as all data is accessible from any node. This last option is what is employed on the *Fyrkat* cluster.

3 Design of UPPAAL PARMOS

The purpose of this chapter is to present the design of UPPAAL PARMOS — a PSA that distributes a parameter sweep of verifications of a UPPAAL model to multiple resources, while employing an optimization scheme to prioritize individual model configurations, thus allowing for faster access to desired results. The chapter starts with a profound description of the design principles, and then follows sections with deeper technical specifications that allows for later implementation.

3.1 Design principles

The construction of UPPAAL PARMOS is designed around a PSA. In order to design such a PSA, it is essential to understand and formulate the *interaction* between a user and UPPAAL PARMOS. The word interaction should be understood as; what does UPPAAL PARMOS require of knowledge in order to conduct a parameter sweep, and what knowledge is UPPAAL PARMOS able to provide the user with.

Given that the PSA is sweeping over a UPPAAL model, it is important that the user provides, not only the model but also the appertaining query file, to the PSA. However, this will only allow the PSA to run a verification of the model as it is specified by the user, thus more interaction is needed to establish a variation of different model configurations to verify. From Section 2.1 it is known that a UPPAAL model can contain both local and global declarations. These declarations can contain *variables* and a forthright idea is to use these variables to establish variation in a model.

As the variables in a model are statically typed, a range of different data types can be utilized as the type of a variable. A variable can also be marked as constant and it is also possible to specify different data containers such as structs and arrays. However, because full support for all data types and structures will require a complex interaction, and based on our view that most parameter sweeps will be initiated with the purpose to fit one or more integer variables, the variation of variables in our design is at first limited to integer variables that are declared globally. Yet this limitation is no greater than it is possible to preset the array and struct declarations, and then use integer variables to control what is read.

Task specification: To allow the user to specify the variation of variables, i.e. the parameter sweep, we establish a concept named task specification. The task specification, is to be viewed as an extension to a UPPAAL system, and contains information required for the PSA to vary a model configuration in the way that is desired by the user.

Task: While the task specification defines *how* the parameter sweep should be run, it do not include its related UPPAAL system. This design is chosen to keep the UPPAAL system as one unit, which at all times can be loaded using the UPPAAL Graphical User

Interface (GUI). Yet, for UPPAAL PARMOS to be able to link the task specification together with its related UPPAAL system we establish a concept named task.

For the user to be able to interact with UPPAAL PARMOS an interface is designed. While an idea to provide a mechanism that allows the user to transfer the files of a UPPAAL system as they are is rather forthright, it requires a profound design for a user-friendly way of letting the user specify the task specification.

Graphical User Interface: Since new complex concepts can be hard to cope for a user, we chose to design a GUI for allowing the user to construct the task specification in a visual environment. Yet, as the task specification can be viewed as a collection of instructions to UPPAAL PARMOS, which potentially can be reused e.g. in case of minor changes, we design the content of the task specification to be stored within a single file. This file can then be constructed, changed and stored locally. The front-end design will then provide the user with two options:

1. Enter information about the parameter sweep through a GUI, which will then generate the task specification based on the input.
2. Allow the user to transfer the task specification in a single file.

The GUI will not only provide ease of use, it will also often result in a faster construction of the task specification. However, this could vary from one task to another, and therefore the choice between the two options is left to the user.

Besides the possibility to add a new parameter sweep task, the GUI also provides the possibility for accessing the results obtained on the basis of the verifications runs.

The ability for the user to specify *where* the parameter sweep should run, is not included in the task specification. This is because that the specification of *how* a parameter sweep should run, is independent of where it should run. The user therefore need a way of informing UPPAAL PARMOS where the verification processes should distributed to.

Resource: A location, where verification processes can be distributed to and then executed, is named a resource. Different types of resources such as clusters, grids or a single personal computer are valid as long as:

1. UPPAAL PARMOS is able to transfer and retrieve model configurations from the resource.
2. The UPPAAL verifier is able to run on the resource.

For a user to utilize such different types of resources, we define a concept named *resource proxy*. A resource proxy is a proxy that is inserted between UPPAAL PARMOS and a resource. This will remove the heterogeneity that exists between the interface of an unknown resource and UPPAAL PARMOS, thus allowing UPPAAL PARMOS to utilize resources whose interfaces are unknown at design time. A resource proxy is viewed as an extension to UPPAAL PARMOS and users will be able to develop their own customized ones.

In order to have UPPAAL PARMOS exploit the resources added by the user, it is important that the system is designed to be *scalable*. This property reflects that UPPAAL PARMOS should remain efficient and run effectively when there is an increase in the amount of resources utilized [7, p. 19].

Multiple challenges arise when designing a scalable distributed system [7, pp. 19-21]. The challenge of *controlling the cost of physical resources* arise, if the system does not utilize additional supplied resources in a proportional scale. If UPPAAL PARMOS is able to run 1000 verifications per minute on one resource, then it should be able to run 2000 verifications on two equal resources, if and only if all model configurations on both resource take an equal amount of time to run.

To cope with the issue of *controlling the performance loss* it is profound to utilize hierarchic communication structures found in e.g. cluster resources, rather than a linear communication structure of a single personal computer. While an increase of data sent will result in performance loss for both structures, a hierarchically structured resource will at best have a communication complexity of $O(\log\ n)$, where a linearly structured resource will at best have one of $O(n)$. This design allows UPPAAL PARMOS to communicate once with the resource manager of a cluster, and then have the resource manager communicate further to its nodes.

Although the execution of verification processes are distributed to multiple resources, controlling this distribution is centralized in the PSA. It is therefore important to design this control such that it *avoids performance bottlenecks* throughout its entire utilization.

Such performance bottlenecks can arise multiple places in the PSA, yet areas such as computation of new model configurations and storage of verification results are obvious.

Storage: Once a task has been sent to UPPAAL PARMOS, it needs to be stored safely such that it can be accessed by UPPAAL PARMOS, both throughout the verification of the task and afterwards. Additionally, all the different model configurations generated from the task specification also need to be stored together with the computed result of their verification.

Since UPPAAL PARMOS is utilizing resource proxies for communicating with resources, storage for these resource proxies is also designed. Given the design of these resource proxies, it is possible to employ multiple instances of each resource proxy. Thus the storage must be designed such that different informations can be related to each instance.

While storage for tasks, model configurations and resource proxies can be designed in multiple ways, it limits the different design options, when considering that the data must be stored such that the user is able to access it both under and after the execution of the verification processes. Given that the user accesses the data through a front-end, this front-end needs to interact with the storage in a way, in which the data presented, is consistent and yield a useful result for the user.

Failure model: When designing a distributed system such as UPPAAL PARMOS it is important to consider a failure model. The failure model is defining the faults of the system

that must be handled. These faults can arise both at UPPAAL PARMOS, at resources or in the communication channels between them [7, pp. 53-57].

A fault that must be detectable, is the crash of an established connection to a resource. This can be detected by incrementing a timer e.g. *Timer A*, which is started when data is sent from UPPAAL PARMOS to a resource, and stopped when response from the resource is received. After *Timer A* is incremented, the value of it is checked and if the value exceed a specified value, e.g. 10 seconds, a time-out signal is thrown. After a time-out, the resource should be removed from the list of resources known by UPPAAL PARMOS. Other resources should then start the verification process of those model configurations that were in the process of being verified on the crashed resource.

The failure model also need to consider the topic of *fault tolerance*. It is important that those data that are available, both to the user and the system, are correct and consistent even after a fault has occurred.

Architecture: Several aspect are considered in the choosing of an architecture for UPPAAL PARMOS. The first consideration is that of the work, which UPPAAL PARMOS has to execute. This work includes the executing of a task, which consist of running multiple verification processes on remote resources. The design of UPPAAL PARMOS is to reflect the fact that, given these verification processes, the only way of knowing the running time of these processes, is to run them and measure the time. Therefore, UPPAAL PARMOS is not to require that the user needs be active once the task has been received.

To cope with this a Service-Oriented Architecture (SOA) for UPPAAL PARMOS is chosen. SOA is a commonly recognized design principle, utilized for building scientific complex applications [10]. A SOA expose network accessible programming interfaces, and the communication between a service and the clients using it, is done through standardized protocols and standardized content encoding schemes [13, 30, 41, 42] [7, pp. 14-15]. A SOA will allow the user to add a task, then disconnect and reconnect whenever desired.

Since we have chosen to design a GUI for interacting with UPPAAL PARMOS, the design of this GUI is also taken into consideration. The chosen service structure can be viewed as having a front-end and a back-end, where the back-end is the PSA and appertaining storage. The design of the front-end, consist not only the GUI, but also of a *web service* that the GUI communicates to UPPAAL PARMOS through. This design is chosen so that users can develop their own customized user interfaces, if the GUI provides insufficient features.

Web service: In order for the GUI to have access to UPPAAL PARMOS, we have chosen to design it as web service, which serves as a proxy between the user and UPPAAL PARMOS. This set-up, with a web service between the front-end and the back-end, gives benefits both in terms of authorization and authentication, and also in terms of providing the user with access to common procedures that can be implemented into the web service. These procedures can involve parsing and mapping of data. This

choice of a proxy also provides an easy way of specifying what the user is allowed to access in the data storage, instead of giving the user full access to the data storage.

The utilization of a web service allows both web based and non-web based applications to interact with UPPAAL PARMOS. The only requirement is that the application supports standardized protocols and standardized content encodings for communicating with the web service.

Our architectural software design of UPPAAL PARMOS is illustrated in Figure 3.1. The light grey areas illustrates the four main parts of UPPAAL PARMOS, the dark grey squares with rounded corners illustrates different modules in the parts and the arrows illustrate the interaction between these modules. The design of the GUI is chosen to be web based, which will provide a speedy deployment of UPPAAL PARMOS, since no software other than a web browser need to be installed on the user's computer.

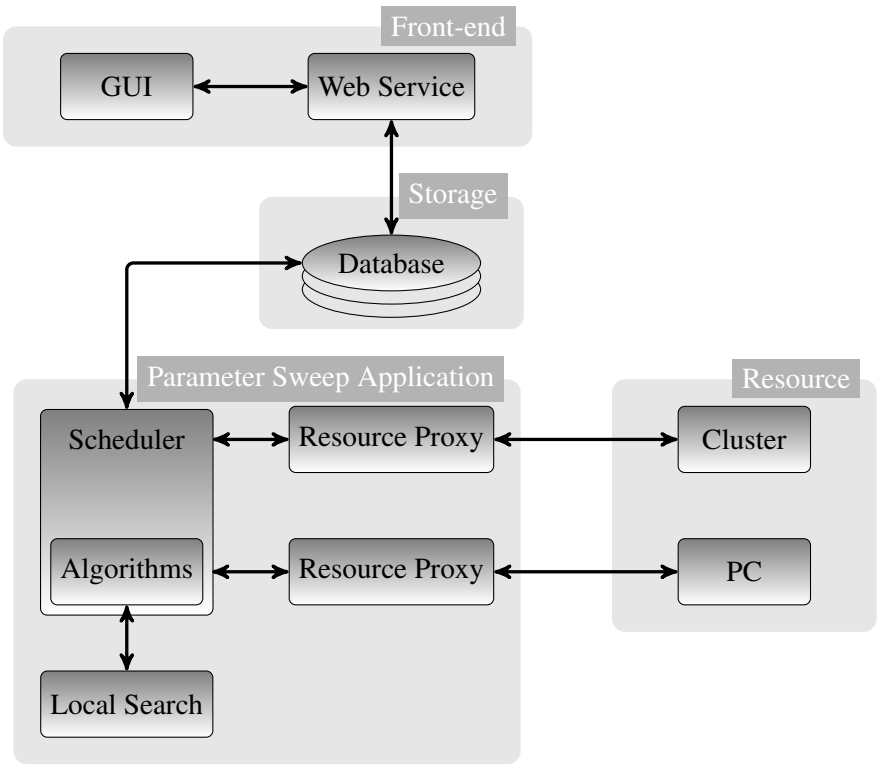


Figure 3.1: Illustrates the four parts of UPPAAL PARMOS. It exemplifies an instance of UPPAAL PARMOS with the PSA employing a Local Search optimization algorithm and distributing the verification of models to two resources, specifically a cluster and a single personal computer.

As illustrated in Figure 3.1 the front-end is decoupled from the back-end i.e. PSA, Storage and Resource, thus providing a separation of concerns. The figure illustrates the point, that the PSA is a centralized part of the UPPAAL PARMOS architecture.

Parameter Sweep Application: The PSA is responsible for conducting the parameter sweep. Once a user submits a task to UPPAAL PARMOS, it is through the Web Service placed in the storage. The PSA is periodically checking the storage to locate new task that are ready to be processed.

As illustrated in Figure 3.1 the PSA contains a module called Algorithms. This module is responsible for loading and initializing of the optimization algorithm that the user has specified in the task specification. The PSA will then utilize the loaded algorithm to compute model configuration in a way that is desired by the user.

After a model configuration has been created, it is wrapped together with task information and shipped to a resource. The PSA then periodically checks the resource to discover whether the verification process has ended. If it has, it locates the produced verification results, and transmit them back to the PSA for parsing. Once a received result has been parsed, the algorithm is notified of the result, such that it can produce new model configurations based on the received results. After this the received result is stored together with its model configuration, such that the user can access it.

In order to avoid a design of the PSA that is locked to a fixed optimization algorithm or some specific types of resources, we have chosen to extend the SOA with component-based principles. In addition to allowing customization of algorithms and resource proxies, the component-based principles will also provide a desired separation of concerns between resource proxies, algorithms and the PSA.

The modular separation of the system will yield the opportunity of delegating development resources to areas that are essential for this thesis, and to settle for a minimum requirement in other areas. The modular separation is an essential property since this thesis focuses mainly on utilizing algorithms to minimize the time consumed before the user can receive desired results.

This completes the design principles of UPPAAL PARMOS. Based on this, deeper specifications of the UPPAAL PARMOS architecture is made. These specifications are found in the following sections *Parameter Sweep Application*, *Task specification*, *Storage*, *Graphical User Interface* and *Web service*.

3.2 Parameter Sweep Application

The PSA is the part of the UPPAAL PARMOS architecture, which is responsible for conducting the parameter sweep. As illustrated in Figure 3.1 the PSA is a centralized component, thus the design of it must be profound in order to avoid performance loss when conduction parameter sweeps.

From Section 3.1 it is known that there are two main responsibilities of the PSA, these are the *scheduling* of model verifications on resources and the *retrieval* of the results produced by these verifications. The two responsibilities have, in their mode of operation, two different behaviours. While scheduling can be designed as a state-based system, which changes state based on events that are raised throughout the processing of a task, it is not profound to use this design for the retrieval responsibility.

The success of a retrieval operation is determined by the number of verification results retrieved. If the number of results is greater then zero, then the retrieval operation was a success, otherwise it was not. To ensure that a retrieval operation always succeed, the retrieval system is designed to utilize time-based events. This will allow a dynamic behaviour, where the retrieval system is able to adjust the execution of the retrieval operation. This adjustment is designed to be based on running time statistics gathered from previous run. However, it shall be noted, that there does not exist a generalized relationship between these running times, thus the adjustment value will only be an estimate.

It is now clear that the PSA is handling two diverse responsibilities, i.e. scheduling and retrieval, which requires concurrent execution if each of their behaviour is to be respected.

Before scheduling and retrieval are presented, two components, which provides expandability to the PSA, are described. These two components are called *Resource proxy* and *Algorithm* and provides key functionality, without which the PSA could not function. The components will enable users to develop their own customized solutions for the optimization scheme employed in a parameter sweep, and to specify a proxy between UPPAAL PARMOS and an arbitrary remote resource.

3.2.1 Algorithm

One of most essential parts of the PSA, is the decision making process where the next model configuration to verify is chosen. We name this part *scheduling* and the decision making occurs inside an external component of the PSA, named *algorithm*. This scheduling design is important, because it allows the user to choose a scheduling algorithm that is appropriate for the given task. Only the user can have knowledge about the behaviour of a task, and that knowledge can be utilized, to gain desired verification results faster, by choosing or developing an appropriate algorithm.

The algorithms that are utilized by the PSA are known as optimization algorithms. These algorithms are based on optimization principles found in mathematics, where the objective is to find the maximum or minimum of some function. It is important to understand that optimization algorithms do not lower the running-time of a full parameter sweep, they simply try to limit the time elapsed before a desireably result is obtained. A thorough study of

these types of algorithms and optimization principles is found in Chapter 4. The rest of this section is devoted to how an algorithm should be structured and how it should interact with the PSA.

It is clear that the PSA need to conduct multiple concurrently running verifications. To make this possible, it is important that the algorithm design does not stipulate that the currently running verification need to be finished, before another can be started. Consequently, we chose to view the algorithm and the PSA as having a *client-server* relationship, where the PSA is, at all times, able to request a new model configuration. This relationship is illustrated in Figure 3.2 where the PSA is concurrently requesting new model configurations.

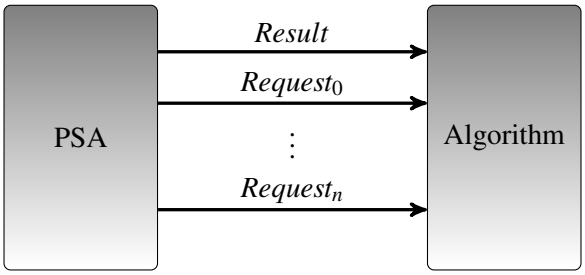


Figure 3.2: An illustration of the client-server relationship between an Algorithm and the PSA. Where *Request*$_0$, ..., *Request*$_n$ depicts a concurrent requesting of model configurations from the Algorithm and *Result* depicts the return of a computed verification result.

The design, as illustrated in Figure 3.2, allows the PSA to control all calls between it and the algorithm. This is preferable since it allows the the algorithm to be disjointed from other processes running in the PSA, and function merely as a pluggable component.

We chose that the PSA should be able to utilize algorithms developed by the user or an algorithm designer. For this to be possible we decided to design a contract for the interaction between the PSA and the algorithm. This will require algorithms to provide a programming interface that the PSA knows how to use, and also provide the algorithm designer with knowledge of how an algorithm receives information from the PSA. We have designed the programming interface to contain the following procedures:

GetModelConfiguration: This procedure is executed the by the PSA, once a resource is ready to start a new verification. The algorithm is then to return the model configuration, which it has decided needs verification.

AddVerificationResult: This procedure is executed by the PSA, once a verification result has been computed and received from a resource. It gives the model configuration and appertaining verification result to the algorithm instance.

GetAlgorithmSettings: Algorithms are task independent, and possibly not developed by the user. Thus we design the programming interface to contain a procedure named GetAlgorithmSettings, which, when executed, is to return a list of settings, available to the algorithm. This feature is designed with the purpose of providing embedded algorithm information to the user, e.g. when the algorithm selection is made at the GUI.

SetInitialCondition: This procedure is called just after the algorithm has been loaded into the PSA. It gives the task specification and user specified settings to the algorithm instance.

The programming interface is designed such that a minimum of interaction between the algorithm and the PSA is obtained. This is important since this interaction could become a bottleneck when the PSA is conduction very large sweeps. Furthermore, the implementation of the programming interface, need to support that the PSA can instantiate multiple instances of an algorithm concurrently. This scenario is possible if the PSA is conducting a scheduling scheme with multiple tasks.

3.2.2 Resource proxy

In order to have a model configuration verified, a computing resource is needed by the PSA. As we have decided that the PSA should be extensible with regard to support of resources, and that multiple disparate resource types exists, we have chosen to design a *resource proxy*, which the user can utilize to create new resource proxies with minimal effort.

As illustrated in Figure 3.3 we have designed the PSA to utilize multiple resources concurrently, and also to have the PSA ship multiple verifications to the same resource. Concurrently utilized resources can be necessary, if the amount of verifications in a task is very large. Multiple verification at a single resource can also be needed, if a resource has multiple computing cores available, e.g. if the resource is a multi-core personal computer, a grid or a cluster.

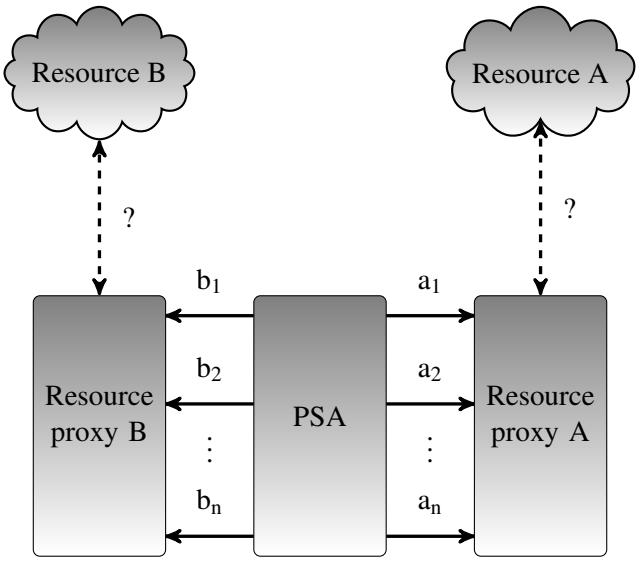


Figure 3.3: An illustration of the relationship between a Resource proxy and the PSA. Where a_1, ..., a_n and b_1,...,b_n depicts a concurrent requesting of model configuration verifications from the PSA to multiple Resource proxies.

For this to be possible, we have chosen to design a contract, stating the necessary interaction between the PSA and the resource proxy. This contract requires a resource proxy to provide a programming interface that the PSA knows how to use. We have designed the programming interface to contain the following procedures:

Initialize: Since multiple resource proxies can be instantiated from a single resource proxy component, the user should be able to specify settings of the resource proxy. Thus, the PSA executes this procedure, just after the resource proxy has been loaded.

ProcessVerirfication: This procedure is called by the PSA to command the resource proxy to execute a verification. It gives the model configuration to verify as argument.

TransferFiles: This procedure is called by the PSA to request transfer of local files to the resources. It gives the files that should be transferred as argument.

RetrieveVerificationResults: This procedure is called by the PSA to notify the resource proxy, that it should start retrieving verification results from the resource.

NoMoreVerificationsToProcess: This procedure is called by the PSA to notify the resource proxy, that no more verifications of the current task is needed, and if the resource proxy has buffered verifications stored, it should flush them i.e. ship them to the resource.

CleanUp: This procedure is called by the PSA to notify the resource proxy, that the files that were transferred should be removed from the resource. It gives the files to be removed as argument.

Status: This procedure is called by the PSA, when it needs to be notified about the status of the resource. The resource proxy should change status depending of whether it has crashed, is running, is connecting to its resource or disconnected from its resource.

CrashRestore: This procedure is called by the PSA in order to notify the resource proxy, that model configurations were found in the storage.

The programming interface of the resource proxy is designed such that most resources can be utilized. This is important, since the resources that the PSA is to utilize in future parameter sweeps, are unknown at design time. We could have chose to allow the user to utilize a limited types of resources, yet this would result in limited usefulness of UPPAAL PARMOS.

While both algorithm and resource proxy are components of the PSA, only the algorithm can have settings that are task dependent. Therefore, the list of available resource proxy instances and their appertaining settings need to be stored. For this we design a data model, which is found in Section 3.4. This will allow the created resource proxy instances to be stored at UPPAAL PARMOS, thus to be used by all submitted tasks.

3.2.3 Scheduling

The Scheduler is the part of the PSA, responsible for the mapping of model configurations generated by the Algorithm onto Resource proxies as well as passing results from said resources back to the Algorithm.

These two parts are designed as two distinct systems: the *Scheduler* and the *Retriever* It is the Scheduler that is responsible for fetching a task to process from the Storage, requesting new model configurations from the Algorithm, distributing the verification processes to resources via Resource proxies and storing the model configurations in the Storage. The Scheduler is designed as a sequential task scheduler, i.e. it will complete the run of a single task before moving on to a new task, as depicted in Figure 3.4.

Before the Scheduler can begin to schedule jobs, it will perform some initialization, before going into the main loop, where it will continually attempt to load new tasks, if the Storage contains any, and start processing them.

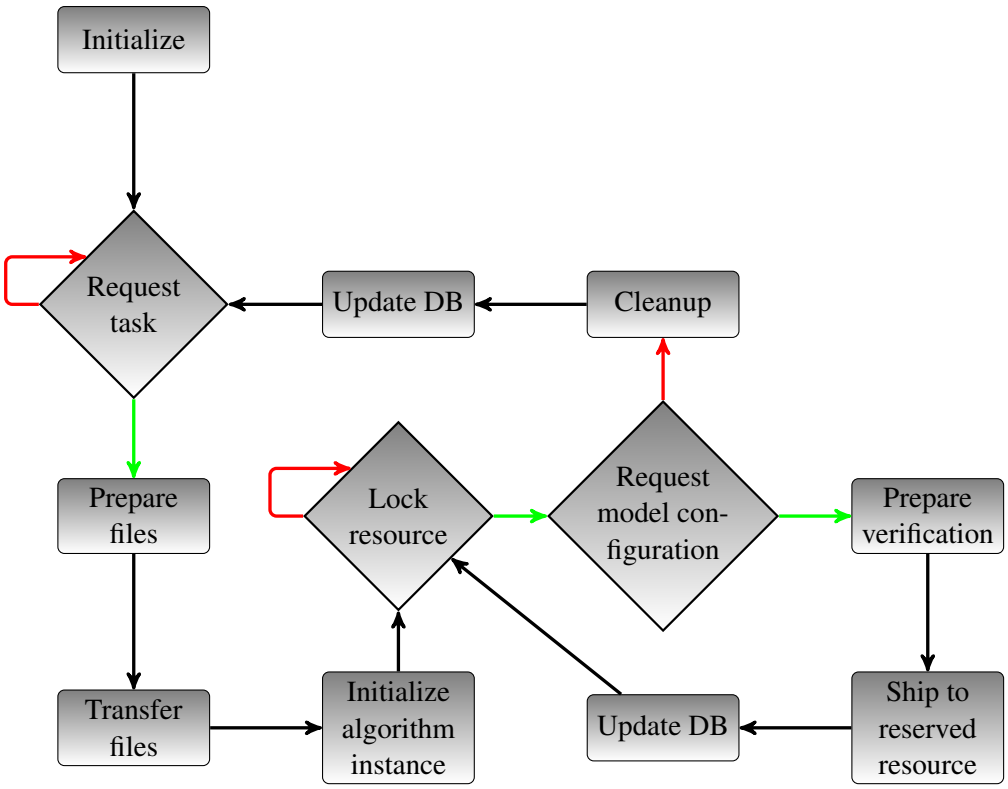


Figure 3.4: Depicts the flow of the Scheduler. Decisions are denoted with a diamond and processing steps are denoted with rounded squares. While all arrows denote the flow, a red arrow from a diamond also denotes "failure" and a green arrow denotes "success."

In order to prepare the task for processing, the UPPAAL system files, i.e. the model and query files, must be available on all resources. However, in order to avoid a continuous transfer of model files with only minimal change in content, the Scheduler implements a scheme to cope with this, thus removing overhead caused by continuous file transfers. This scheme is illustrated in Figure 3.5, where the Scheduler, after having extracted the UPPAAL system from the Storage, splits the model into two pieces, and instructs the Resource proxies to transfer both pieces, formatted as two files, together with the query file.

This will allow the resource proxies to command their resource to create a new model file by merging a model configuration in between the two model file pieces already existing at the resource.

Next, the Scheduler is initializing the optimization Algorithm, which decides which model configurations to run. Once the Algorithm is loaded, the Scheduler enters the main loop, in which it attempts to lock a free resource, before requesting a new model configuration from the Algorithm. If no configuration can be obtained from the Algorithm, it has decided that no further verifications are necessary, and the Scheduler should end the task. If a model configuration is obtained, it will prepare it for dispatching to the resource, such as assigning it a unique id. Once the preparations are done, the model is shipped to the locked resource, and the model configuration information will be added to the Storage, and the main loop start over.

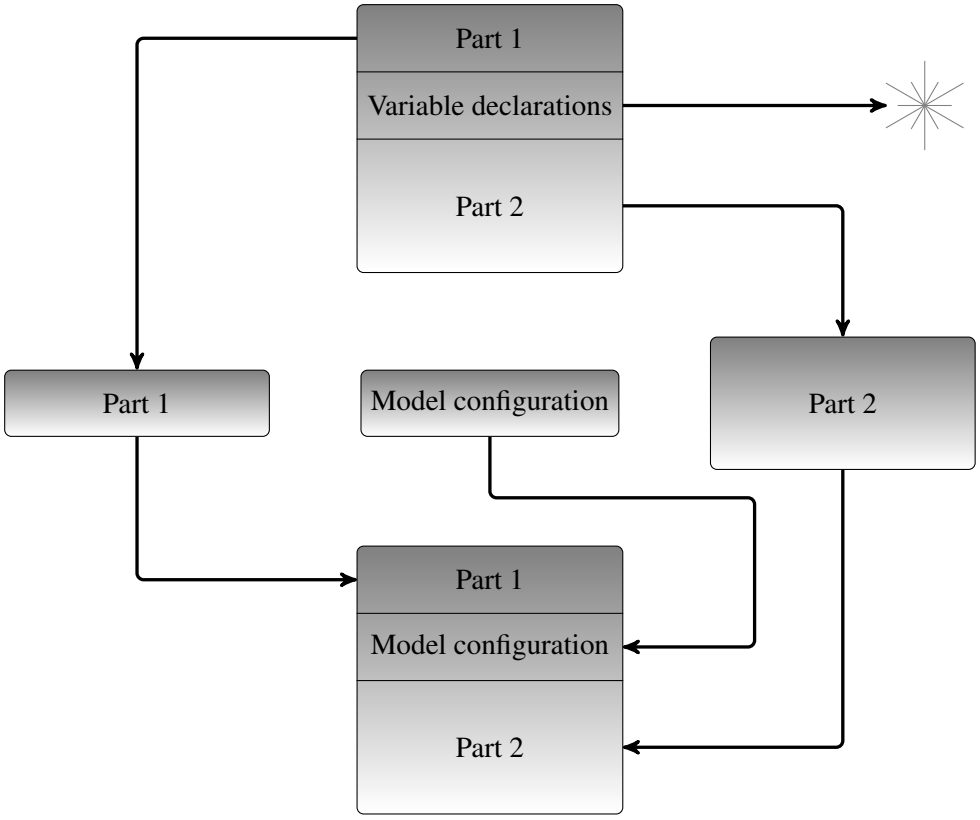


Figure 3.5: The splitting of a UPPAAL model, and the assimilation of model configurations

Once the Algorithm reports that no more configurations are necessary, the Scheduler will clean up the state of the task, including waiting for, and fetching, the results of any unfinished verifications, as which point it will commit the final update for the task to the database, before going back to check for new tasks.

If no new tasks are ready in the Storage, the Scheduler will merely sleep and continue to try requesting a new task.

3.2.4 Retrieval

The control flow in the Service handles all events related to the distribution of a parameter sweep. However, while the flow employed by the Service exhibit a time independent control flow i.e. it is event based, the process of retrieving results from resources exhibits a dynamic behaviour. This dynamic behaviour is found in the variations of execution time of the model verifiers, which employs different model configurations. This excludes an event based controlled process for retrieving results, if unnecessary retrieves are to be avoided.

The retrieval of job results is unquestionably a necessary process and it is a common duty of all resource proxies. In order to release the resource proxies of the burden of managing the retrieval frequency, a *retrieval scheme* is designed. This scheme provides the resource proxies with a simple notification at the times when execution of the retrieval is necessary.

In order to design a retrieval scheme it is necessary to, at first, establish the retrieval flow. This flow of requesting, transmitting, parsing, storing and notification of job results is designed as illustrated on Figure 3.6.

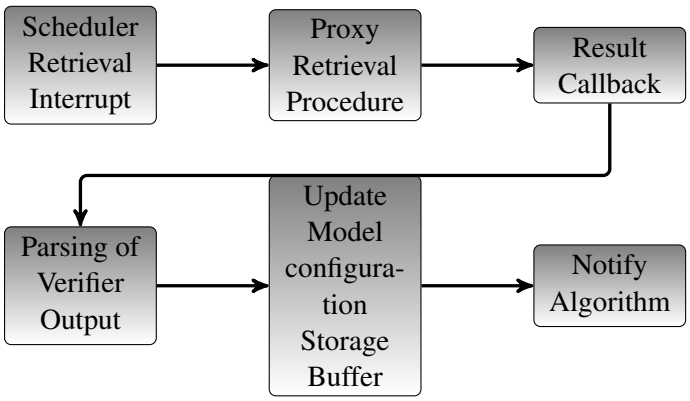


Figure 3.6: Illustrates the flow of the retrieval process.

Feedback

Each retrieval interrupt requires the use of one or more network connections and the attention of the Service, which is interrupted and forced to prioritize other procedures than scheduling. It is therefore desirable to minimize the amount of retrieval interrupts to an absolute necessary minimum. This section addresses this issue by suggesting a solution, which involves the use of concept of a feedback loop from control theory [11].

Figure 3.7 illustrates a basic feedback loop that consist of a *Plant*, a *Controller* and a *Feed-*

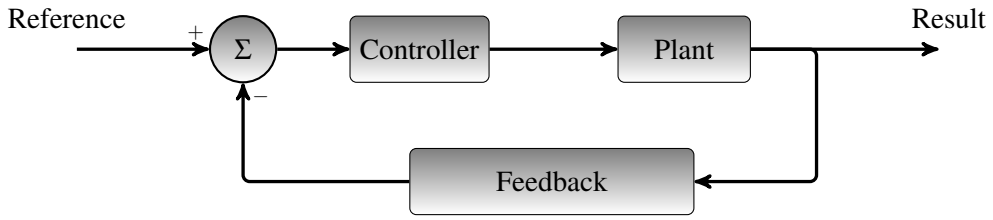


Figure 3.7: Illustrates a feedback loop with Plant, Controller and Feedback.

back. The Σ process on Figure 3.7 is a summation of the *Reference* value and the feedback value. Where the *Plant* is the system that is affected by the input from the *Controller*, which manipulates the summation in order to obtain a desired result. The *Feedback* is a measured value, which has been changed, based on the output value of the system.

The idea is to view the verification processes, on a resource, as a dynamical system, which, with different frequencies, produces results. The challenge is to predict this frequency, such that the resource is notified a minimum number of times. This idea is illustrated in Figure 3.8 where the idea is abstracted onto the feedback system of Figure 3.7.

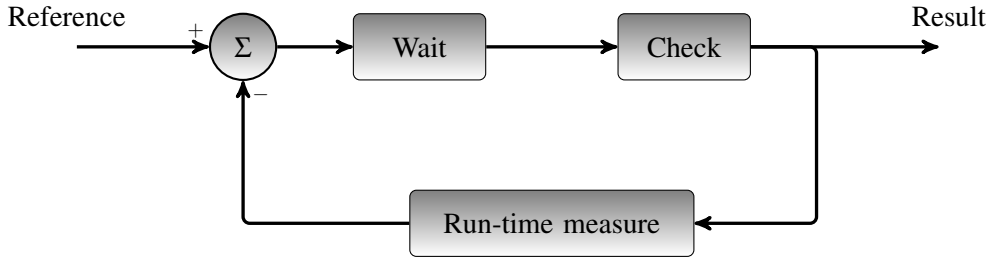


Figure 3.8: Illustrates the idea of utilizing a feedback loop for minimizing the frequency of retrieval interrupt procedure calls.

In Figure 3.8 a *Wait* procedure that implements a delay between retrievals, serve as the Controller of the feedback system, where the Plant is the *Check* procedure. The feedback value shall be found by having the resources stamp the job results with the running time of the verifier before they are passed on to the scheduler.

3.3 Task specification

As described earlier in the design principles of Section 3.1, the parameter sweep need only to be over integers. Thus a superset can be defined in the form of a hyperrectangle, i.e. defined using a lower and upper bound on each of the parameters. Given this, we have chosen to design a concept named *Parameter*. A Parameter is designed to contain a name, a lower bound value, an upper bound value and an increment value to state by which value to step towards the upper bound. The name contained in the Parameter, is the name of the integer variable in the UPPAAL model, which the user want to sweep over.

To limit the parameter set, we have chosen to design a concept named *Constraint*, which will reduce the superset to the actual necessary parameter space. A Constraint is designed

to contain a boolean expression which must be true, for the PSA to accept the current set of Parameters.

In order for the PSA to know which solutions are desired by the user, we have defined a concept named *Objective*. The user can specify two different types of objectives:

1. *Simple Objective* which defines a requirement which must be fulfilled.
2. *Optimization Objective* which directs the parameter sweep, towards a desired result.

Both in the case of the constraints on the parameter space, and in the objectives, having the ability to describe these using expressions is necessary. The expressions should handle not only arithmetic expressions, but also boolean expressions and comparisons. A Constraint is designed to contain a boolean expression, although, by use of comparison expressions, it can contain arithmetic expressions. In order to make the expressions more useful, we have chosen that the user should be able to access not only the value of the current parameters, but also those of the results returned by UPPAAL, both whether a query was satisfied, and the numeric result of any *inf* and *sup* query.

In order for the PSA to direct the sweep in the direction of better results, an optimization algorithm must also be specified, since different algorithms may behave differently, and only the user can know which one suits a specific model. Thus, we have designed the Task Specification to also contain information on the Algorithm to use, as well as any settings appertaining the Algorithm.

3.4 Storage

As stated in the desing principles of 3.1, UPPAAL PARMOS requires a location to store information about resource proxies, tasks, model configurations and their results.

We have decided to utilize using a database-management system, which gives an convenient and efficient way of handling data [35]. The database provides a range of benefits that a specialized storage system would not be able to match, given the evolution of modern database systems, without extensive development effort. These benefits includes properties such as atomicity, consistency, isolation and durability that all serve to guarantee that database transactions are processed correctly. Furthermore, we also decided that the database need to support relational models, which allows the database to be modelled as collection of entities and relationships between them. Another decision is to use a database supporting a data-definition language, for handling database schemas, and a data-manipulation language to handle data addition, removal and changes.

3.4.1 Entity-Relationship Model

In order to produce a database design, we chose to create a Entity-Relationship Model (ERM) of each of the elements that we decided should be stored. The ERM allows objects to be modelled using entities and relationships between them. The ERM uses four symbols for representing different concepts, these are illustrated in Figure 3.9.

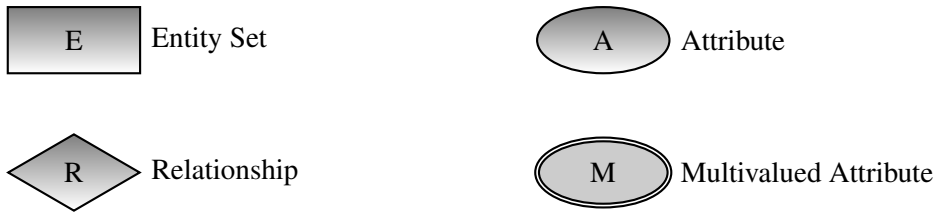


Figure 3.9: Illustrates the ERM concepts used for modelling the storage.

The Entity Set, Attribute and Multivalued Attribute of Figure 3.9 all represents objects, yet with minor differences. The Entity Set can be viewed as a group of objects, the Attribute as a single data object that belongs to a group and Multivalued Attributes as containing multiple instances of the same data object. It is also possible to group Attributes together as children with another parent Attribute. Furthermore, relations between Entity sets can be expressed using the Relationship symbol, which links the individual entities together

The required database design for UPPAAL PARMOS is modelled using three entities as illustrated in Figure 3.11, 3.12, 3.13 and a relationship between them as is illustrated in Figure 3.10.

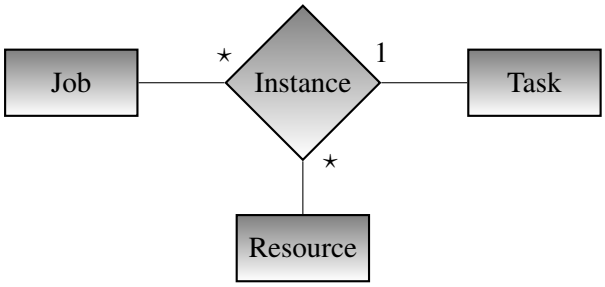


Figure 3.10: Illustrates of the relationship between the Task, VerificationResults and Resource entities.

The relationship illustrated in Figure 3.10 states that a UPPAAL PARMOS instance will include a single task, multiple verification results and multple resources.

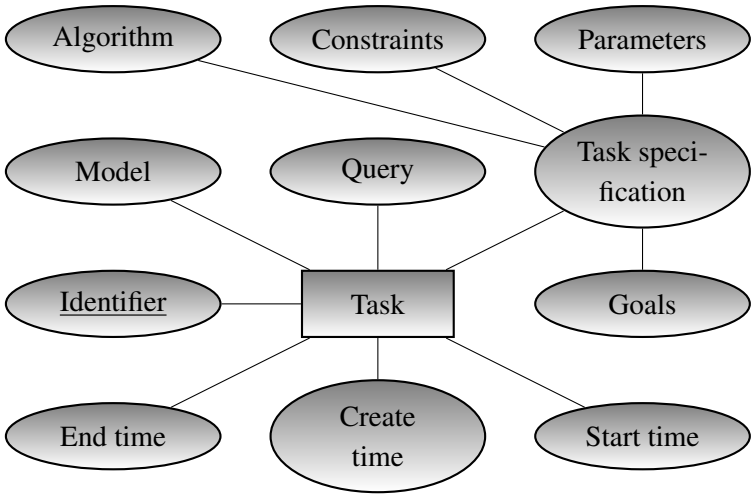


Figure 3.11: Illustrates the Task entity, and appertaining attributes.

As stated in Section 3.3, a task must contain both a model and a query. Furthermore, in order for the PSA to know what parameter space to iterate over, and how to optimize the search, it must also contain a task specification. Besides this, it also contains certain timing statistics, as well as an identifier, to uniquely identify a specific task.

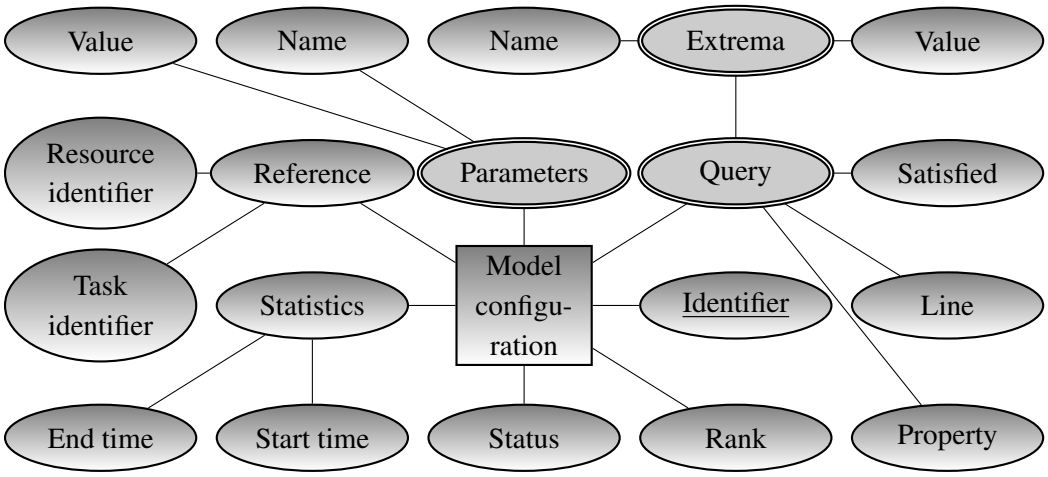


Figure 3.12: Illustrates the Model configuration entity, and appertaining attributes.

The Model configuration, representing a single configuration taken from the parameter space of the Task, contains the names and apertaining values of the parameters. It also contains the results of its verification, split up into multiple attributes:

Property: Stores the property number, as determined by its appearance in the UPPAAL query file.

Line: The line in which the query appears in the UPPAAL query file.

Satisfied: Whether the query was verified or not.

Extrema: A number of results obtained from an *inf* or *sup* query, containing both the name and values.

Furthermore, the Module configuration also contains the status of the verification, whether it is pending on the resource, running or finished, as well as the Rank of the task, which indicates whether it is the best job found so far. Some statistics are also collected from the task, specifically Start time and End time, which are also stored. Lastly, a reference to the Resource and Task pertaining to the Model configuration must also be saved, by storing their unique identifiers.

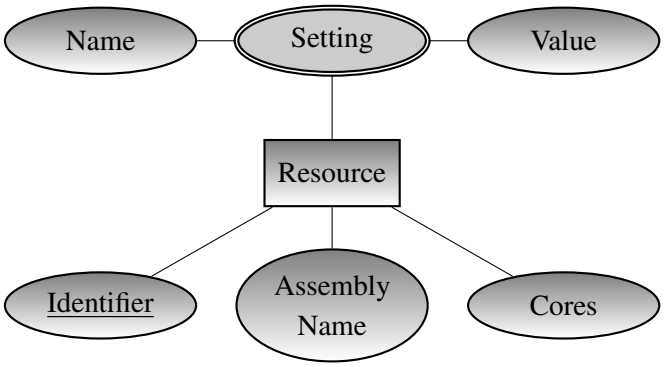


Figure 3.13: Illustrates the Resource entity, and appertaining attributes.

The Resource consists of the name of the assembly pertaining to the resource, along with a number of settings, each containing the name of the setting and the appertaining value. Furthermore, the Resource also contains the number of cores it has available, as well as an identifier, to uniquely identify the Resource among other resources

3.5 Web service

As stated in the design principles of Section 3.1, we have chosen to design a web service for providing a programming interface for the user to access UPPAAL PARMOS. This design will provide a prudent separation between a user a running instance of UPPAAL PARMOS. While we in this thesis does not focus on security in terms of authorization and authentication, this design with a proxy between the user and UPPAAL PARMOS can be necessary in future versions of UPPAAL PARMOS.

We have designed the web service to allow the users of UPPAAL PARMOS to develop their own GUI for managing their tasks. The web service is designed to provide not only accessors and mutators to elements in the Storage, but also to provide the user with ready-to-use procedures such as mappers and parsers. This will allow the users to develop their own semi complex user interface, by using library procedures.

We have designed the web service to include the following procedures:

AddModelQuery: Adds a UPPAAL system to the storage of the PSA. This forms the first of two parts required to form a task. The task will first be complete, i.e. active, once a task specification has been added. This procedure takes the content of a model and query as argument.

GetTasks: Returns a list of tasks located in the storage of the PSA. This procedure accepts arguments to filter on the running, completed or pending status of the tasks, and a range to select a subset of tasks to return. Another argument is used to sort the tasks based on status or running statistics.

GetTaskInformation: Returns the status and statistics for a task. This procedure takes the identifier of a task as argument.

GetTaskModel: Returns the model of a task. This procedure takes the identifier of a task as argument.

SetTaskModel: Sets the model of a task. This procedure takes the identifier of a task as argument.

GetTaskQuery: Returns the query of a task. This procedure takes the identifier of a task as argument.

SetTaskQuery: Sets the query of a task. This procedure takes the identifier of a task as argument.

GetTaskSpecification: Returns the entire task specification of a task. This procedure takes the identifier of a task as argument.

SetTaskSpecification: Sets the entire task specification of a task. This procedure takes the identifier of a task as argument.

GetTaskSpecificationParameters: Returns the parameters located in the task specification of a task. This procedure takes the identifier of a task as argument.

SetTaskSpecificationParameters: Sets the parameters located in the task specification of a task. This procedure takes the identifier of a task as argument.

GetTaskSpecificationObjectives: Returns the objectives located in the task specification of a task. This procedure takes the identifier of a task as argument.

SetTaskSpecificationObjectives: Sets the objectives located in the task specification of a task. This procedure takes the identifier of a task as argument.

GetTaskSpecificationConstraints: Returns the constraints located in the task specification of a task. This procedure takes the identifier of a task as argument.

SetTaskSpecificationConstraints: Sets the constraints located in the task specification of a task. This procedure takes the identifier of a task as argument.

GetTaskSpecificationAlgorithm: Returns the optimization algorithm and its appertaining settings located in the task specification of a task. This procedure takes the identifier of a task as argument.

SetTaskSpecificationAlgorithm: Sets the optimization algorithm and its appertaining settings located in the task specification of a task. This procedure takes the identifier of a task as argument.

GetVerificationResultMatrix: Returns a list of verification results appertaining a task. This procedure takes the identifier of a task as argument or an identifier of a verification result. Another argument is used to filter by rank, input values or output values and another is used to sort said options.

Get2DGraphResultValues: Used to extract values from the verification results such that they can be applied to a two-dimensional graph. This procedure takes the identifier of a task as argument, together with the name of a parameter to vary and a list with the names of those parameters to keep fixed. This procedure accepts arguments to filter on the running, completed or pending status of the tasks, and a range to select a subset of tasks to return. Another argument is used to sort the tasks based on status or running statistics.

Get3DGraphResultValues: Used to extract values from the verification results such that they can be applied to a tree-dimensional graph. This procedure takes the identifier of a task as argument, together with the name of two parameters to vary and a list with the names of those parameters to keep fixed. This procedure accepts arguments to filter on the running, completed or pending status of the tasks, and a range to select a subset of tasks to return. Another argument is used to sort the tasks based on status or running statistics.

3.6 Graphical User Interface

While the GUI is not the focus of this thesis, it still gives rise to a requirement of an interface, rich enough to fulfil the minimum needed functionally for managning a parameter sweep. In the design principles of Section 3.1, we stated the decision of a web-based GUI for managing tasks. A web-based application gives the benefit that no other software than a web browser needs to be installed at the users computer. The GUI has been designed to provide two options; the *submitting* of a task, and the *presentation* of the verification results.

3.6.1 Submitting

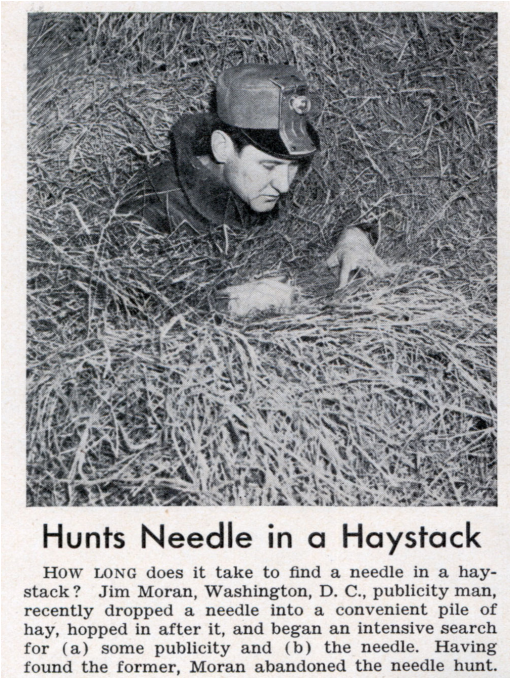
In order to create a task, the user needs to submit the UPPAAL system, i.e. model and query file, which the parameter sweep is to be conducted over. In order to complete the task, the user also needs to submit the task specification describing the parameter sweep to be conducted. The design principles states that the user should be able to choose between submitting a task specification as a file or construting one using the GUI. If the last option is chosen, a range of boxes and appertaining help text are displayed on GUI in order to guide the user through the process of creating a task specification.

3.6.2 Presentation

As soon as the first model verification result is stored in UPPAAL PARMOS, the GUI is updated to show this and other results. The user is provided with options for having a graphical representation of the verification results presented. The user also has the choice of having the verification results presented as they are i.e. model configuration and the computed result.

4 Algorithms

The phrase “needle in a haystack”, as illustrated in Figure 4.1, describes fully the problem that arises, when searching for a solution in an partially unknown space. A solution may or may not exist, and if it exists it may be the only one, or possible not even an optimum, but just a candidate solution. A range of algorithms that are suitable for conducting a non-exhaustive search exists, and some of those are studied in this chapter. These algorithms, known as meta-heuristic algorithms, allow the exploration of unknown space without having much, if any, knowledge at disposal [22].


Hunts Needle in a Haystack

HOW LONG does it take to find a needle in a haystack? Jim Moran, Washington, D. C., publicity man, recently dropped a needle into a convenient pile of hay, hopped in after it, and began an intensive search for (a) some publicity and (b) the needle. Having found the former, Moran abandoned the needle hunt.

Figure 4.1: Needle in a haystack [33].

When a problem needs to be solved, several solutions can exist, yet the optima are often preferred. All of these solutions exists in the same space, called the *solution space*, where each element in the space represent a possible solution. To find the best possible solution, one should be able to differentiate the solutions according to some value. This value is called the *quality* of the solution. Given this search space and a way of measuring the quality, a solution can be obtained that has either a minimum or maximum value i.e. the solution is the best according to some minimization or maximization criterion, called the *objective*. An essential problem with the search space is that by the time of solving a problem, all elements in the search space is often only partial known. That is, the locations are known but not the quality value of those locations. This is a problem because it limits the amount of algorithms that can be used for solving a problem. This problem is further extended when the decision of choosing a better solution, called the *candidate solution*, needs to be taken. At that time, the quality of a candidate solution may not be known and need to be calculated, thus adding extra time for iterating through the search space. This is also a reason for why an exhaustive search is often not feasible, given that the scale of the search space is often very large, as it would simply take too long. Thus, in order to find a candidate solution, one must apply a strategy that limits the number of interesting solutions in the search space, and, at the same time, contains a criterion for handling the decision of elevating a particular candidate solution to become the current optimum.

Along with the study, some assumptions and details necessary for a practical implementation are described, such that the algorithms can be implemented in UPPAAL PARMOS.

4.1 Notation and definitions

To create a uniform and legible representation of the content of this and the following chapters, some rules for syntax and styling of both mathematical and other abstract terms are introduced. Some definitions are provided to ease the descriptions following the pseudo-code of the algorithms.

Literals	Integers are written plainly, like 0, and a boolean value is in bold face, like **true** or **false** .
Variables	All variables are written with a mono-space font. Integer, boolean, and candidate solution variables are written as small letters, like `variable`, and variables containing sets start with a capital letter, like `Variable`.
Sets	Sets are denoted enclosed in curly brackets with each element delimited by a comma, like {`a`, `b`, `c`}.
Assignment	`a` ← `b` means that `a` is assigned the value of `b`.
Procedures	Procedures are written in small caps face with an initial capital letter, like PROCEDURE.

Table 4.1: List of notations.

A combinatorial optimization problem is defined as the search for an optimum, or optimum approximation, from a finite set of solutions [1]. This problem of either minimisation or maximisation requires a search algorithm that does not require a complete iteration over the search space and a way of identifying the quality of a solution. From this definition a formal abstraction of a combinatorial optimization problem is written in Definition 1.

Definition 1

A combinatorial optimization problem is called an *instance* and is represented as a tuple $(S,\ O,\ f)$ where S represent a finite *solution space*, O the set of finite *objectives* and f is a mapping $f : S_i \times O \mapsto \mathbb{R}$ that denotes a *cost function*, where $S_i \in S$. The cost function is in the Algorithms of called COST

The concept of an index-based *solution space* in Definition 1 allows for a single joined abstraction of a *parameter collection* and the computed *quality* of this collection. The solution space can therefore be viewed as the container of all *valid* parameter combinations to a UPPAAL model and their appertaining computed quality measure. This measure is obviously first available once a *cost function* has been applied. The set of objectives of an instance is the same for all solutions, in the solution space, in that instance. These objectives are necessary for the cost function to measure the quality of a solution. The relation between a cost function, objectives, a solution and its quality is depicted in Figure 4.2.

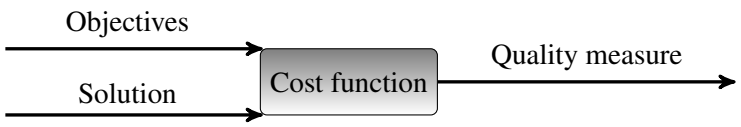


Figure 4.2: A illustration of the cost function.

By using index-based notation information can be extracted from a solution $a \in S$ of an instance. The available options are listed in Table 4.2.

$\mathtt{a}_{\mathtt{Parameters}}$	Is the set of all parameters in the solution `a`.
$\mathtt{a}_{\mathtt{Result}}$	Is the set of all computed results in the solution `a`.

Table 4.2: Notation for extracting informations of a solution $a \in S$ of an instance.

When iterating though a solution space, a procedure for finding the next candidate solution is needed. A profound idea in optimization theory is that the next good solution should be found around, or close, to a current good solution instead of being found nearby an arbitrary solution [18]. This idea is formalized in Definition 2.

Definition 2

Let $(S,\ O,\ f)$ represent an instance, where i and j are two solutions st. $i,\ j \in S$. Then a *neighbourhood*, denoted S_i, is the set of solutions that surrounds the solution i where $S_i \subset S$. Each solution $j \in S_i$ is called a *neighbouring solution* or simply *neighbour* of i.

The relation between the sets S and S_i of Definition 2 is depicted in Figure 4.3.

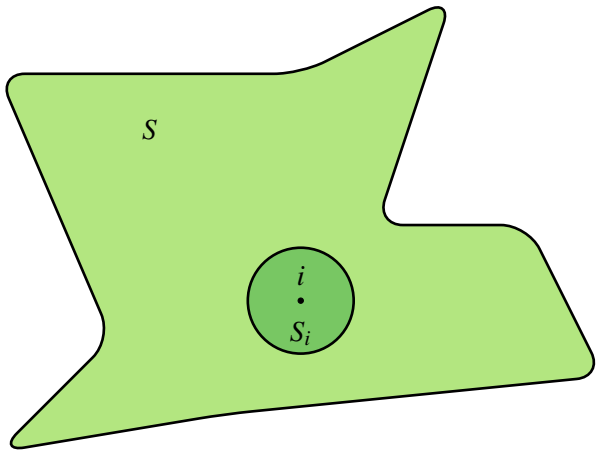


Figure 4.3: Illustration of the neighbourhood as defined in Definition 2.

In order to utilize the concepts of Definition 2 an algorithm, which provides predictability of the order of which the neighbouring solutions appear, is needed. Definition 3 provides such an algorithm.

Definition 3

An algorithm named NEIGHBOUR is defined as NEIGHBOUR: $S_i \mapsto S_j$ where $S_i,\ S_j \in S$ and S_j is the closest unvisited neighbour to S_i.

Figure 4.4 illustrates an example of the function NEIGHBOUR as defined in Definition 3. This example illustrates the path taken by the algorithm as it is applied to the same origin multiple times, each time finding the closest unvisited neighbour.

Some heuristic algorithms includes an iteration course that can continue for a long period

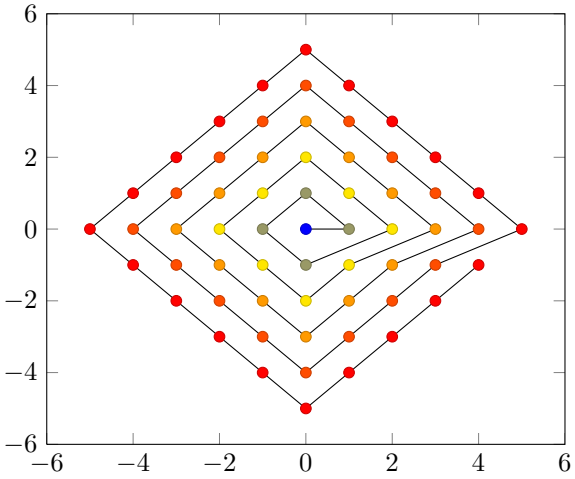


Figure 4.4: An example of the NEIGHBOUR function as defined in Definition 3.

of time. The decision to break the iteration in order to obtain the current best solution is defined in Definition 4.

Definition 4

The expression that decides when an continues algorithm in an iteration course is to stop is called the *stop-criterion*.

4.2 Algorithm classes

Algorithms that search for an objective can be divided into classes. This section provides descriptions of classes that support a single objective and multiple objectives. This section also contains a subsection that describes a special kind of algorithm that implement an evolutionary strategy to reach its objective.

4.2.1 Single objective algorithms

A single objective algorithm optimizes towards a single goal i.e. a value that should either be as small or as large as possible. The algorithm can be viewed as a maximization or minimization of a function $f(\vec{x})$ as in Equation 4.1 [18]. It should be noted that a maximization problem can be turned into minimization problem by inverting the sign of the quality measure hence the same algorithm can be used for either.

$$\begin{array}{l} \textit{maximize } f(\vec{x}) \\ \text{subject to } \vec{x} \in \vec{X} \end{array} \quad \vee \quad \begin{array}{l} \textit{minimize } f(\vec{x}) \\ \text{subject to } \vec{x} \in \vec{X} \end{array} \tag{4.1}$$

Where $\vec{x}$ in Equation 4.1 is a discrete solution vector, $\vec{X}$ is a finite set of feasible solutions and $f : \vec{x} \mapsto \mathbb{R}$.

If the function $f(\vec{x})$ is known and differentiable at each $\vec{x} \in \vec{X}$ then a gradient descent method can be applied as an optimization solver. However, when $f(\vec{x})$ is unknown it becomes a *black box* and other types of algorithms that do not require a derivative for optimization is needed. Meta-heuristic algorithms are such a type. They allow the exploration of unknown space and iterates through it using only available knowledge.

In Section 4.3 and Section 4.4 two single objective meta-heuristic algorithms, *Hill-Climbing* and *Simulated Annealing*, are studied for later implementation.

4.2.2 Multi objective algorithms

When the objective for an algorithm need to be expressed using multiple objectives, the algorithms of 4.2.1 are no longer directly sufficient. However, those algorithms could be utilized if an aggregate objective value, i.e. a combination of the objective values into one objective value, is used. This can be done as in Equation 4.2, where the objectives, denoted a_i, are scaled in according to some weights, denoted w_i.

$$\sum_{i=1}^{|a|} a_i \cdot w_i \tag{4.2}$$

This will result in the objectives being weighted evenly and a single value can be maximized or minimized. It is of course possible to use different weights, or use a non-linear formula, but it may not always be easy to figure out the most optimal formula, thus resulting in less than optimal solutions, in which case another strategy is needed.

Pareto dominance is a strategy to determine which one, if any, of two solutions are the best, using multiple objectives. In order for a solution A to dominate another solution B, it must be at least as good as B in all objectives, and better in at least one objective.

In other words: For two solutions Z and X, X dominates Z if both Equation 4.3 and 4.4 holds, denoted $X \succ Z$.

$$\forall i \in \{1,...,m\} : Z_i \leq X_i \tag{4.3}$$

$$\exists i \in \{1,...,m\} : Z_i < X_i \tag{4.4}$$

When having multiple objectives, a number of things can happen when comparing two solutions:

1. A is better than B in all objectives
2. A is as good as B in all objectives, and better in at least one
3. A is as good as B in all objectives
4. A is better than B in at least one objective, but B is better in at least one objective.
5. A is as good as B in almost all objectives, but worse in at least one
6. A is worse than B in all objectives.

In the cases 1 and 2, A dominates B, and in cases 5 and 6, A is dominated by b. In the remaining cases, neither clearly dominates the other, they are thus nondominated. This

may produce multiple 'best' results, as can be seen in Figure 4.5, which represents two objectives, each having to be maximized, with the line of blue dots representing what is known as the pareto front, where each point is nondominated.

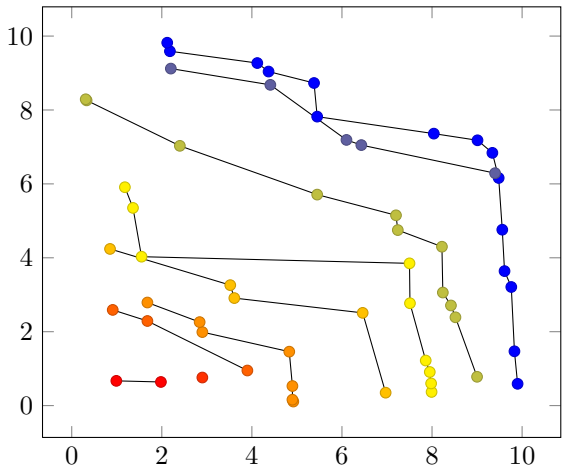


Figure 4.5: The Pareto front.

The Pareto Front may comprise excessively many solutions, with no clear way to distinguish which one may be best. Many algorithms therefore do not attempt to find the single best solution, but instead provide a sampling of the Pareto Front, from which a human user or perhaps another algorithm is able to choose the solution best fitting the situation, perhaps in the case of a human user, using some not easily formalizable criterias.

In Section 4.5 on page 52 a multi objective meta-heuristic algorithm, *Pareto Archived Evolution Strategy (PAES)*, is studied for later implementation.

4.2.3 Genetic and evolutionary algorithms

The class of genetic and evolutionary algorithms are inspired by the theory of evolution as established by Charles Darwin. He stated that all species of life have descended over time from a common ancestry and that a process of natural selection would lead to a more homogeneous population due to the survival of the strongest genetic variation within a population [8]. This principle of evolution was introduced in computing by [34], which initiated the idea of natural evolution as a method to solve parameter optimization problems. This idea was further researched and developed by [14] where he introduced the concept of Genetic Algorithms (GAs).

The class of genetic and evolutionary algorithms differs from the algorithms described in 4.2.1 and 4.2.2. They do so by their structure, which holds a population of candidate solutions rather than a single candidate solution. Based on the implementation of these algorithms, each of the candidate solutions can affect the population mechanism and quality measure, which thereby affect how new candidate solutions evolve [22]. As a result of this, good candidate solutions could influence poor candidate solutions to become extinct, or, by affecting them to change behaviour, to become better.

Two common versions of the GA exists, one called a *generational* and once called *steady-*

state. The generational version of a GA updates the entire population once per iteration, while the steady-state version only updates the population with a few new candidates at a time. The outline of a generational GA is as depicted in Figure 4.6.

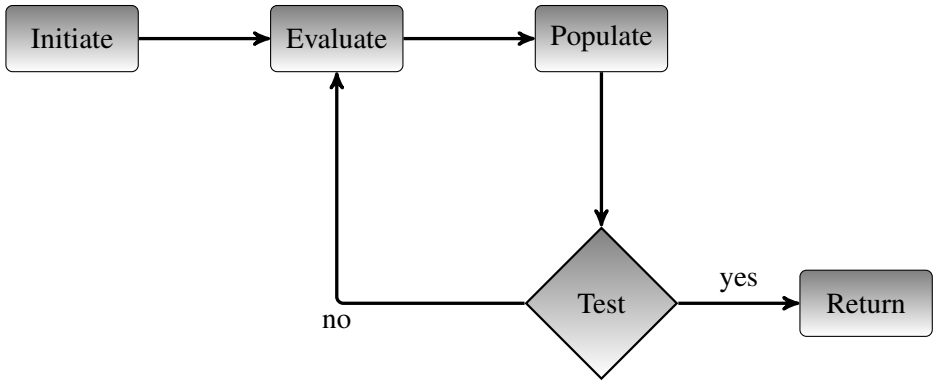


Figure 4.6: The flow of a Genetic Algorithm.

The first process step in Figure 4.6 contains the initiating phase where a population, i.e. a set of candidate solutions, of chromosomes is randomly generated. The term chromosome is drawn from the natural science of biology in which evolution and genetics are a parts of. It is used as an abstraction of a solution that has a data type that can be represented in form of a fixed-length vector, which is necessary for the Populate process [22], depicted in Figure 4.7.

Then follows a process of evaluation where each chromosome in the population is measured according to some function that returns its quality. In the third process is the Populate process where a new population is generated, using the current population as origin. This process contains internal processes which are depicted in Figure 4.7 and elaborated on later in this section. The fourth process, called Test, contains an end condition for when no more populations should be generated. The fifth and final process returns the best solution in the current population.

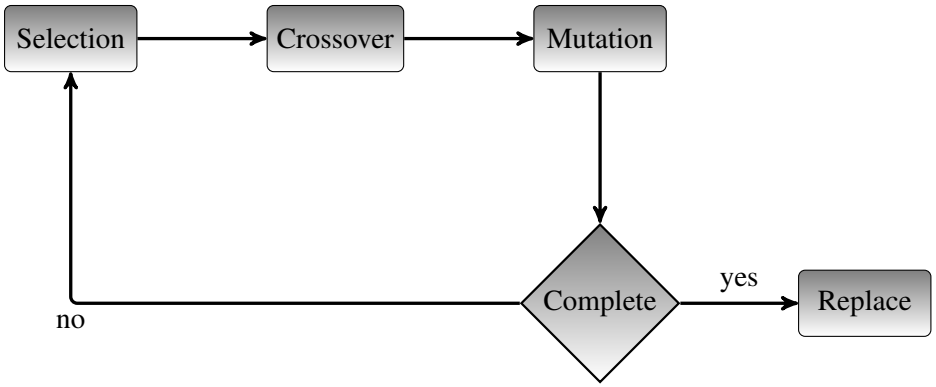


Figure 4.7: The internal flow of the populate process in a Genetic Algorithm.

The internal processes of the Populate process in Figure 4.6 is depicted in Figure 4.7. It consists of three process steps surrounded by an iteration, that break on a specified end-condition, and a final Replace process. The first *Selection* process selects two parent chromosomes from the current population. This is done in according with the quality values

of the chromosomes i.e. the higher quality value, the higher chance to be selected. The second process is the *Crossover* process where a new offspring is conceived. A probability condition decides whether a crossover is performed using the parent as origin otherwise the offspring is just an exact copy of the parents. Then follows a *Mutation* process where a new probability condition decides whether the new offspring is mutated. If the the condition is met, the chromosome is mutated at each locus i.e. each position in the chromosome.

It is now clear that, in order to map the GA on to any specific problem, some prerequisites exists. The GA need to be able to measure the quality of each solution, in order to distinguish them, and the algorithm should know a termination criterion, for when a solution is regarded as optimum. It is also important that the data is contained in a way that allows for the processes depicted in Figure 4.7 to operate.

4.3 Hill-Climbing

Hill-Climbing is a simple iterative algorithm, suitable for finding optima in a *solution space*. It assumes that as long as an optimum has not been reached, better solutions exist in the vicinity of the current solution. The neighbouring solutions can thus be explored, and the current solution in the direction where the better nearby solutions are found.

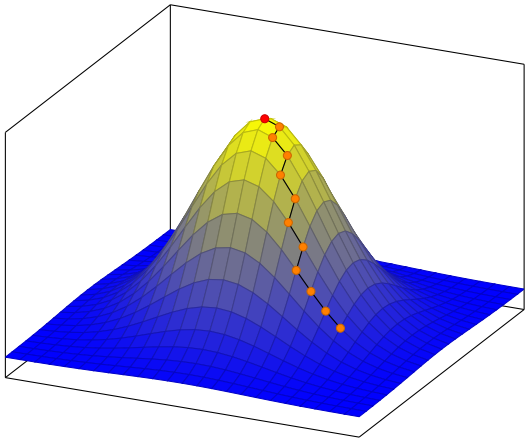


Figure 4.8: Visualization of the Hill-Climbing algorithm.

Hill-Climbing works, in its simplest form, by starting at some solution in a solution space, tests a neighbouring solution, and continue to this solution, if it is better than the current best solution. This continues until an optimum has been reached.

Figure 4.8 shows an illustration of the path taken by a Hill-Climbing algorithm in a two-dimensional parameter space (The vertical axis denotes how good the solutions are). The red dot marks the point where the algorithm has reached a maximum and ended the search.

4.3.1 Formalisation

The Hill-Climbing algorithm generally requires that there is a strong relationship between change in parameters and the quality of a solution. Neighbouring solutions must show some degree of similarity, in order for Hill-Climbing to be useful.

The algorithm is good for finding maxima or minima, but because the algorithm moves locally in every iteration, it does not guarantee to find the global optimum. If the initial solution is in an bad area, the search will end in a local optimum rather than the global one. This is illustrated in Figure 4.9, where a Hill-Climbing path ends up in an optimum which is not the global optimum.

A number of variations of the Hill-Climbing algorithm exists. Most notably is Steepest Ascent Hill-Climbing and Random Restart Hill-Climbing.

Steepest Ascent Hill-Climbing is a variation of simple Hill-Climbing, where the immediate neighbours of the current best solution is tested in each iteration. More than one of these solutions may be better than the current best solution, but the best of all these solutions is likely to indicate the fastest direction to an optimum.

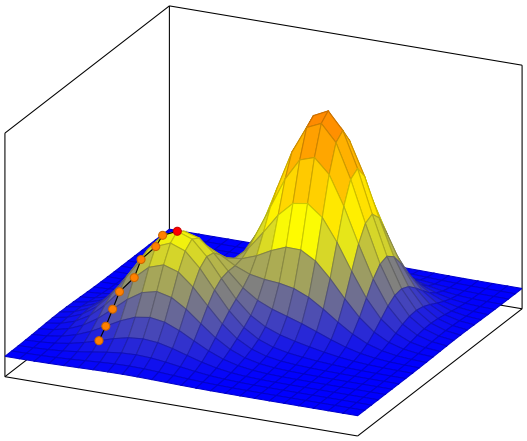


Figure 4.9: Visualization of the Hill-Climbing path .

Random Restart Hill-Climbing (also known as Shotgun Hill-Climbing) is an approach where Hill-Climbing is run multiple times using different starting solutions in the same parameter space, and the best solutions is chosen among the solutions returned by each agents. Since this approach explores broader in the parameter space, is increases the likelihood for finding a global optimum significantly.

4.3.2 Pseudo-code

A simple Hill-Climbing algorithm is shown in Algorithm 4.1.

Algorithm 4.1 Simple Hill-Climbing Algorithm

```
    input:
    output: solution
 1: procedure HILLCLIMBING
 2:     solution ← Initial random candidate solution    ▷ Holds the current best solution.
 3:     repeat
 4:         candidate ← NEIGHBOUR(solution)
 5:         if COST(candidate) < COST(solution) then
 6:             solution ← candidate
 7:         end if
 8:     until an optimum has been reached, or other stop criterion has been met
 9:     return Solution
10: end procedure
```

The Steepest Ascent Hill-Climbing algorithm shown in Algorithm 4.2, is a slight modification of the simple Hill-Climbing algorithm, where more than one neighbour is tested in each iteration.

Algorithm 4.2 Steepest Ascent Hill-Climbing Algorithm

```
    input:
    output: solution
 1: procedure SteepestAscentHillClimbing
 2:     solution ← Initial random candidate solution    ▷ Holds the current best solution.
 3:     repeat
 4:         improved ← false
 5:         for each neighbour ∈ S_solution do
 6:             if Cost(neighbour) < Cost(solution) then
 7:                 solution ← neighbour
 8:                 improved ← true
 9:             end if
10:         end for
11:     until not improved, or other stop criterion has been met
12:     return Solution
13: end procedure
```

4.3.3 Implementation details

Because our framework requires that an algorithm is able to deliver a new candidate solution whenever asked, "an iteration" becomes a very loosely defined concept.

Having a great degree of parallelism to process results, it makes sense to explore not only the neighbouring solutions, but also other nearby solutions - like neighbours neighbours, and maybe further away, depending on how many computing resources are available, and thus, every time the algorithm is asked for a new solution it just iterates a little further away from its current best solution. Since the current best solution cannot be changed until better results have been returned from solutions being computed, the algorithm must find new nearby solutions to be computed. The number of solutions being computed in each iteration will therefore increase with the number of available resources. Because of this, the algorithm for finding neighbours must be an incremental one, always able to return the n^{th} neighbour from a given current solution.

The current best solution can then be changed when new better results are returned. The results may also return in any given order, and since new solutions are queried for computation when others return results, the current best solution is therefore bound to change before all solutions return results.

4.4 Simulated Annealing

In the Simulated Annealing (SA) algorithm the criterion of elevating a particular candidate solution to be the current optimum, is inspired by the thermal process of condensed matter physics. This process includes two steps;

- an increase in temperature to a value where some solids melts and
- a carefull decrease in temperature until the particles in the solids arrange themselves in a ground state.

Once the solids melt, and are in a liquid phase, all their particles arrange themselves according to some stochastic process. This process has successfully been modeled by [16]. However, [24] developed in 1953 a small algorithm for simulating the evolution of a solid in a heat bath. This algorithm was based Monte Carlo techniques and generates a sequence of states of the solid. The algorithm consists of a criterion that compares the energy of a current solid state with the energy of a new solid state, derived from a perturbation mechanism [9]. This energy difference decides whether a new candidate solution state is accepted as the new current solution. This criterion, see Equation 4.5, is known as the Metropolis criterion, which is at the heart of the SA algorithm.

$$\exp\left(\frac{E_i - E_j}{K_B T}\right) \tag{4.5}$$

Where T denotes the temperature of the heat bath, K_B is the *Boltzmann constant* and $E_i - E_j$ is the energy difference of two states. In order to utilize this physical concept in a combinatorial optimization problem some analogies are stated.

- The states in the physical process are equivalent to solutions in the combinatorial optimization problem.
- The energy of a state in the physical process is equivalent to the cost of a solution in the combinatorial optimization problem.
- The Boltzmann constant and chosen temperature utilized in the Metropolis criterion are equivalent to a control parameter in the SA algorithm.

These definitions allow the SA algorithm to be viewed as iteration of Metropolis criterions, with the control parameter as a candidate solution acceptance criterion controller, which provides the ability to choose whether a lesser qualified candidate solution should be elevated to optimum. This choice of a lesser qualified candidate solution is a strength of the SA algorithm because it allows for a wider exploration of a search space where other local search algorithms have problems escaping a local minimum.

This principle of elevating a candidate solution, of lesser quality, to be the optimum is depicted on Figure 4.10. The function ζ is given, and the objective is to locate the global minimum. The point A on the figure indicates a local minimum of the function ζ where a local search algorithm would be trapped. This entrapment is avoided because the SA algorithm is able to move to another point such as B, and although this point seems worse than A. This allows the algorithm to reach the point C.

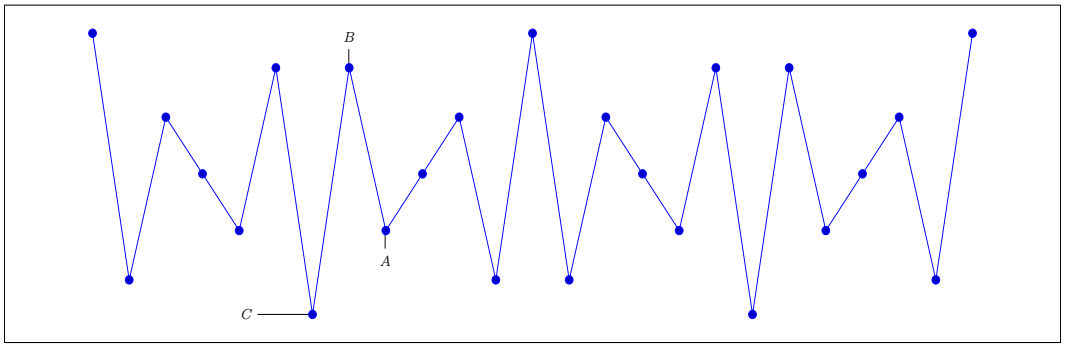


Figure 4.10: A segment of a periodic function denoted ζ.

4.4.1 Formalisation

In order to present an essential description of the algorithm some necessary definitions are provided. These definitions are in accordance to those provided by [1].

Definition 5

If $(S,\ O,\ f)$ represent an instance and i and j two solutions such that $i, j \in S$ with cost function f. Then an *acceptance criterion* determines whether j is accepted instead of i by applying the probability $\mathbb{P}_{\mathbb{C}}$:

$$\mathbb{P}_{\mathbb{C}} = \begin{cases} 1 & \text{if } f(j) \leq f(i) \\ \exp\left(\frac{f(i)-f(j)}{\mathbb{C}}\right) & \text{otherwise} \end{cases}$$

$\mathbb{P}_{\mathbb{C}}$ is derived from the Metropolis criterion as previously described, where $\mathbb{C}$ is the configuration i.e. the value of the control parameter.

Definition 6

The transformation process of a current solution to a new candidate solution is called a *transition*. This process contains two steps;

1. Run the perturbation mechanism.
2. Validate the new candidate solution against the acceptance criterion.

4.4.2 Pseudo-code

Algorithm 4.3 contains the main procedure for the SA algorithm. The algorithm uses some auxiliary procedures that are found in Algorithm 4.4. Algorithm 4.4 is special, in the sense that its procedures control the number of transitions and acceptance ratio in the SA algorithm. These procedures are elaborated on in Section 4.4.3 given that, in order to utilize the SA algorithm practically, they must be computeable in finite-time, which they by definition are not.

Algorithm 4.3 executes on an instance and contains six variables that are listed in Table 4.3.

Algorithm 4.3 Simulated Annealing Algorithm

```
1:  procedure SIMULATEDANNEALING
    input: An initial solution i.
    output: The new solution.
2:      k ← 0
3:      Seed_init ← initial state
4:      L_0 ← initial value
5:      C_0 ← initial value
6:      repeat
7:          for L_k times do
8:              j ← NEIGHBOUR(i)
9:              if COST(j) ≤ COST(i) then
10:                 i ← j
11:             else
12:                 if exp((f(i)−f(j))/C_k) > RANDOM(Seed_init) then
13:                     i ← j
14:                 end if
15:             end if
16:         end for
17:         k ←k + 1
18:         L_k ← CALCULATELENGTH(L_k)
19:         C_k ← CALCULATECONTROL(C_k)
20:     until stop criterion
21:     return i
22: end procedure
```

k	denotes the iteration count value.
$Seed_{\text{init}}$	denotes the state of the seed.
C_k	denotes the value of the control parameter for the k$^{\text{th}}$ iteration.
L_k	denotes the number of transitions generated for the k$^{\text{th}}$ iteration.
i	denotes the current solution.
j	denotes the candidate solution.

Table 4.3: Variables of Algorithm 4.3.

The first part of Algorithm 4.3 sets up the initial values of its variables. It moves on into an iteration, which continues as long as the search for a new solution of higher quality is possible, within the boundary of the stop criterion. In each iteration multiple transitions are taken in order to mutate the current solution into a new solution. The number of transitions taken is possibly diverse in each iteration, yet this is decided by the implementation of the *CalculateLength* procedure in Algorithm 4.4. The decision of elevating a candidate solution to be an optimum, is controlled by the acceptance criterion. The acceptance criterion, depends on the *CalculateControl* procedure in Algorithm 4.4, and states that if the current solution is better than a candidate solution, then a stochastic method shall decide whether the candidate solution should be elevated although it is of a lesser quality.

Algorithm 4.4 Simulated Annealing Algorithm Auxiliary Procedures

1: **procedure** CALCULATELENGTH
input: The length variable $\texttt{L}_\texttt{k}$
output: The number of transitions to be taken at the k$^{\text{th}}$ iteration.
2: **end procedure**

3: **procedure** CALCULATECONTROL
input: The control parameter variable $\texttt{C}_\texttt{k}$
output: The control parameter to be used at the k$^{\text{th}}$ iteration.
4: **end procedure**

Algorithm 4.4 contains two important procedures for controlling both the run-time and transitions of the SA algorithm. The *CalculateLength* procedure computes the number of transitions needed for the next iteration. The *CalculateControl* procedure computes the decreasing value of the control parameter.

4.4.3 Implementation details

Multiple factors must be taken into account when the SA algorithm is to be implemented. This implementation relies on literature provided by [1] where that of importance is derived and presented in this section.

The most important factor is that the algorithm is designed in such a way that, to serve as an optimization algorithm guaranteeing a globally optimum solution, it requires an infinite number of transitions [1]. However, an implementation that allows for finite-time execution of the algorithm is considered instead. This implementation comes with the cost of yielding the guarantee of finding the optimum solution, to allow a optimum approximation to be satisfying. This is handled by implementing a cooling schedule, see Definition 7, that allows for polynomial-time execution Algorithm 4.3.

Definition 7

A *cooling schedule* includes the specification of finite values for;

- An *initial value* of the control parameter.
- A *decrement function* to decrease the control parameter.
- A *final value* of the control parameter to specify the *stop criterion* of the algorithm.
- A *length* to specify the number of transitions at each value of the control parameter.

Definition 7 contains four parameters, which needs to be found. This requires further algorithms and these are presented as derived from the abstract guidelines of [1].

Initial value

At initialization, the value of the control parameter C_0 should be sufficient large to allow almost all transitions to be accepted. This requirement is useful when an adequate value of C_O is to be found. Some definitions, listed in Table 4.4, are made before an adequate approximated value can be calculated.

Let I be an instance $(S,\ O,\ f)$.

Let α be a chosen integer st. $1 \leq \alpha \leq |S|$.

Let β be a appropriate control parameter st. $\beta \in \mathbb{R}$.

Let a_{Random} be a randomly chosen solution st. $a_{Random} \in S$.

Let m_1 be the number of transitions from i to j where $f(j) \leq f(i)$.

Let m_2 be the number of transitions from i to j where $f(j) > f(i)$.

Let $\overline{\Delta f^+}$ be the average difference in cost over the m_2 cost-increasing transitions.

Table 4.4: List of notations.

Now the acceptance ratio ϕ for the acceptance criterion, see Definition 5, can be approximated by Equation 4.6.

$$\phi \approx \frac{m_1 + m_2 \cdot \exp\left(\frac{-\overline{\Delta f^+}}{\beta}\right)}{m_1 + m_2} \tag{4.6}$$

From Equation 4.6 a new Equation 4.7 can be obtained by isolating β in Equation 4.6. To emphasise the use of an approximated value ϕ in Equation 4.7, which is found using a pre-selected β value in Equation 4.6, β is renamed to C_0.

$$C_0 = \frac{\overline{\Delta f^+}}{\ln\left(\frac{m_2}{m_2 \cdot \phi - m_1(1-\phi)}\right)} \tag{4.7}$$

The value of C_0 can now be calculated using Equation 4.7 and Algorithm 4.5.

Decrementing the control parameter

The choice between a large number of transitions or small changes in the value of the control parameter is problem specific. A way of decrementing the control parameter is to have a uniform decrementataion constant e.g. τ, which is multiplied with the control parameter in each iteration.

$$C_{k+1} = \tau \cdot C_k,\ k = 1, 2, \ldots \tag{4.8}$$

[1] argue for typical values of τ to lie between $0.8 \leq \tau \leq 0.99$.

Final value of the control parameter

The cost function is suggested to be the key criteria in the stop criterion of the algorithm. Once the quality of the last solution in a sequence of transitions remain unchanged for a number of consecutive iterations the algorithm should stop.

Number of transitions needed

The number of transitions needed should intuitively be as “small” as possible. Here “small” should be read as, the number of transitions where the algorithm has a sufficiently large probability of discovering a large part of the neighbourhood of a given solution.

Algorithm 4.5 Simulated Annealing Algorithm Initial Control Parameter Calculator

```
1:  procedure SAAICPC
2:      m1, m2, w ← 0
3:      i ← random solution
4:      for α times do
5:          j ← NEIGHBOUR(i)
6:          if COST(j) ≤ COST(i) then
7:              m1 ← m1 + 1
8:          else
9:              m2 ← m2 + 1
10:             w ← w + COST(j) - COST(i)
11:         end if
12:         i ← j
13:     end for
14:
15:     Δf+ ← w / m2
16:
17:     ϕ ← (m1 + m2 · exp(−Δf+ / β)) / (m1 + m2)
18:
19:     m1, m2 ← 0
20:     i ← random solution
21:     for α times do
22:         j ← NEIGHBOUR(i)
23:         if COST(j) ≤ COST(i) then
24:             m1 ← m1 + 1
25:         else
26:             m2 ← m2 + 1
27:         end if
28:         i ← j
29:     end for
30:
31:     C0 ← Δf+ / ln(m2 / (m2 · ϕ − m1 · (1 − ϕ)))
32:
33:     return C0
34: end procedure
```

4.5 Pareto Archived Evolution Strategy

PAES is an evolutionary algorithm, using the currently best solution, known as the *current* solution, to generate a new *candidate* solution, by performing a small mutation on the *current* solution, thus making it a local search algorithm similar to the Hillclimbing algorithm described in Section 4.3.

It evaluates the *candidate* solution against the *current* solution, using Pareto Dominance in order to be able to optimize towards multiple objectives, to determine whether the *candidate* should be promoted to the new *current*.

However, as described in section 4.2.2, it is not always possible to establist clear dominance between two solutions. Therefore PAES holds in an archive a number of the nondominated solutions found so far. These are used to establish dominance when neither the *current* nor the *candidate* solution dominates the other, by comparing the *candidate* solution to each solution in the archive.

If the *candidate* dominates any solution in the archive, it gets replaced by the *candidate*, and the *candidate* is considered to dominate the *current*, which then gets replaced.

In order to keep the size of the archive managable, only a limited number of solutions are stored. Once the archive is full, selection of which solutions are stored, is based on a diversity criterion, if clear dominance cannot be established. If a new solution introduces more diversity to the archive, than an existing solution, it is added, otherwise it is ignored, thus providing the most diverse set of optimal solutions [17].

4.5.1 Pseudo-code

Algorithm 4.6 PAES Algorithm

```
1:  procedure PAES
2:      Archive ← ∅                                  ▷ Contains the best solutions found so far
3:      current ← random solution                    ▷ Holds the most recent best solution.
4:      Archive ← Archive ∪ {current}
5:      while stopping criterion is not met do
6:          candidate ← a, a ∈ S_current             ▷ S_current is the neighbourhood of current.
7:          if COST(current) ⊁ COST(candidate) then
8:              if ARCHIVECHECK(candidate) then
9:                  current ← candidate
10:                 Archive ← Archive ∪ current
11:             end if
12:         end if
13:     end while
14: end procedure
```

The PAES algorithm described in Algorithm 4.6 consists of two main parts

- the initialisation from line 2 to 4 and
- the optimisation loop from line 5 to line 13.

The initialization initializes the `Archive` and the `Current` solution, and, since there are no other solutions that dominates `current`, it is automatically added to `Archive`, which is necessary for the optimization loop.

Algorithm 4.7 ArchiveCheck

```
    input: solution a
    output: boolean                          ▷ does a dominate any solution in the archive
 1: procedure ARCHIVECHECK
 2:     result ← false
 3:     tied ← true
 4:     for each Solution_i ∈ Archive do
 5:         if COST(Solution_i) ≺ COST(a) then
 6:             Archive ← Archive \ {Solution_i}
 7:             result ← true
 8:             tied ← false
 9:         else
10:             if COST(Solution_i) ≻ COST(a) then
11:                 result ← false
12:                 tied ← false
13:             end if
14:         end if
15:     end for
16:     if tied then
17:         result← DIVERSITYCHECK(a)        ▷ inserts a into Archive if it introduces
                                                greater diversity.
18:     end if
19:     return result
20: end procedure
```

4.5.2 Implementation details

As our framework requires that an algorithm is able to deliver a new solution whenever asked, it is modified to have one *current* solution and multiple *candidate* solutions. These candidates are evaluated as results come in, and the *current* replaced when the domination check dictates. This implies that a *candidate* can be checked against a different *current* than the one that it was created from.

However, the only difference is that the *current* solution may be different for a number of iterations, before reverting to the 'correct' solution, thus providing a bit more exploration, for any *candidate* solutions generated in that period. This may even possibly add to the diversity of the archive, thus perhaps even finding otherwise undiscovered optimal solutions.

Another detail of the implementation is the diversity check. The diversity check compares the distance between solutions in the archive, and the *candidate* solution, using the manhattan distance of the results.

In order to ummarize the results of different solutions, which may have different ranges, they need to be normalized to a common range, to avoid having one result weigh more than another. This is done using Equation 4.9, which normalizes it to a range of $]0,1[$. In order to do this, a minimum and maximum value for each result must be known, however a global minimum or maximum for each result may not be known, when needed, in which case the maximum and minimum value currently residing in the archive as well as the *candidate* solution can be used, but in this case it must be recalculated each time a new solution is tested.

Once the normalized distance is calculated for each solution in the archive, as well as the *candidate* solution, it finds the solution in the archive closest to the *candidate*, known as the *current*, and the two closest to it, known as the *predecessor* and *successor*, as shown in Figure 4.11. Using these two points as reference, it calculates which one of *candidate* and *current* is closest to the middle of *predecessor* and *successor*.

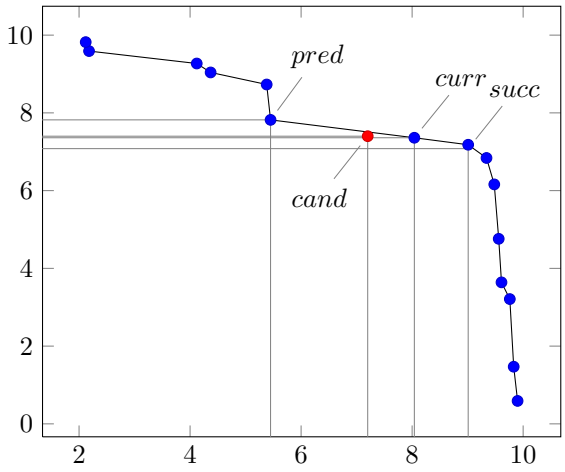


Figure 4.11: Diversity calculation

$$
\begin{aligned}
dist &= \sum_{i=0}^{|S_{result}|} \frac{\mathtt{S}_i - \mathtt{Min}_i}{\mathtt{Max}_i} \\
Min_i &= min\{\forall j \in Archive : j_{Result_i}\} \\
Max_i &= max\{\forall j \in Archive : j_{Result_i}\}
\end{aligned}
\tag{4.9}
$$

4.6 Parallelization

A way of parallelizing existing algorithms, is to, whenever multiple solutions needs to be tested, before a decision can be made, let them test all at once. This is usable e.g. in Steepest Ascent Hill-Climbing, where all solutions next to the current needs to be evaluated. However, this provides only a limited parallelization, as only the solutions immediately next to the current solution can be tested simultaneously, and is not applicable to all algorithms, as some may require a result immediately, before the next solution can be calculated.

As the algorithms this chapter are all local search algorithms, some parallelization can be done in a similar way for all. Two methods for this, speculative scheduling and multiple agents, are described in the following subsections.

4.6.1 Multiple instances

In order to parallelize a local search algorithm, it is possible to run multiple instances of the algorithm, either each within its own subset of the parameter space, or all in the entire parameter space. In both cases, a monitor is needed, in order to keep track of the global best solution. If all instances are searching in the entire parameter space, it must also monitor for convergence, in which case one instance must be stopped, or possibly restarted from a new origin point. The concept of convergence of multiple instances is illustrated in Figure 4.12, where two instances are running simultaneously. Once the red instance reaches the blue, the red one must be terminated, as it will simply follow the same path as the blue already has taken.

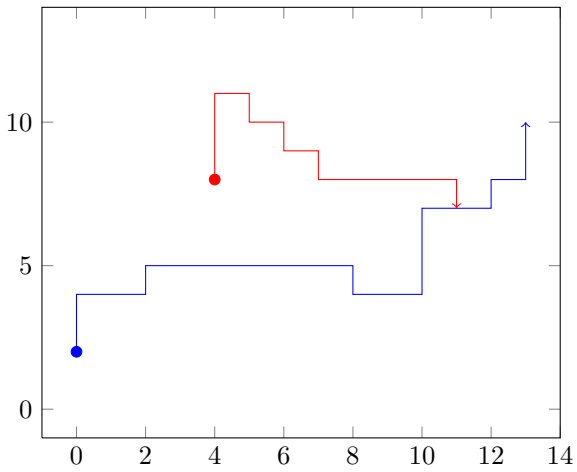


Figure 4.12: Convergence of multiple local search algorithm instances

This method is similar in nature to the *random restart* described in Section 4.3.1, except that each run is executed in parallel.

4.6.2 Speculative testing

When doing speculative testing, solutions that are not necessarily needed by the algorithm right now but might be necessary in the near future, are tested anyway, just in case that they are needed later. This allows the results to be available to the algorithm as soon as it needs them, as they have already been calculated beforehand.

In a local search algorithm, this is fairly straightforward to implement, as the solutions possibly needed later on, are the ones nearby the current solution. Thus, using a pattern such as the one generated by the *Neighbour* algorithm, shown in Figure 4.4, can be used to predict which solutions may be neccesary in the near future.

5 Implementation of UPPAAL PARMOS

The purpose of this chapter is to present our implementation of UPPAAL PARMOS, which is based on the design specified in Chapter 3. Furthermore, this chapter contains not only the implementation of UPPAAL PARMOS, but also the implementation of some components to be utilized for testing of the system. These components are the three algorithms described in Chapter 4, i.e. Hill-Climbing, SA and PAES, as well as a Resource proxy named *upslurm*, in order to utilize the cluster placed at our disposal for testing purposes.

Based on the design, we have chosen to utilize a range of existing software components, This is decided in order to faster create a working prototype, but also in recognition with the fact that, it would be impossible within the time frame of this thesis, to develop these components to a level where they are available on the marked today.

One of the most essential software components that we have chosen to utilize, was the programming framework called .NET Framework. It is build according to the Common Language Infrastructure (CLI) specification [15], which provides basis for executing UPPAAL PARMOS on other platforms e.g. the open source implementation of CLI called Mono. Currently, only the .NET Framework supports the full implementation of CLI, yet the Mono Project is continuously developing, in order to support the full implementation of the CLI. Utilizing this programming framework has minimized the amount of new code that was needed to be written by providing already written and exhaustive tested libraries.

There exists compilers for a range of programming languages that compiles to the intermediate language which the CLI supports. We have chosen to utilize one of these languages called the C# programming language, which supports an object-oriented programming style. This matches with the design decision of developing a system using a component based approach, where components can be implemented using object-oriented techniques e.g. data abstraction, encapsulation, polymorphism etc. [23].

Throughout of this chapter we present implementation of the design using class diagrams. Figure 5.1 gives an overview of the different types of objects and the relation between them. There are to kind of objects on Figure 5.1, a class and an interface. The class is illustrated as a rectangle, where an interface is illustrated using rounded corners. Figure 5.1 illustrates also the four different types of relations there can exist between classes.

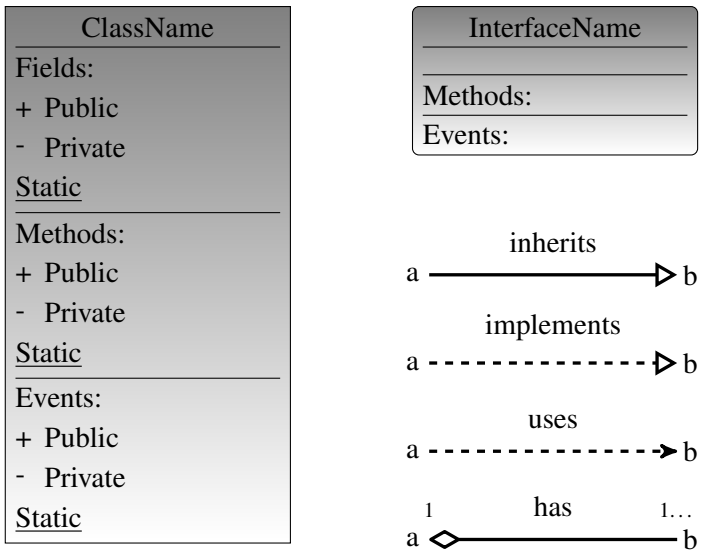


Figure 5.1: Illustrates the different class types used throughout this section.

5.1 Parameter Sweep Application

The design of the PSA has been implemented as a background running service. The PSA receives all its instructions from the database and are not directly in contact with the user accessing UPPAAL PARMOS through the GUI.

As illustrated in Figure 5.2 the PSA is constructed around two class instances, a *Service* and a *Resource Handler*.

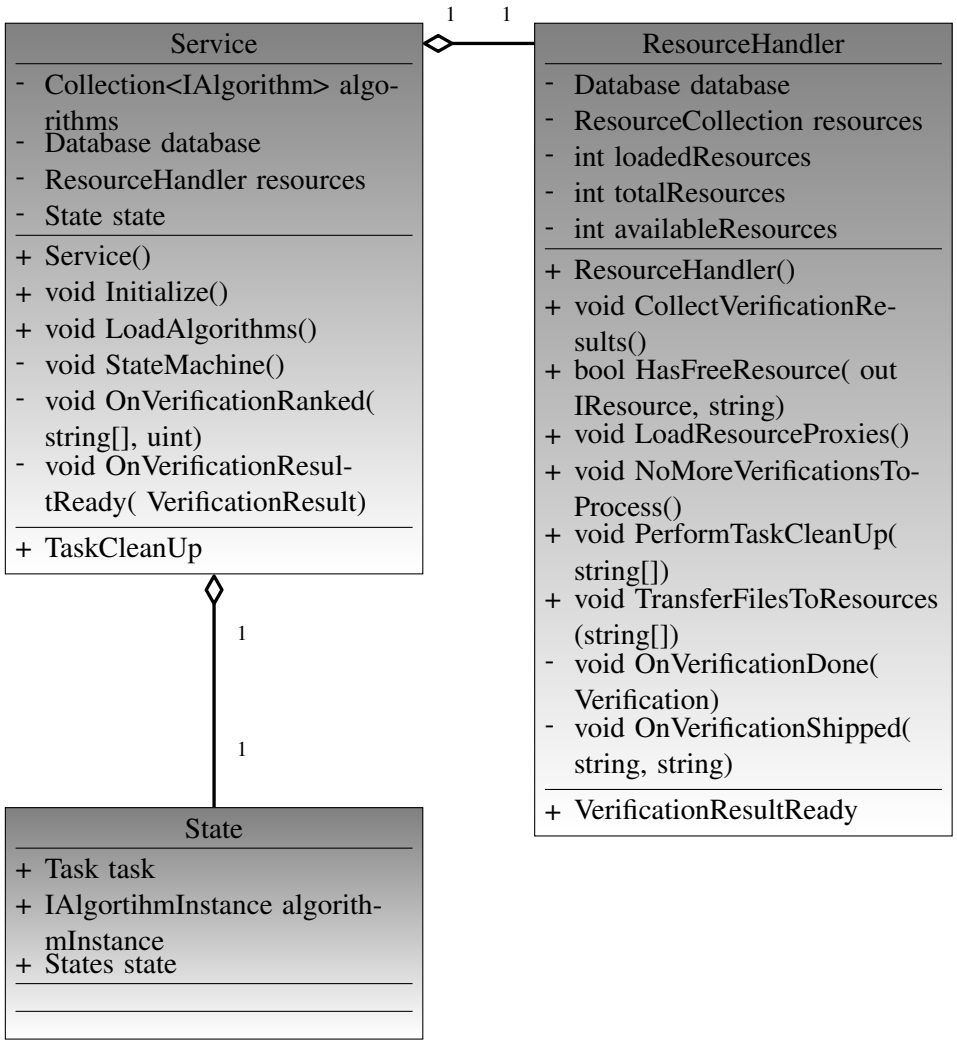


Figure 5.2: Class diagram of the main PSA component: Service, and its appertaining classes

The Resource Handler instance is responsible for controlling the attached resource proxies and provides an interface to the Service, where the Service can lock and ship verifications to a resource proxy. The Service instance is responsible for requesting model configurations from the conduction Algorithm and shipping these using the Resource Handler. It is also the Service that tells the Resource Handler when to check for results, what to transfer of auxiliary files and when to clean-up after a sweep.

The Service is implemented as a state machine an utilized a class named *State* for storing

information related to the current state, which the state machine is in. The Service class contains the following methods.

Initialize is a method called once by the main program to initialize an instance of the Service.

LoadAlgorithms is called both at unitialization both also when the user supplies a new algorithm.

StateMachine is called once to initiate the state machine.

OnVerificationRanked is called as a callback when the rank of a solution is found, and need to be stored in the database.

OnVerificationResultReady is called as a callback when a resource proxy has a verification results ready for the Service.

TaskCleanUp is an event that is fired by the Service to notify the Resource Handler to start clean-up of the current task at the attached resources.

The Resource Handler contains the following methods:

CollectVerificationResults is called by the Service to notify the resource proxies to check their resource for finished verifications.

HasFreeResource is called by the Service to check whether a resource is available.

LoadResourceProxies is called both in the constructor and once the user adds new resources to the system.

MoMoreVerificationsToProcess is called by the Service to notify the resource proxies that there are no more verifications to run in the current task.

PerformTaskCleanUp is called by the Service to notify the resource proxies that a clean-up of the current task should be performed.

TransferFilesToResources is called by the Service to notify the resource proxies that a list of files need to be transfered to its resource.

OnVerificationDone is a callback called by the resource proxy when it has a verification result ready for the Service.

OnVerificationShipped is a callback called by the resource proxy when it has shipped a verification to its resource.

5.1.1 Components

In order to implement a pluggable component-based design of UPPAAL PARMOS, we have chosen to utilize *assemblies*, which are libraries that contains compiled CLI supported code. Assemblies can, using the *System.Reflection* library of the .NET Framework, dynamically at run-time be loaded and instantiated into an already running process [25–27].

The choise of implementing a components such as algorithms and resource proxies as assemblies is further supported by the practice of sandboxing, i.e. running code in a restricted environment [28]. Even though our implementation does not support this, it is possible to have it supported in the furture based on the current component-based implementation of e.g. algortihms and resource proxies.

5.1.2 Application Programming Interface

For allowing users to develop their own customized solutions, we have chosen to provide a library of commonly used structures and methods. The Application Programming Interface (API) provides classes, interfaces, delegates and enumerations all needed for creating an assembly to be used with UPPAAL PARMOS. While this subsection only contains a subset of all the classes, more is found in Appendix 8.

SettingCollection

The *SettingCollection*, shown in Figure 5.3, is a class, created to ease the passing of settings, and is utilized in both the implemented Algorithms, as well as the Resource.

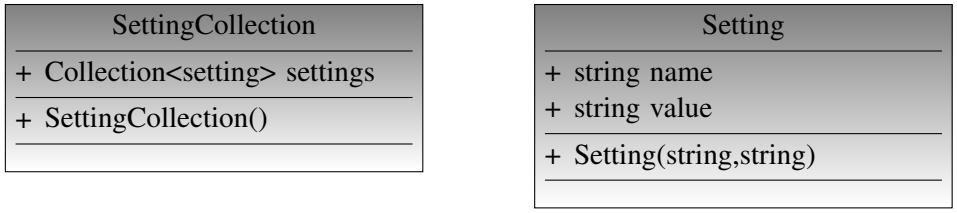


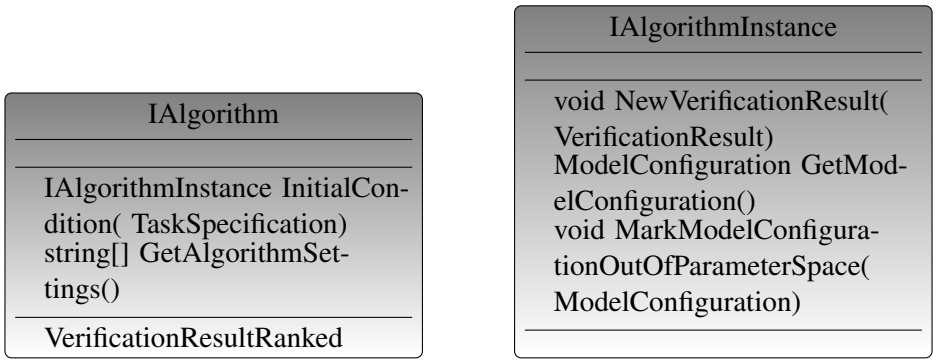
Figure 5.3: Class diagram of the SettingCollection class

IAlgorithm interface

The *IAlgorithm* interface serves to provide information of the algorithm, as well as to provide a reference to an instance of the *IAlgorithmInstance*. The class diagram is shown in Figure 5.4(b).

InitialCondition is a method called by the scheduler to initialize an instance of an algorithm.

GetAlgorithmSettings is used to obtain the settings the algorithm accepts



(a) Class diagram of the IAlgorithmInstance interface

(b) Class diagram of the IAlgorithm interface

Figure 5.4

IAlgorithmInstance interface

The *IAlgorithmInstance* class, shown in Figure 5.4(a), is the actual algorithm implementation, responsible for calculating new model configurations to be verified. It contains three methods, which are described below

NewVerificationResult informs the algorithm of the results of a previously supplied model configuration

GetModelConfiguration will return a new model configuration to the scheduler whenever called.

MarkModelConfigurationOutOfParameterSpace informs the algorithm that a specific model configuration is outside the parameter space.

IResource interface

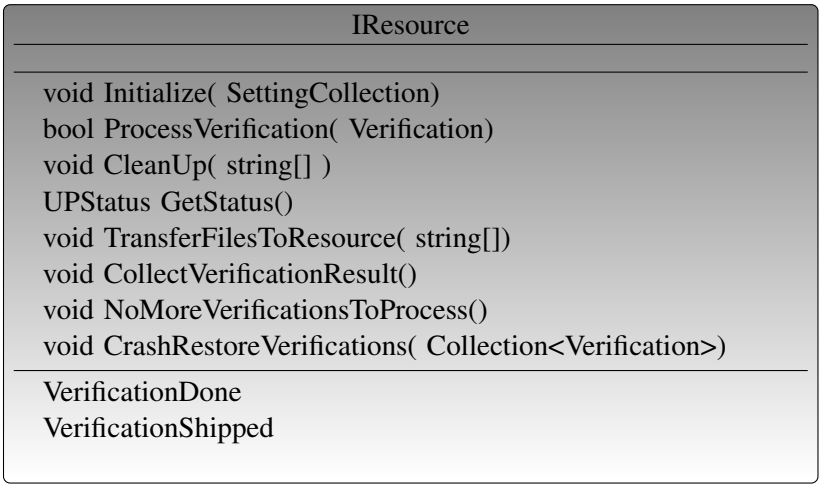


Figure 5.5: Class diagram for the IResource interface

The *IResource* interface, shown in Figure 5.5, specifies the methods which a Resource proxy must implement, which mainly concerns sending files or commands to and from the resources which it serves as a proxy for, and are described below.

Initialize is called once by the Resource Handler to initialize the loaded resource proxy.

ProcessVerification is called by the Resource Handler when a new verification needs to be shipped.

CleanUp is called by the Resource Handler when a clean-up need to be performed.

GetStatus is called by the Resource Handler to retrieve the status of the resource.

TransferFilesToResource is called by the Resource Handler to when files need to be transferred to the resource.

CollectVerificationResult is called by the Resource Handler when a retrieval of verification results should be performed.

NoMoreVerificationsToProcess is called by the Resource Handler when there are no verification to process in the current task.

5.1.3 Task specification

The layout of the task specification file is split up into 4 sections, each of which is defined below. Each section start of with the name of the section, and the contents of that section is enclosed in curly brackets. General for each section (unless otherwise noted in the individual sections) is that each line is delimitered by a semicolon, all whitspace is ignored and c-style comments are allowed, and similarily ignored.

```
task
  = 'parameters{' parameters '}'
    'constraints{' constraints '}'
    'objectives{' objectives'}'
    'optimization{' optimization {}'}'
  :
```

Listing 5.1: General task specification file gramma

Parameters

The parameters section specifies the parameters making up the parameter space, taking the form of a *from* value, a *to* value and a *step* value, with the *from* and *to* values describing the lower and upper bound respectively, and *step* defining the step size. Each such definition is prefixed with a parameter name. This is used to hold the current value of that parameter, for use in later expressions in the other sections of the task specification file.

```
parameters
  = (parameter ';' )*
  :
parameter
  = ID '={' number ':' number',' number'}'
  :
number
  = UNARYINTOPERATOR? INT
  :
INT : '0'..'9'+
  :
UNARYINTOPERATOR
  = '-'
  :
ID
  = ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
  :
```

Listing 5.2: Task specification file parameter section

Constraints

The constraints are used to limit the parameter space, and take the form of a boolean expression which must be true, for the parameter set to be considered valid.

```
constraints
  = (constraint ';')*
  :
constraint
  = boolExpr
  :
boolExpr
```

```
  = boolTerm (OR boolExpr)?
  :
boolTerm
  = boolean (AND boolTerm)?
  :
boolean
  =
  UNARYBOOLOPERATOR?
  (
    BOOL
    | ID
    |  LPAREN boolExpr RPAREN
    | '['compare']'
  )
  :
compare
  = intExpr COMPARATOR intExpr
  :
intExpr
  = intTerm ((SECONDORDERINTOPERATOR intExpr) )?
  :
intTerm
  = intPow ((FIRSTORDERINTOPERATOR) intTerm  )?
  :
intPow
  = factor (INTPOWOPERATOR intPow)?
  :
factor
  =
  UNARYINTOPERATOR?
  (
    INT
    | ID('.'ID)?
    | intParens
  )
  :
intParens
  =LPAREN intExpr RPAREN
  :
AND
  = '&&'
  :
OR
  = '||'
  :
UNARYBOOLOPERATOR
  = '!'
  :

COMPARATOR
  = '<='|'>='|'<'|'>'|'!='|'='
```

```
59    :
60  BOOL
61    =  'true'|'false'
62    :
63  LPAREN
64    ='('
65    :
66  RPAREN
67    =')'
68    :
69  SECONDORDERINTOPERATOR
70    = '+'|'-'
71    :
72  FIRSTORDERINTOPERATOR
73    = '*'|'/'|'mod'|
74    :
75  INTPOWOPERATOR
76    = '^'
77    :
78  UNARYINTOPERATOR
79    = '-'
80    :
```

Listing 5.3: Task specification file constraints section

Objectives

An objective can be one of two things; either a true/false objective, which must be satisfied for the solution to be considered valid, and optimization objective, which are used by the optimization algorithms in order to optimize in the right direction. The true/false objectives simply consist of a boolean expression much like the constraints, whereas the optimization objectives consist of an optimization direction, telling the algorithm to either minimize or maximize the expression inside the parentheses. In the expressions of both types of objectives, it is possible to use several variables, such as the parameter values, but also the return values of queries. These variables take the form of *#doneinf*, where each query is named according to how the user specifies them in the GUI of UPPAAL.

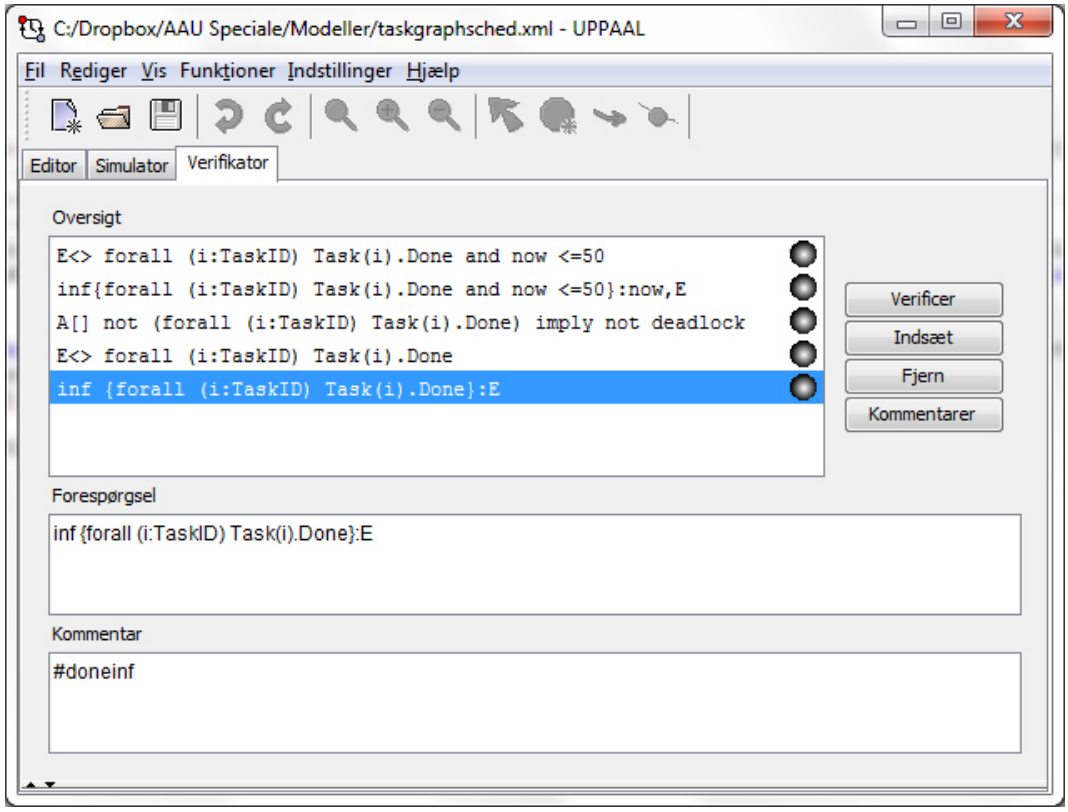


Figure 5.6: A screenshot of the UPPAAL GUI with an inserted variable in the "Kommentar" field.

Furthermore, UPPAAL queries files may contain *inf* or *sup* queries, which may return the value of one or more UPPAAL integers or clock values. These can be accessed using the same name notation as above, but appending a dot and the variable name. As illustrated in Figure 5.6, *#doneinf.E* refers to the variable called *E* in the fifth query in the query file.

```
1  objectives
2    = (objective ';')*
3    :
4  objective
5    = ( DIRECTION '(' intExpr ')' )
6    | boolExpr
7    :
8  DIRECTION
9    = 'min' | 'max'
10   :
```

Listing 5.4: Task specification file objectives section

Optimization

This section tells UPPAAL PARMOS which optimization algorithm to use, along with a set of options for the algorithm, in the form of key-value pairs. The reason for the *ID ('.'ID)+* part is that the namespace of the algorithm must also be specified.

```
ooptimization
  = ID('.'ID)+
    '{' (optimizationOption ';')* '}'
  :
optimizationOption
  = ID '=' (STRING|ID|INT)
  :

//match on anything inside '"' '"', except '"', unless it is
    prepended by a '\'
STRING
  : '"' .* '"'
  ;
```

Listing 5.5: Task specification file optimization section

Task specification file example

In listing 5.6 an example of a full Task specification file is given. It is the actual task specification file for the *taskgraphsched* task used in the test of UPPAAL PARMOS, which is found in Chapter 6.

```
parameters{
  NP={1:8,1};
  P={5:32,1};
  B={5:32,1};
}
constraints{
}
objectives{
  done50;
  notdeadlock;
  done;
  min(doneinf.E);
}
optimization{
  upalgorithms.PAES{
    ArchiveSize=10;
    DeadSpotMaxSize=100;
    Restarts=0;
  }
}
```

Listing 5.6: Example of the task specification file.

As can be seen, the task has three parameters, with a total parameter space of 6.272, making it a rather small task.

Further down, in the *objectives* section, three true/false objectives are defined. these must evaluate to true, for the solution to be considered valid. If they do not, all evaluation of the current solution halts, and it is discarded.

The section also contains three optimization objectives, which all attempts to minimize a certain expression, the first being the time used by the scheduler, to execute all tasks, the two last each are a measure of energy consumption per work unit, the first when all tasks must execute within 50 time units, the last without any regard for time.

A fourth optimization should have been defined, to find *a reasonable tradeoff between time, energy, and cost*. This is, however, quite difficult to quantify. Luckily, using the PAES algorithm, as defined in the optimization section, this last objective is easily fulfilled by the user, who will only have to compare a limited number of solutions, in this case 10, as defined by the *ArchiveSize* option.

5.2 Storage

In order to implement the designed storage for UPPAAL PARMOS, we have chosen to utilize a Relational Database Management System (RDBMS). This allows us to easily map the ERM design of our storage to the RDBMS. A range of different RDBMSs exists on the marked, and we have chosen to utilize the freely available MySQL Database System RDBMS [32]. MySQL runs on both Windows or Linux and thereby keeps UPPAAL PARMOS cross-platform supported.

MySQL Connector/Net

Since the .NET Framework does not support the MySQL Database by itself, we have utilized the MySQL provided implementation of a ADO.NET[29] connector to communicate with the database. This connector is called the MySQL Connector/Net[31], and provides all the functionality needed for UPPAAL PARMOS to store and retrieve data from a MySQL database. The version of the library used in the current version of UPPAAL PARMOS is release version 6.3.5.

5.3 Front-end

The front-end of UPPAAL PARMOS is implemented using ASP.NET of the .NET Framework. ASP.NET is a collection of web oriented technologies, used for e.g. building dynamic web pages and web services.

5.3.1 Graphical User Interface

In order for the GUI to communicate with the web service, a proxy is implemented for wrapping messages in a Simple Object Access Protocol (SOAP) envelope. The proxy to the web service is generated using a tool available in the Visual Studio development environment. This tool retrieves the Web Services Description Language (WSDL) from the web service, which it uses for automatic generation of C# proxy code [40]. The allows us to view the web service as was it a class in the namespace.

As illustrated in Figure 5.7, 5.8 and 5.9 the GUI provides features for submitting and viewing information about a task. For developing these feature, we have implemented the needed procedures, as described in Section 3.5. The web service implementation is found in the appertaining code of this thesis.

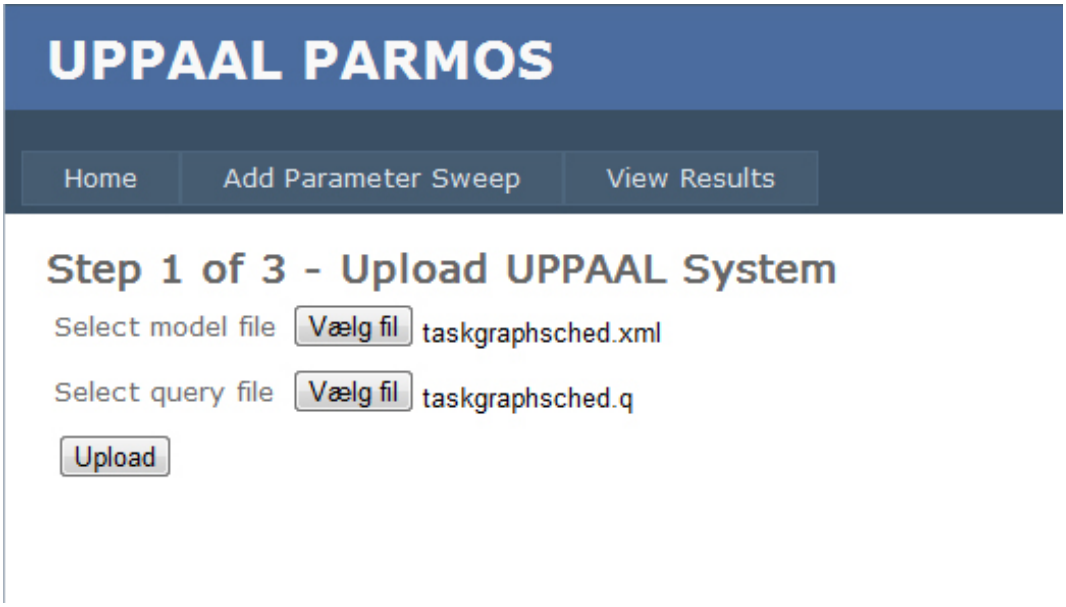


Figure 5.7: Screenshot of the first step in adding a new task.

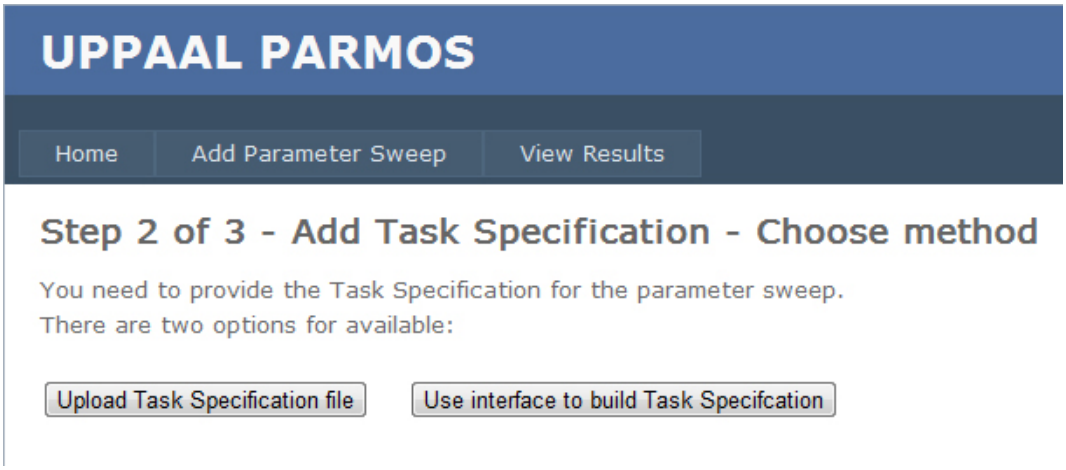


Figure 5.8: Screenshot of the second step in adding a new task.

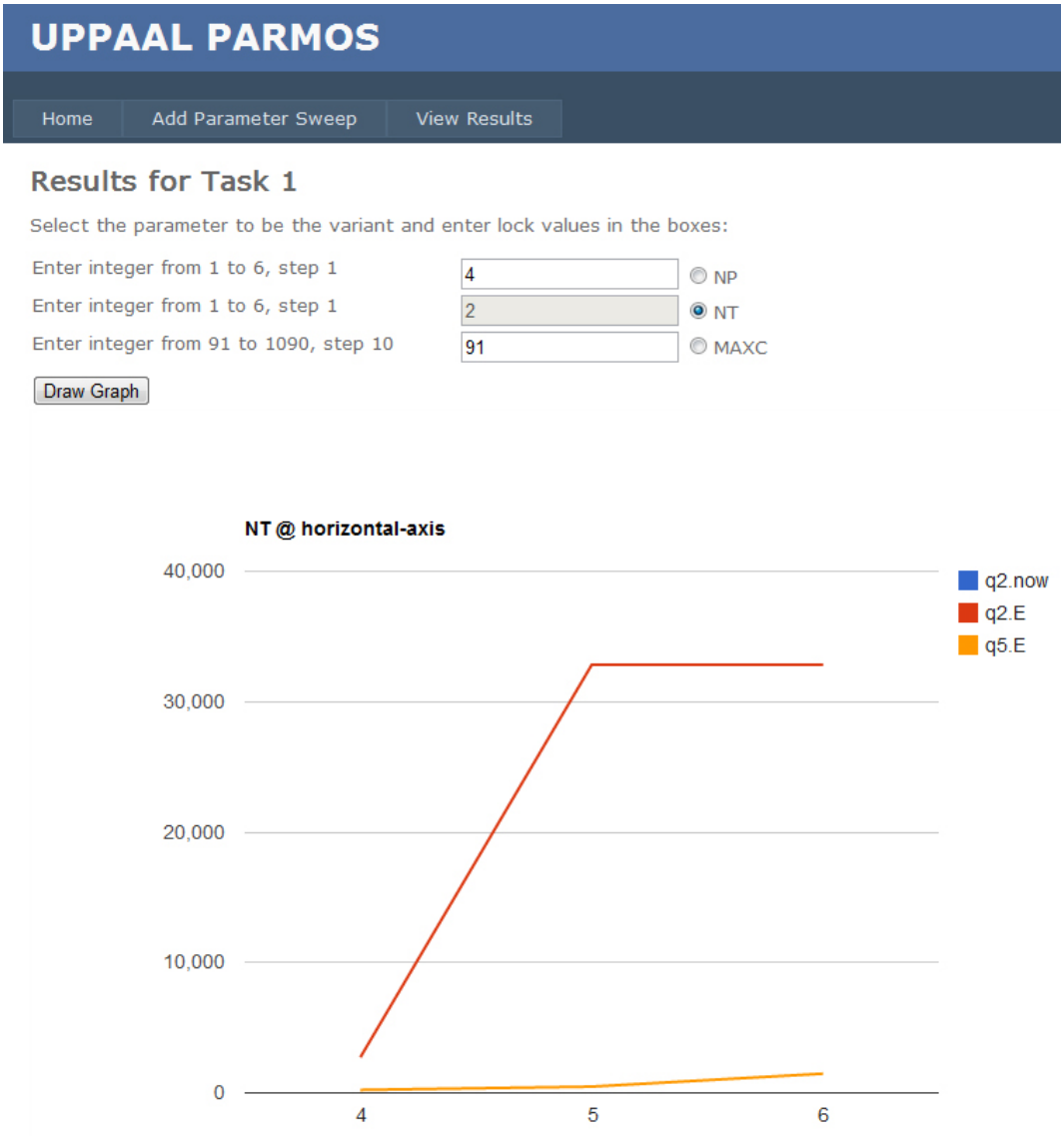


Figure 5.9: Screenshot of the verification result viewer.

5.4 Algorithms

Below, the implementations of the three algorithms described in Section 4.

Common to them is the implementation of the data structure in which the state of the Model configurations is stored, i.e. whether it has been scheduled or not. The class diagram of this data structure, called *ConcurrentInclusionSet* is shown in figure 5.10. This is implemented using a bit array, the size of the entire parameter space.

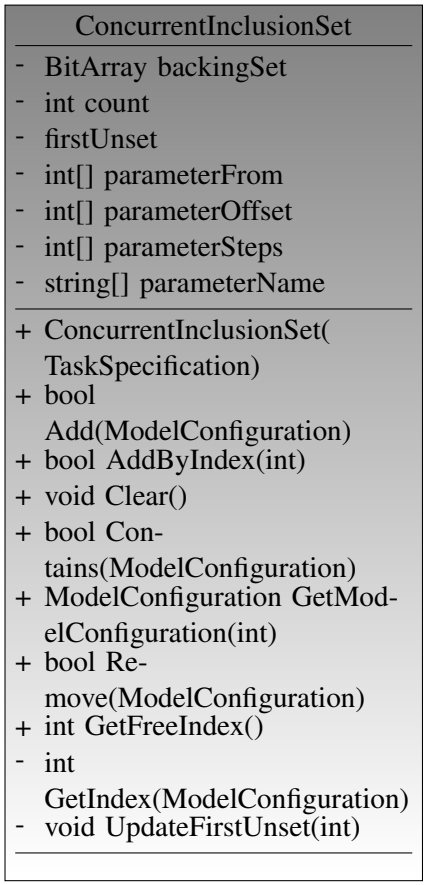


Figure 5.10: Class diagram of the ConcurrentInclusionSet data structure

This allows for the state of all model configurations to be stored, using relatively little space. The array is a one dimensional array, where the index is calculated, using the formula provided in Equation 5.1, where P is the set of parameters, and O is the offset pertaining to each parameter, calculated using the formula shown in 5.2, although this formula does not take step size into account, as it would complicate the formula unnecessarily, and it is a trivial expansion.

$$index = \sum_{i=0}^{|P|} P_i \cdot O_i \tag{5.1}$$

$$\begin{aligned} O_i = &(max_{P_0} - min_{P_0} + 1) \\ &\cdot (max_{P_1} - min_{P_1} + 1) \\ &\vdots \\ &\cdot (max_{P_i} - min_{P_i} + 1) \end{aligned} \tag{5.2}$$

5.4.1 Brute-force

The Brute-force algorithm is a simple algorithm iterating over all possible model configurations, and is implemented in order to verify the optima found by the other algorithms, as it is guaranteed to find the global optima.

5.4.2 Hill-Climbing

A simple Hill-Climbing algorithm is implemented as described in 4.3.3. In order to find out when to stop, a *set of thresholders* is generated from a *threshold* value, and the current best solution, when ever that changes. Thresholders are essentially the neighbourhood of the current best solution, which size is defined by a threshold setting of the algorithm. Thresholders are removed from the set if their result is known, and it is not better than the current best one, are out of the parameter space, or fall for constraints. This way, when the threshold set is empty, all neighbours of the current best solution in the vicinity defined by the threshold value, has been found worse than the current best one. Thus threshold has been fulfilled, and the algorithm ends search.

5.4.3 Simulated Annealing

The SA algorithm is not implemented as specified in the initial design in Section 4.4.3. That design included two cost function calls that in each transition would block the generation of new model configurations. Instead of this, we have implemented the acceptance criterion from Definition 5. This criterion is used to compare a current best solution, with each of the received candidate solutions from the verification process.

In order to stop the algorithm a concept of a *dead spot counter* is implemented. This counter is on initialization sat to a predefined value and decremented each time a received candidate solution fails the acceptance criterion. The dead spot counter reset when a candidate solution becomes the current best solution. When the dead spot counter equals zero or there are more candidate solutions, the algorithm will stop.

5.4.4 Pareto Archived Evolution Strategy

The PAES algorithm is implemented as described in 4.5, with the addition of both speculative testing, as described in Section 4.6.2, and random restart as in the Hill-Climbing algorithm in Section 4.3.

In order to minimize the impact on the scheduler of returning a job result, the algorithm utilizes a buffer, into which the result is loaded, and the function call immediately returns. A thread then regularly empties it, in order to evaluate the results.

When evaluating a result, if the Pareto Dominance algorithm cannot establish clear dominance between two results, it will compare it against the results in the archive, as specified in the PAES Algorithm 4.5.1. If clear dominance still cannot be established, the diversitycheck will be employed. This will check whether the new result will introduce greater diversity into the archive. The diversitycheck is implemented as the normalized Manhattan distance of the results. Thus, the result of evaluating the expression of each optimization query is normalized compared to the upper and lower bound of the expression, when evaluated for all results in the archive as well as the new result. This produces a number between 0 and 1 for each expression, which is then summed over all expression, to produce a number representative of the distance between the results. This is shown in Equation 5.3.

$$dist = \sum_{i=0}^{|R|} \frac{max_i - min_i}{R_i - min_i} \tag{5.3}$$

$$min_i = Min(\{\forall A \in archive : A_i\} \cup R_i) \tag{5.4}$$

$$max_i = Max(\{\forall A \in archive : A_i\} \cup R_i) \tag{5.5}$$

PAES also implements the concepts of a dead spot counter as described in Section 5.4.3.

5.5 Resource proxies

In order to test our PSA we have implemented as resource proxy, for the cluster sat at our disposal.

5.5.1 Simple Linux Utility for Resource Management

As mentioned in Section 2.3.1 on page 11, SLURM is an open source resource manager, which enables users to execute jobs on a cluster. The SLURM resource proxy is a resource proxy that is able to utilize the resources of a SLURM enabled cluster, through the SLURM resource manager.

Batch

To be able to group a portion of verifications, we establish a concept named *batch*. A batch is essentially the specifications for processing a number of verifications as a single SLURM job on a single node.

We also establish a concept of *batch size*, to be the number of verifications specified by a given batch. Two factors of a batch make the batch size:

- The number processes to run on the node to process the batches.
- The number of verifications that should be processed sequentially in each process.

Trivially, the product of these two factors define the batch size. By grouping a verifications, batches are used to reduce overhead, and thus make better use of resources. The larger the batch size, the less batches are needed to process a certain amount of verifications, and thus fewer batches need to be transferred and handled by SLURM.

Settings

To make use of a SLURM resource, and control how it is used, a number of settings must be applied to the SLURM resource proxy. This section consists of an overview of these settings. Additional details on how settings affect performance, is described in the following section.

Host A Domain Name System (DNS) resolvable address or an Internet Protocol (IP) address of the SLURM resource.

Port Is an optional parameter, specifying the port number for the Secure Shell (SSH) connection to the SLURM resource. Not specifying this parameter, will default to the SSH default port 22.

Username The username used to access the SLURM resource.

Password The password needed for authentication of the given *username*.

Cores Defines the number of cores available for a batch on each node at the SLURM resource.

Verifications per core Defines the number of verifications to be processed sequentially, for each process started in a batch.

Batches Defines the maximum number of batches the resource proxy is allowed to have present at the resource.

Batch options Defines an optional string passed directly to the `sbatch` command as extra arguments when running batches.

Performance optimization

The settings *host*, *port*, *username* and *password* are trivial connection settings for establishing a SSH connection to the resource, through which the resource proxy communicates with the resource.

Passing *batch options* to the `sbatch` command can prove very useful if e.g. the SLURM resource is divided into partitions, and one want the batches to run in another partition than the default one.

The remaining settings *batches*, *cores* and *verifications per core*, however require further explanation on their influence on the behaviour of the resource proxy and UPPAAL PARMOS as a whole.

Tuning the values for *batches*, *cores* and *verifications per core* right, can provide a large increase in performance. How to tune them depends on the task being processed by UPPAAL PARMOS.

Collectively these three settings define the number of verifications the SLURM resource proxy will strive to have at the resource at all times.

The resource proxy will always try to keep as many batches at the resource as it is allowed. Since we have from the batch definition on the facing page, that $cores \cdot verifications\ per\ core$ defines the batch size, we trivially have that $batches \cdot cores \cdot verifications\ per\ core$ is the total amount of verifications that it will try to keep at the resource. Figure 5.11 illustrates the interrelation between *batches*, *cores* and *verifications per core*.

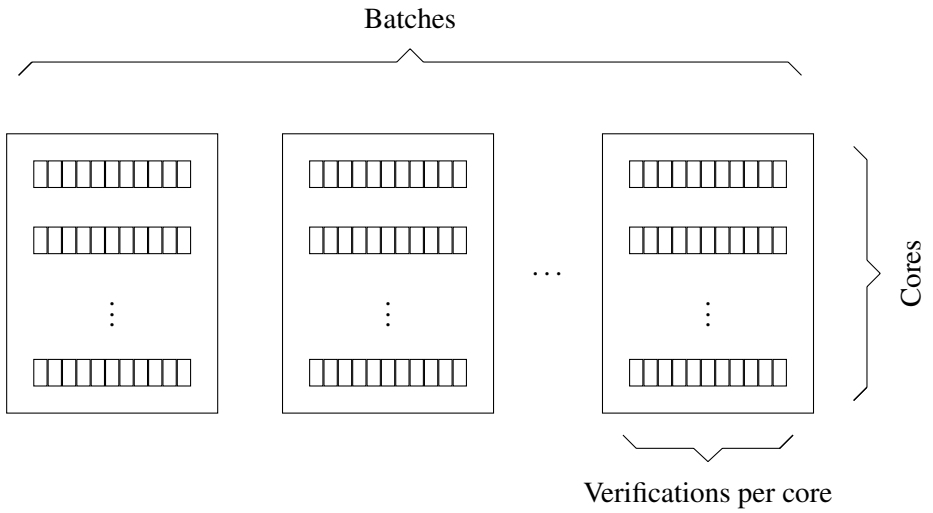


Figure 5.11: Illustration of the interrelation between the *batches*, *cores* and *verifications per core* settings. The big boxes illustrate batches, the lines of conjoined small boxes illustrate processes, and the small boxes illustrate verifications.

A batch will always be processed by a single node on a SLURM resource, so naturally in order to utilize several nodes, at least equally many batches should be dispatched. Usually nodes have multiple cores however, and SLURM may therefore choose to process several

batches on a single node. This is good for performance, since processing verifications in parallel is naturally faster than processing them sequentially.

However, overhead from SLURM batch handling can be reduced, by letting each batch occupy more cores on a single node. The value of *cores* should therefore ideally be set to the number of cores available at each node — providing that all nodes have the same amount of cores.

The value of *cores* is used by the resource proxy:

- to define the number of processes to be started at the node.
- to tell SLURM that the specified number of cores is required to run the batch.

The last of the two, cause that setting the value of *cores* higher than the available amount of cores, will block batches from being processed. SLURM will conclude that insufficient resources are available, and will therefore not let them run at all.

Even though *cores* is set to the amount of cores present on each node, it may still prove beneficial to set the value of *batches* higher than the amount of nodes. Generally it is likely to be beneficial to set the value higher than there are resources available for. This is because of he behaviour of the Service. When a resource is considered occupied, because it holds its defined maximum capacity of verifications, a new batch will not be dispatched to the resource before one has been retrieved from it. SLURM however, makes it possible to submit a batch, even though there are no resources available for processing, so increasing the value of *batches* to be higher than the available amount of nodes, thus sending more batches than there processing resources for, will make SLURM queue the additional amount of batches. The extra batches will then wait for resources to become available, and start processing while the results of previous batches are retrieved. Thus overhead from resources standing idle between batches is reduced. However setting the value to the amount of nodes available, will result in overhead, since UPPAAL PARMOS will not send a new batch before having retrieved one. A node will then stand idle while the batch is being retrieved, and a new one is being dispatched. Setting the value of *batches* higher than the amount of available nodes, makes it possible to ensure maximum load on the resource. This is because the extra batches that are shipped, will be queued at the SLURM resource, and executed while the resource proxy retrieves the results of previous batches, calculates new model configurations and dispatch new batches.

5.6 Software platform

The current version of UPPAAL PARMOS has the following requirements for running:

- .NET Framework 4.0
- MySQL Server 5.5 (Community Edition)

Note that the choice of .NET Framework 4.0 excludes the use of Mono, and thus makes a requirement for a Microsoft Windows operation system. The reason for this choice is that the SSH.NET Library, we use for SSH connectivity, require .NET 4.0, and Mono currently

does not support all functionality in .NET 4.0. See the section about this library on the current page for more information on this issue.

5.6.1 Third-party libraries

In the implementation of UPPAAL PARMOS, a number of third party libraries, has been utilized, in addition to the ones included in .NET 4.0. This section provides an overview of these libraries and their usage.

ANother Tool for Language Recognition (ANTLR) Parser Generator

ANTLR[38] is a language recognition framework. It supports defining a grammar in Extended Backus–Naur Form (EBNF), and provides runtime binaries for several languages.

The version of the library used in the current version of UPPAAL PARMOS is 3.3.

We use this library for parsing our task specification file.

SSH.NET Library

The SSH.NET Library[39] is a .NET library for SSH and SSH File Transfer Protocol (SFTP) connectivity. On the homepage of the library, the creator states in the introduction that:

> This project was inspired by Sharp.SSH library which was ported from java and it seems like was not supported for quite some time. This library is complete rewrite using .NET 4.0, without any third party dependencies and to utilize the parallelism as much as possible to allow best performance I can get.

Initially we used the Sharp.SSH library[12] for SSH and SFTP connectivity, which enabled us to host UPPAAL PARMOS in a Linux + Mono + MySQL environment. We were however forced to change because we experienced instabilities using the library under high load.

The choice then fell on the SSH.NET Library, and as previously mentioned, Mono does not support all functionality needed by the library. This is, to our knowledge, primarily functionality used for treating large numbers in cryptographic functions.

Additional to the functionality of the Sharp.SSH library, the SSH.NET Library supports asynchronous calls with callback, and also allows us to download the contents of files directly to memory. The version of the library used in the current version of UPPAAL PARMOS is the one found in the repository of [39] revision 7733.

6 Test

Testing is required to determine the successfulness of achieving our design goal of performance, described in Section 3.1.

This chapter describes every aspect of our testing of UPPAAL PARMOS.

6.1 Test environment

This section describes the general aspects of the environment in which our test were conducted.

6.1.1 Resource

As primary resource for our tests, we have used the Fyrkat cluster of Aalborg University[2], introduced in the preface. Since Fyrkat is a SLURM enabled cluster, our SLURM resource proxy, described in Section 5.5.1, is used for the testing. The overhead tests in Section 6.3 on page 83 and the scalability tests in Section 6.4 on page 89, therefore test the overhead and scalability of UPPAAL PARMOS in conjunction with this resource proxy.

Fyrkat is divided into partitions of different worker node types. The node types shown in Table 6.1, are the ones that have been utilized for our test.

	Sister nodes	**Killing nodes**
CPU	Xeon X3220 quad core 2.40GHz	Two Xeon E5345 quad core 2.33 GHz
Memory	8GB	16GB
NIC	Gigabit ethernet	Gigabit ethernet and Infiniband
OS	Ubuntu 10.04.2 LTS	Ubuntu 10.04.2 LTS

Table 6.1: Specifications for utilized Fyrkat worker nodes [2]

A total of 5 Sister and 14 Killing nodes have been at our disposal. However, even though there are more Killing nodes, and they provide more cores for processing, most of our testing have taken place using the Sister nodes, as the Killing nodes were only available in the final week of our project period.

6.1.2 Server

For testing purposes we have used a test server, installed with Windows 7 Enterprise 32 bit (Service Pack 1) and the required software described in 5.6.

Our test server has the following hardware specifications:

CPU	Intel ® Core™2 Duo E8400 @ 3.00 GHz
RAM	4,00 GB (3,46 GB usable due to 32 bit OS)
NIC	100 Mbit/s
Storage	Seagate Barracuda 7200.10 SATA 3.0Gb/s 320-GB Hard Drive (ST3320620AS)

6.1.3 Network

Our test server is placed at Cassiopeia, the building of residence for the Department of Computer Science at Aalborg University, and connected to the campus network. 100 Mbit/s is the minimal bottleneck speed of the campus network, however since our data transfers consist mostly of small pieces of text, we find latency to be the of greater concern.

From running the Windows `tracert` command on the server, targeting the Fyrkat cluster, we can see that there are 4 hops between the server, and the cluster, and all hops are internal to the Aalborg University network. The output of the `tracert` command is shown in Listing 6.1.

```
C:\Users\Administrator>tracert fyrkat.grid.aau.dk

Tracing route to fyrkat.grid.aau.dk [130.225.196.202]
over a maximum of 30 hops:

  1    <1 ms    <1 ms    <1 ms  h253.net.klient.slv.site.aau.dk
     [172.25.23.253]
  2    <1 ms    <1 ms    <1 ms  aau-gw1.aau.dk [130.225.52.1]
  3    <1 ms    <1 ms    <1 ms  gi2-3.aau-edge1.aau.dk
     [192.38.59.66]
  4    <1 ms    <1 ms    <1 ms  fyrkat.grid.aau.dk
     [130.225.196.202]

Trace complete.
```

Listing 6.1: The output of the `tracert` command from the test server to the Fyrkat cluster

The output in Listing 6.1 also shows that there is very little latency between the server and the cluster, thus removing the concern for the latency.

6.2 Settings

As described in Section 5.5.1, three settings of the SLURM resource proxy, affect the throughput and scalability of UPPAAL PARMOS, namely *cores*, *verifications per core* and *batches*. Furthermore we have made tests on modifying the retrieval timer of the Scheduler. The effect of modifying these settings has been tested in Sections 6.3 and 6.4. While the effects of one of these settings is tested, the others usually have fixed values. Section 6.5 concerns itself with testing the implemented algorithms, described in Section 5.4.

The TGS UPPAAL system, described in Sections 2.1.1 and 2.1.2, is used for most testing. Another system, EKC described in [19] has been used for throughput benchmarking described in 6.6.

Tests are conducted in sets, varying on only one setting between them, however a small exception is made for the scalability tests (See Section 6.4 for details). A set usually translates into a graph, so multiple sets can easily be compared in a graph.

6.3 Overhead

The ability to get jobs processed with low overhead, is one of the most important goals of UPPAAL PARMOS. Therefore great effort has been put into reducing it.

Overhead in UPPAAL PARMOS may stem from a range of different sources. Internally in UPPAAL PARMOS, the most significant overhead exists in the selection of new parameters, when an ID is queried from the database for every verification that is prepared. All other database requests are buffered, and executex asynchroneously, and does not really introduce any overhead. Other than this, internal overhead can be introduced from the employed algorithm being slow at selecting new parameters for verification. Time constraints have kept us from conducting profiling of UPPAAL PARMOS at runtime, so this is a claim based on our knowledge of UPPAAL PARMOS rather than empirical data.

Observations however show, that the biggest source of overhead overall, stems from the handling of resource. The following sources of overhead have been identified in relation with resources, and can be alleviated by tweaking settings in the resource proxy:

Network connection to the resource The SSH protocol used for communicating with the resource, introduce some overhead for every command string that is to be executed at the resource. This command string has a limited size of maximum 32 KB, so traffic is not likely to be the problem, the overhead thus lies in initiating the command string, and is increased for every command string that is to be executed. Lowering the amount of command strings to be executed is thus preferred.

A command string is executed for every batch dispatched to the resource, so the more batches a task requires, the more overhead. Thus lowering the amount of batches lowers the overhead. Besides from an employed algorithms ability to choose verifications well, increasing the batch size can be used to lower the amount of batches dispatched in relation with a task. This can be achieved by increasing the value for the *cores* and *verifications per core* setting of the SLURM resource proxy.

Batch handling at the resource Along with the above described overhead of command strings, SLURM also introduce some overhead for every batch, in its internal handling of it. This is again a per batch overhead, and thus again calls for lowering the amount of batches.

Results waiting at the resource When a batch has finished processing its verifications, its results should ideally be retrieved immediately, such that the employed algorithm may benefit from them, when selecting new parameters. Since UPPAAL PARMOS employs a loop for periodically checking for results, this loop should ideally check

for results as soon as a batch has finished processing, in order for the results not to wait too long at the resource. There is no setting available in the SLURM resource proxy for this, but we have made experiments with three different settings of the retriever loop.

Nodes/cores standing idle New batches cannot be dispatched to the resource, before results of others have been retrieved. This is yet another reason that results should be retrieved as fast as possible. Another way of dealing with this issue however, is increase the value for the setting *batches*, so extra batches are queued up at the resource. SLURM can then start processing of these batches, as soon as others finish. Queueing batches however, forces the algorithm to select more parameters for verification, with less knowledge of results than if no batches were queued.

The rest of this Section present results for tests that show configuration tendencies related to the issue of reducing the overhead of UPPAAL PARMOS.

6.3.1 Test setup

An exhaustive search, using the brute-force algorithm implemented in UPPAAL PARMOS, is used for these tests. In order for overhead to have maximum impact during testing, it is preferable that verifications have as little processing time at the resource as possible, yet still generate result data to be retrieved and parsed. This can be used to ensure that the throughput capability of the Service is stressed as much as possible from overhead, and tendencies from changing settings between experiments may show better.

To ensure this very low processing time of verifications, a task has been created using the TGS UPPAAL system, and the task specification shown in Listing 6.2.

```
1  parameters {
2    NP = {1:1,1};
3    NT = {1:1,1};
4    MAXC = {91:1090,1};
5    }
6
7  constraints {
8    NP <= NT;
9    }
10
11 objectives {
12   done50;
13   notdeadlock;
14   done;
15   min(done50inf.now);
16   min(doneinf.E);
17   min(done50inf.E);
18   }
19
20 optimization{ upalgorithms.BruteForce{} }
```

Listing 6.2: The task specification used for overhead testing.

The settings in this task file obviously describes a parameter space of a 1000 parameter combinations, as `NP` and `NT` only have one combination each, and `MAXC` has a 1000. Since `NP` and `NT` are locked at their lowest values, the processing time of verifications will be minimal. The reason why `MAXC` starts at 91, is that lower values of `MAXC` makes UPPAAL fail to verify the model, which would lessen the amount of data to be retrieved and parsed from the resource.

To make comparable experiments with a higher work load, we have inserted a loop in every part of the scripts that process a verification, such that the verification is processed multiple times discarding results first, after which a final processing is run to get the results. The loop is illustrated in Listing 6.3.

```
1 for (( i = 0; i < 49; i++ )); do
2   ./verifyta [arguments] &> /dev/null
3 done
```

Listing 6.3: Structure of the loop inserted to process a verification multiple times.

Two types of overhead experiments have been conducted, respectively testing the impact of the *batches* and *verifications per core* settings of the SLURM Resource proxy. Furthermore, results of the overhead effects of the retriever loops behaviour in UPPAAL PARMOS, has also been tested. Further effects of the retriever loop is described in Section on scalability, as we, during these test, also got results on its significance to overhead.

6.3.2 Retriever loop

The retriever loop in UPPAAL PARMOS is far from optimal, and is an area of UPPAAL PARMOS that needs further work. During our testing of scalability, described in Section 6.4 on page 89, we tested 3 different retriever loops — one with a fixed interval, and 2 with auto-adjusting variable intervals, one tuned to a specific workload, the other not.

In both the auto-adjusting retriever loops, the actual processing time for batches are used for predicting when new results are ready.

$$T = \frac{T_{actual} + T_{old}}{2} \tag{6.1}$$

$$T = \frac{T_{actual} + T_{old}}{2} + c \tag{6.2}$$

For the first auto-adjusting retriever loop, Equation 6.1 is used to calculate a new loop time every time a batch is retrieved. This way of setting loop time has however a tendency of undershooting the points in time where results are ready by a small margin. This results in looping twice before retrieving results again, possibly making results wait longer at the resource. Figure 6.2 shows an illustration of this tendency.

The second auto-adjusting retriever loop uses Equation 6.2 to calculate the new loop time. A constant value added to the calculated value, is used to tune the retriever loop to avoid undershooting, and overshoot a little instead. Thus if it is tuned right, the result will wait a shorter amount of time before it is retrieved. Figure 6.1 shows an illustration of this.

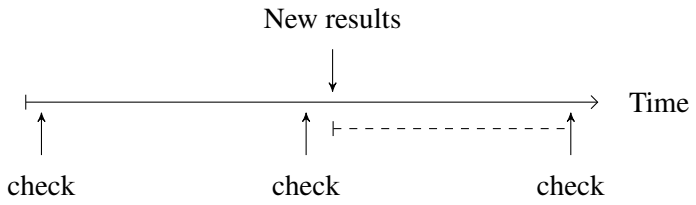


Figure 6.1: Time line illustrating the effect of the retriever loop undershooting results by a small margin. The dashed line is the time the results wait at the resource.

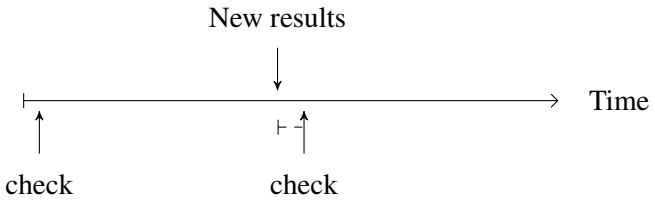


Figure 6.2: Time line illustrating the effect of the retriever loop overshooting results by a small margin. The dashed line is the time the results wait at the resource.

The third retriever loop with a fixed interval simply checks for results every second. This way results will wait a maximum of one second for being retrieved, and overhead from results waiting for being retrieved is thus very low, but a lot of checks are made for long running tests, and a high latency could give issues with requests queueing up. This is not a problem in our test setup, and we have used this retriever loop to minimize the overhead influence from waiting results in the tests not examining the effects of the retriever loop. A small factor that may induce delay, is that it is not possible to dispatch and retrieve jobs at the same time.

We will, for the remainder of the thesis refer to these three retriever loops by name. The first will be referred to as the *untuned* retriever loop, the second will be referred to as the *tuned* retriever loop and the third will be referred to as the *fixed* retriever loop.

Both the auto-adjusting retriever loops tend to let results wait longer at the resource than the fixed loop. However, they require notably fewer checks than the fixed loop.

In testing this, 100 batches was made using the task specification from Listing 6.2, with *verification per core* set to 10. Three sets of experiments have been made, where every model configuration verification was processed 50, 250 and 500 times for set 1, set 2 and set 3 respectively.

The values for *batches* and *cores* were both 1, to impose the greatest possible waiting time, by only having one batch at the resource at all times. This experiment was conducted for all three retriever loops, with the tuned retriever loop adding a constant of 1. The tuning is made from multiple experiments with verification processed 50 times, and the results from these experiments are shown in Table 6.2.

The results are shown in Table 6.3. It is obvious that the untuned loop must overshoot, since it is takes quite a bit longer to complete than the other two in all three cases. The fixed loop, does logically check every second, and thus make checks correspondingly, while we unfortunately have no numbers for how many times the auto-adjusting loops check.

Constant	3	2	1.5	1	0.75
Time	1904	1668	1566	1488	1546

Table 6.2: Experiments used to of tune the tuned auto-adjusting retriever loop

Also, as shown in the Section on test of scalability (6.4), the tuned loop scales bad, and thus must be re-tuned for each specific task in UPPAAL PARMOS. Therefore we conclude that this is an area of UPPAAL PARMOS that require an effort to improve.

Loop	Set 1 time	Set 2 time	Set 3 time
Fixed	1573	5359	10558
Untuned	2563	5943	12359
Tuned	1488	7125	10583

Table 6.3: Running times of experiments on retriever loops. Times are in seconds.

6.3.3 Batches

Testing of the impact of the *batches* setting was conducted on the 5 Sister nodes of Fyrkat. The goal of these tests is to test the *batches* settings potential for reducing overhead from nodes standing idle at the resource.

Prior to testing, the impact of increasing the value for *batches* was that the throughput would rise, until a certain point, where it will just cause batches to queue up, and wait for longer time. Thus the increase in throughput will stall. This point was expected to be reached relatively quickly. A graph illustrating this expectation is depicted in Figure 6.3.

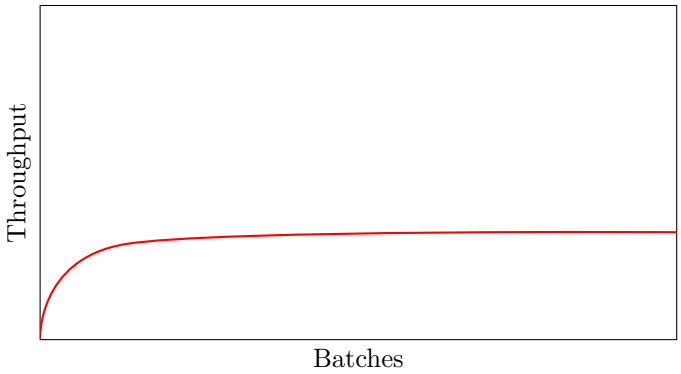


Figure 6.3: Graph illustrating our expectations for the overhead tests, varying on the value of *batches*.

Tests were conducted on Sister nodes, with settings for *verifications per core*, and *cores* fixed to 10 and 4 respectively, and values for *batches* was tested starting at 5 through 20. Since a Sister node has four cores, a batch that takes possession of 4 cores, will then take possession of a full node. Testing with lower values of *batches* than 5, will not allow batches

to queue, since there will always be nodes ready to process them. Thus, the starting value for *batches* was equal to the number of Sister nodes available.

One sets of experiments was conducted with a loop for inducing work. Each verification was processed 500 times. Without the loop, verifications would finish processing too fast for the queueing to occur, since the value of *verifications per core* was set to 1 to test with as many batches as possible.

Results are shown in 6.4, and show something in the lanes of the expected. However results are coloured from tests being run with the untuned retriever loop, and we expect this to be the source of the spiking from *batches* values 12 through 16. Unfortunately time did not permit us to rerun the tests with the fixed retriever loop.

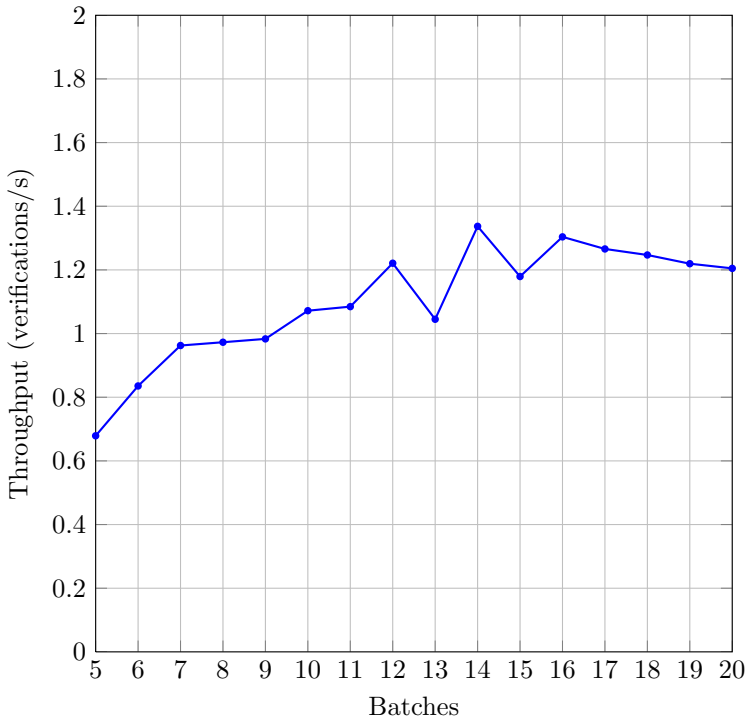


Figure 6.4: Graph illustrating our expectations for the overhead tests, varying on the value of *verifications per core*.

6.3.4 Verifications per core

Increasing the value for *verifications per core* is expected to decrease the amount of overhead, corresponding to it's reduction of the amount of batches, as described in Section 5.5.1.

Figure 6.5 shows a graph illustrating our expectations for increasing the value for *verifications per core*. The graph is generated with values for *verifications per core* from 1 through 2000, using the formula in Equation 6.4. 1000 is the total number of jobs to be processed, 3 is the time for batch handling overhead, and 0.01 is the job processing time. Equation 6.3 is used to calculate the number of batches with the *verifications per core* value in every sample, and a *cores* value of 1.

$$maxbatches = \left\lceil \frac{1000}{verifications\ per\ core \cdot 1} \right\rceil \tag{6.3}$$

$$Throughput = \frac{1000}{maxbatches \cdot 3 + 1000 \cdot 0.01} \tag{6.4}$$

By a *verifications per core* value of 1000, increase in throughput stops, since the value of *verifications per core* reaches a point where it is equal to or higher than the number of verifications defined by the task, and all verifications can thus be dispatched in one batch.

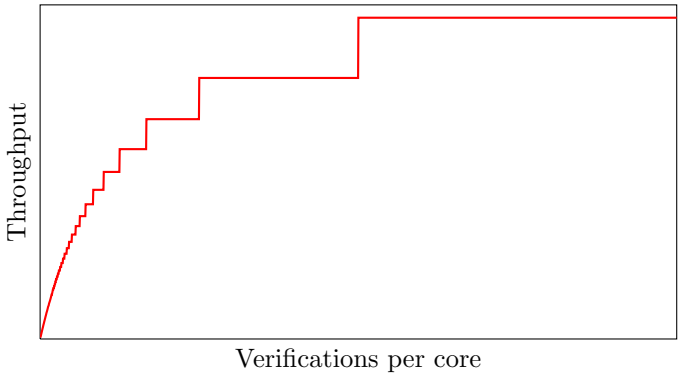


Figure 6.5: Graph illustrating our expectations for the overhead tests, varying on the value of *verifications per core*.

Time, however did not permit us to run 2000 experiments, so experiments have been run for the values 1, 10, 20, 30, 40, 50, 100, 150, 200, 250, 300, 350 and 400 to reveal the tendency. There reasons for why 400 is the last value for the experiment, is because of the 32 kB limitation on SSH described in Section 6.3.

The result of the experiments are shown in Figure 6.6. As can easily be seen, the tendency follows our expectations very well.

6.4 Scalability

With minimal overhead, UPPAAL PARMOS should be able to take full advantage of the speed-up gained by parallel processing of verifications. The goal of the tests in this Section, is to test how well UPPAAL PARMOS scales under different workload conditions, when extra resources are added.

The experiments were again exhaustive searches of 1000 verifications, using the brute-fore algorithm, the Sister nodes on Fyrkat and the TGS system with the task specification in listing 6.2. A fixed value of 10 is used for *verifications per process*.

For all test sets, an initial sequential search, with *cores* and *batches* both set to 1, was performed to have a base for measuring speed-up when processing the rest in parallel. The remaining experiments in every set, was then made with the value of *cores* set to 4, and the value for *batches* being from 1 to 5 to adjust parallelism further.

Between experiments, the loop inducing work has been changed between 50, 250 and 500

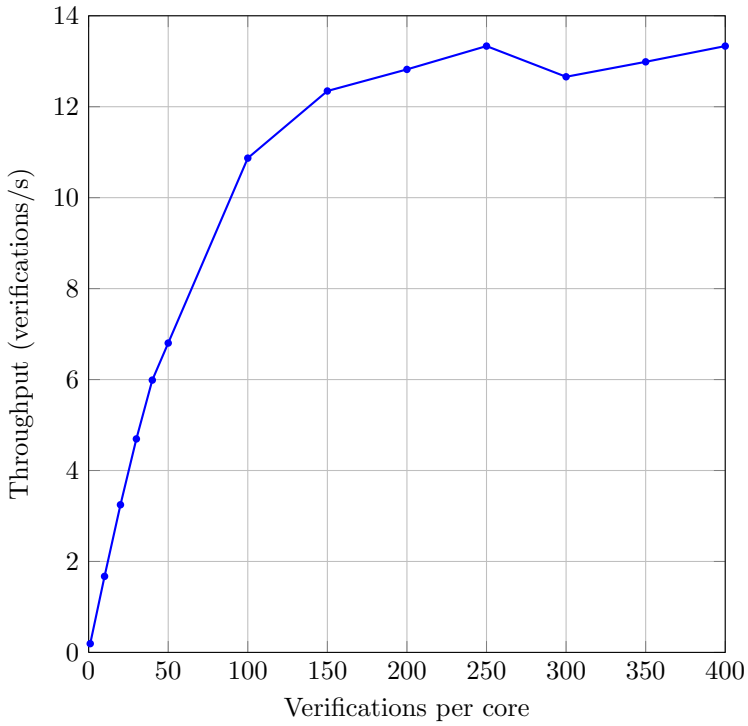


Figure 6.6: Results from experiments with *verifications per core*

verifications, and further the retriever loop has been varied between the fixed, the tuned and the untuned, as introduced earlier. Thus it is possible to compare results between different work loads, as well as the retriever loops. Note that the same tuning as presented in Section 6.3.2 was employed for all tests with this retriever loop.

Prior to testing, our expectations was that the speed-up would be cose to linear. There is no dependencies between two verifications then processing them in parallel, so we expected overhead to be the only obstacle. If the amount of resources should reach a point where overhead from starting batches becomes a bottleneck, this would present a degrade in performance. This is illustrated in Figure 6.7 with the red line. The grey line shows linearity for reference.

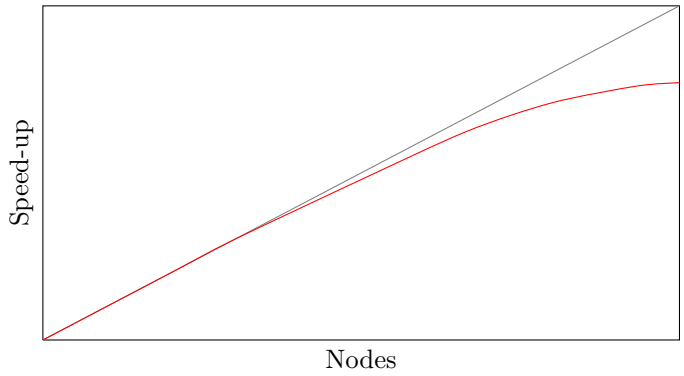


Figure 6.7: Graph illustrating our expectations for speed-up when scaling.

The results reveal that UPPAAL PARMOS scales quite well, in conjunction with the SLURM

resource proxy. Figure 6.8 shows that nearly linear speed-up is achieved, when adding resources, except when using the tuned retriever loop.

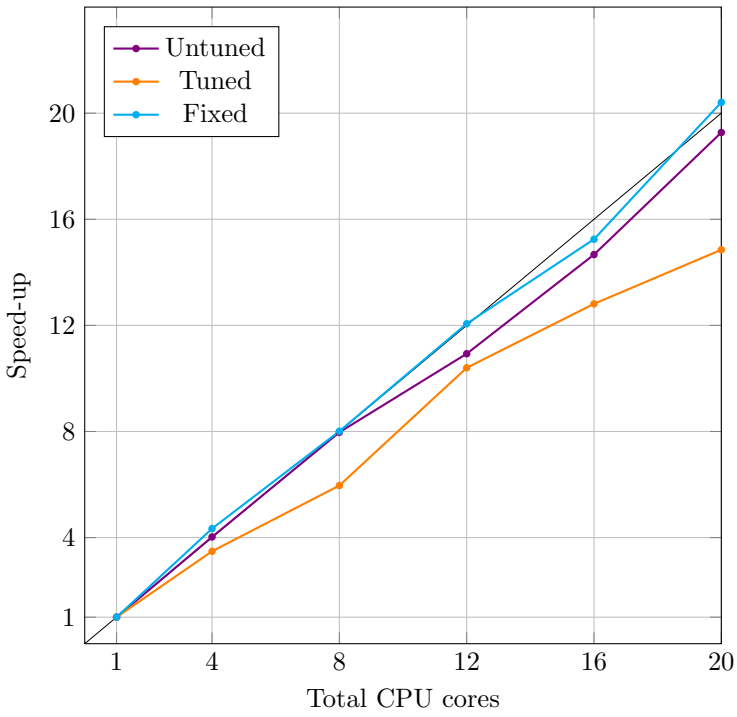


Figure 6.8: Average speed-up for the three retriever loops over all three work amounts

Figure 6.9 shows the average throughput over all three workloads, and illustrates a general tendency, that the fixed retriever loop generally performs better in these tests, than the two others. However as discussed earlier, the fixed retriever loop performs a lot more check than the two others. The results presented in Section 6.3.2, should also be taken into consideration, that the tuned retriever loop performs very well under the circumstances it is tuned for, but the effect of this is averaged away here.

Figure 6.9 however shows that the Tuned retriever loop performs very well around the area it is tuned for. With 50 times processing for every verification, it performs as good as the fixed retriever loop for lower amounts of cores.

6.5 Algorithms

The implemented algorithms, described in Section 5.4, have been tested for efficiency, by measuring how many jobs they need to process before exiting in the “belief” that they have reached the global maximum.

The TGS model is used for this purpose, but with another task specification than for the overhead and scalability tests. The one in Listing 5.1.3 on page 68 is used, in order to have rank-able results for the algorithm to compare on.

In order to know the value the optima found by the algorithms, we have first run an exhaustive search over the parameter space.

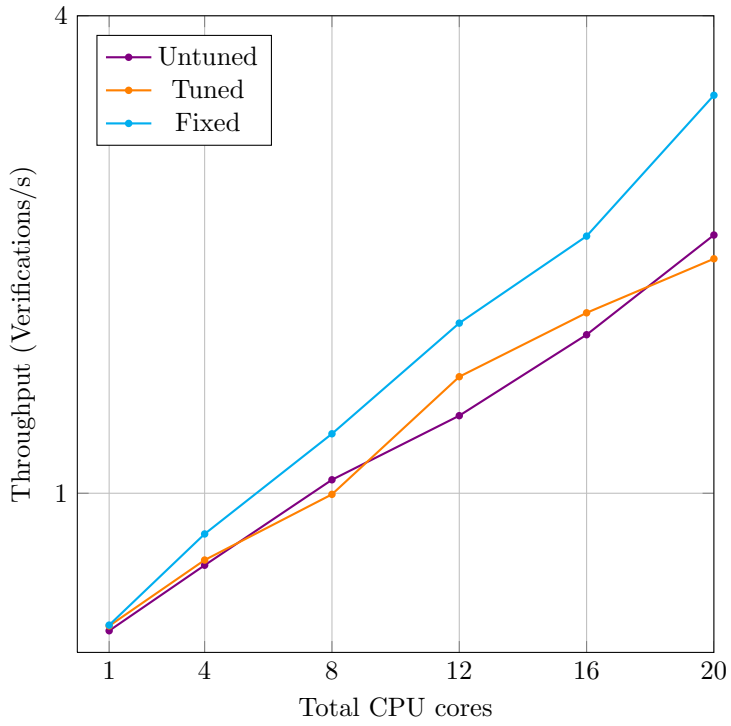


Figure 6.9: Average throughput for the three retriever loops over all three work amounts

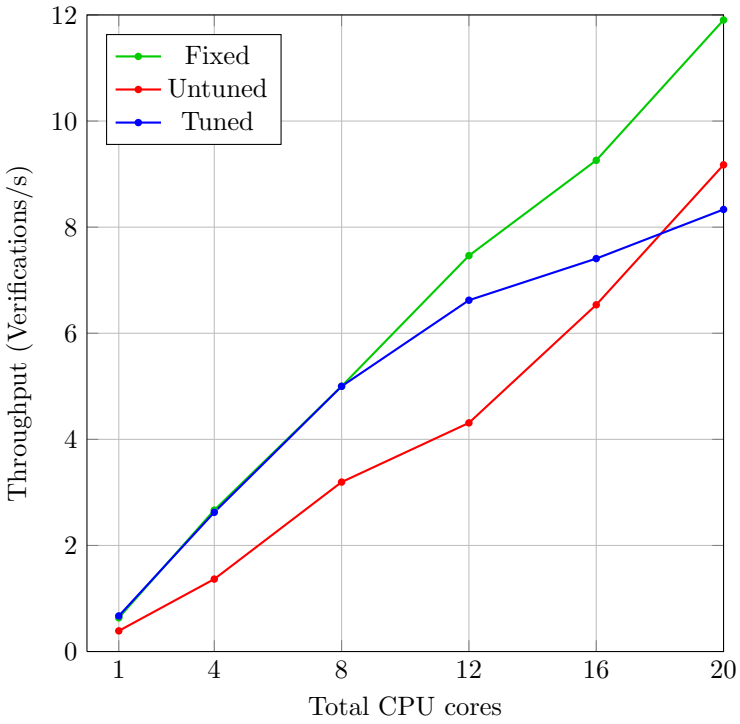


Figure 6.10: Throughput for the three retriever loops with 50 times processing of every

Speed is not a factor, when we look at in the algorithm tests, so in that respect the settings of the resource proxy are without importance. They are however influential on the number of jobs processed by the algorithms, since they set the number of verifications sent to

the resource for verification before any results return, as well as the potential number of verifications still at the resource when the algorithm meets its stopping criterion.

To compare the results of each algorithm, an exhaustive search has been conducted in order to find the global optimum for the model.

A good starting location will, for all algorithms, except brute force, make it end earlier Since the starting location is random for all algorithms, each experiment has been therefore been run a number of times, to reduce the impact of this.

We must regrettably conclude that there is a bug in our implementation of the Hill-Climbing algorithm. It searches the parameter space well and finds the optimal solution in all three experiments, but unfortunately it does not know when to stop. If not stopped, it will eventually have performed an exhaustive search before halting, as was the case with our experiments. We anticipate the bug to be found in the algorithms use of a threshold set.

SA and PAES both worked pretty well, finding the optimal solutions. Results are shown in Table 6.4 Unfortunately we only managed to run one successful experiment with SA, as the Fyrkat cluster was shut down.

Experiment	Verifications	Rankings	Found global optimum
PAES 1	2691	13	Yes
PAES 2	5665	36	Yes
SA	921	3	Yes
Hill-Climbing* 1	2552	34	Yes
Hill-Climbing* 2	389	17	Yes
Hill-Climbing* 3	335	13	Yes

Table 6.4: Results for algorithm tests. * Hill-Climbing verifications are when the algorithm *should* have stopped, had it worked properly

6.6 Performance

To get an idea of the throughput capability of UPPAAL PARMOS, we employed the Brute-Force algorithm on the EKC system described in [19], and tweaked the settings of UPPAAL PARMOS to achieve a maximum throughput. 5 Sister and 14 Killing nodes of Fyrkat was used. We achieved a maximum throughput of 14.45 verification/s on a total of 132 cores. The average processing time of a batch was 650 seconds. With 400 verifications per batch, this gives us 1.625 seconds per verification.

7 Conclusion

As the content of thesis shows, we have successfully designed and implemented a PSA that distributes a parameter sweep of verifications of a UPPAAL model to multiple resources, while employing an optimization scheme to prioritize individual model configurations, thus allowing for faster access to desired results.

This conclusion grounds in the fact that we were able to implement the design of Chapter 3, and sequentially conduct experiments that produced successful results. One of the most important design features, was the decision to make the PSA expandable, with components containing some of the most essential functionality. This decision, which result in a more customizable PSA, is likely to extend the life cycle of UPPAAL PARMOS, since users are able to adjust the two most important aspects in UPPAAL PARMOS, the Algorithm, which decides what to run, and the Resource proxy, which decides where to run it.

In order to measure the performance of UPPAAL PARMOS, multiple experiments have been performed. These experiments were performed on two different cluster set-up, that totally provided 132 processors for us to use. In these experiments we have measured throughput, speed-up and efficiency for three different workloads and a varied number of allocated processors.

7.1 Results

We have conducted experiments of the implemented optimization algorithms. Their results, compared to the one produced by an exhaustive search algorithm, clearly shows that the required amount of verifications needed can be lowered tremendously.

The result from the experiments showed, in almost all cases with different workload, a very close to linear tendency. The limited resources available to us, during testing, is however not sufficient to push UPPAAL PARMOS to the limit, and illustrates that further experiments with more resources available are needed, in order to locate conditions where UPPAAL PARMOS is no longer scalable, in order to identify the bottlenecks which needs to be worked on in future versions.

The decision to have resource proxies between UPPAAL PARMOS and the computing resource has proven usable, yet it have also bred the conclusion that the scalability of UPPAAL PARMOS is truly decided by how the implementation of the communication between a resource proxy and a remote computing resource is done.

The decision to design a feedback loop, for handling the dynamic behaviour of retrieving verification results from resources, has proven valid yet not perfect. We conducted experiments, which showed that a fixed retrieval time of 1 sec. instead of a varied, was in close to all cases superior to the dynamic one. An attempt to tune the the varied time by adding a fixed constant produced a successful result, yet only in and close to the measure that was

tuned against. However, were additional resources available to us, the sheer number of connections made by the fixed retriever loop might have proved to be a bottleneck, thus requireing the use of a feedback loop.

7.2 Future work

Although we with UPPAAL PARMOS have created a working system, there is still much to attend to.

While we for this version of UPPAAL PARMOS have decided not to focus on user right, in terms of authentication and authorization, this area is essential and should be attended to.

In this version of UPPAAL PARMOS we allow only constant global integers to be parametrized. This a limitation in the system, which should be extended to include struct and array types as parameters. We have made some considerations regarding a realisation of this, yet have found most models can be adjusted, such that a global integer parameter is enough. However, we believe that in future versions of UPPAAL PARMOS this should be available to the user.

Another place that should be attended is the retrieval system of the PSA. While we believe that the feedback system is a profound idea, the current set-up with one single retriever for all resources is not profound and should be attended to. The feedback timer should also be tuned towards more than one result.

The resources available to the PSA are currently totally independent from a task. This decision was made to allow the resources to be shared between user, yet we have made considerations about have users specify their "own" resources to be used only with their task. This is however in the area of user rights which we do no attend to in the version of UPPAAL PARMOS.

Regarding the use of available resources, currently the PSA views all resources as being equal, thus utilizing one single resource fully before moving on to the next. In future versions of UPPAAL PARMOS we suggest that a more fair work-balanced scheme is utilized for.

While we have conducted multiple experiments for measuring speed-up and throughput we have not done any profiling on our internal PSA code. This should also be done in the future, such that all possible computing power can be extracted.

A bottleneck in the current system is the requirement to generate a verification identifier in the database. This identifier is used throughout of the system and is necessary to distinguish between the different verifications. We have made some considerations about generating an identifier from a hash-value of the model configuration, to be used as identifier instead of the database identifier.

8 Appendix

This appendix contains class diagrams that we have found it necessary to remove from the content of this thesis.

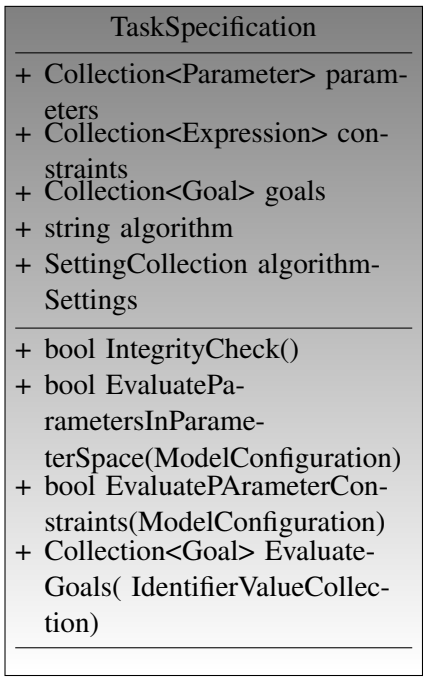


Figure 8.1

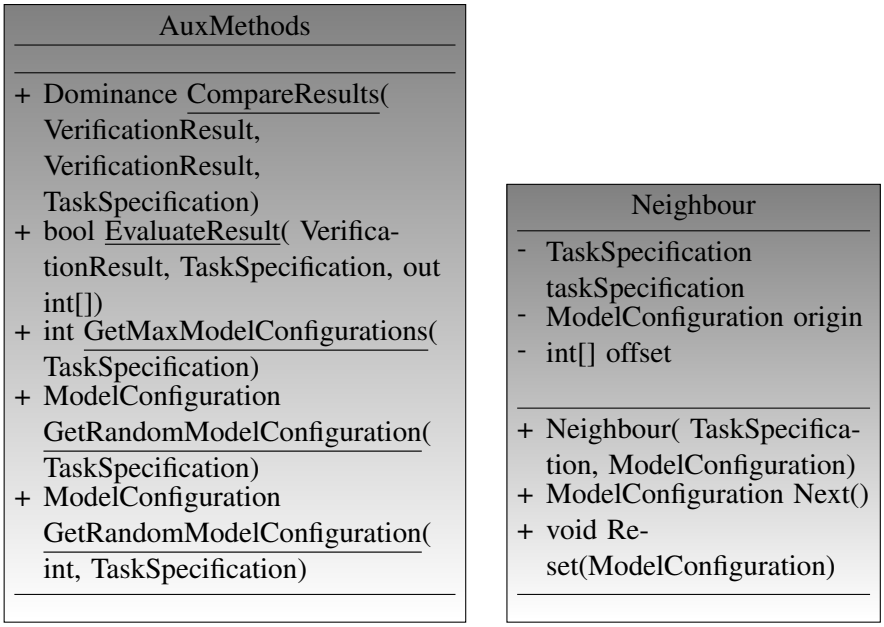

AuxMethods
+ Dominance CompareResults(VerificationResult, VerificationResult, TaskSpecification)
+ bool EvaluateResult(VerificationResult, TaskSpecification, out int[])
+ int GetMaxModelConfigurations(TaskSpecification)
+ ModelConfiguration GetRandomModelConfiguration(TaskSpecification)
+ ModelConfiguration GetRandomModelConfiguration(int, TaskSpecification)
Neighbour
- TaskSpecification taskSpecification
- ModelConfiguration origin
- int[] offset
+ Neighbour(TaskSpecification, ModelConfiguration)
+ ModelConfiguration Next()
+ void Reset(ModelConfiguration)

Figure 8.2

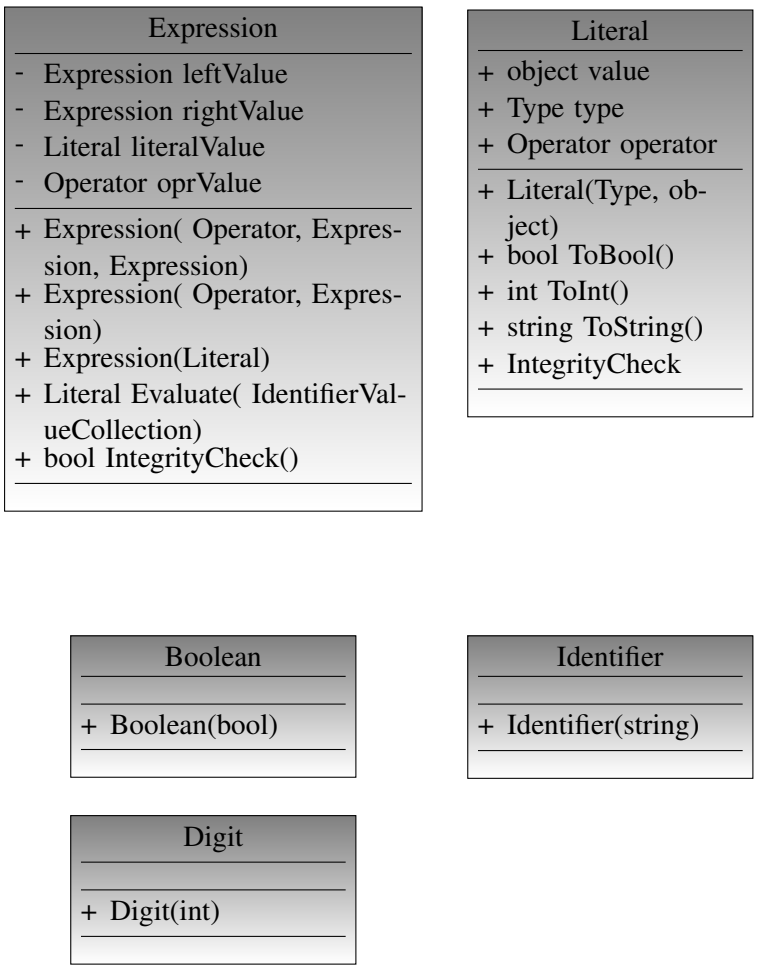


Figure 8.3

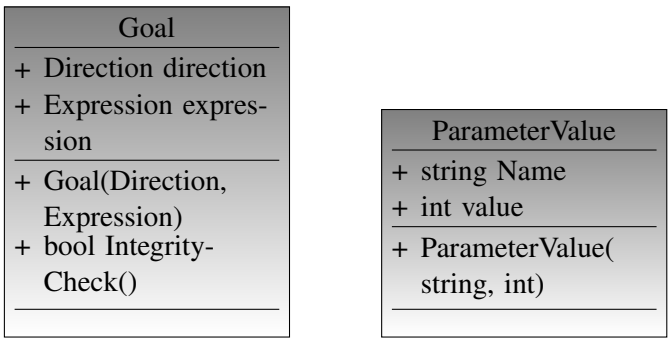


Figure 8.4

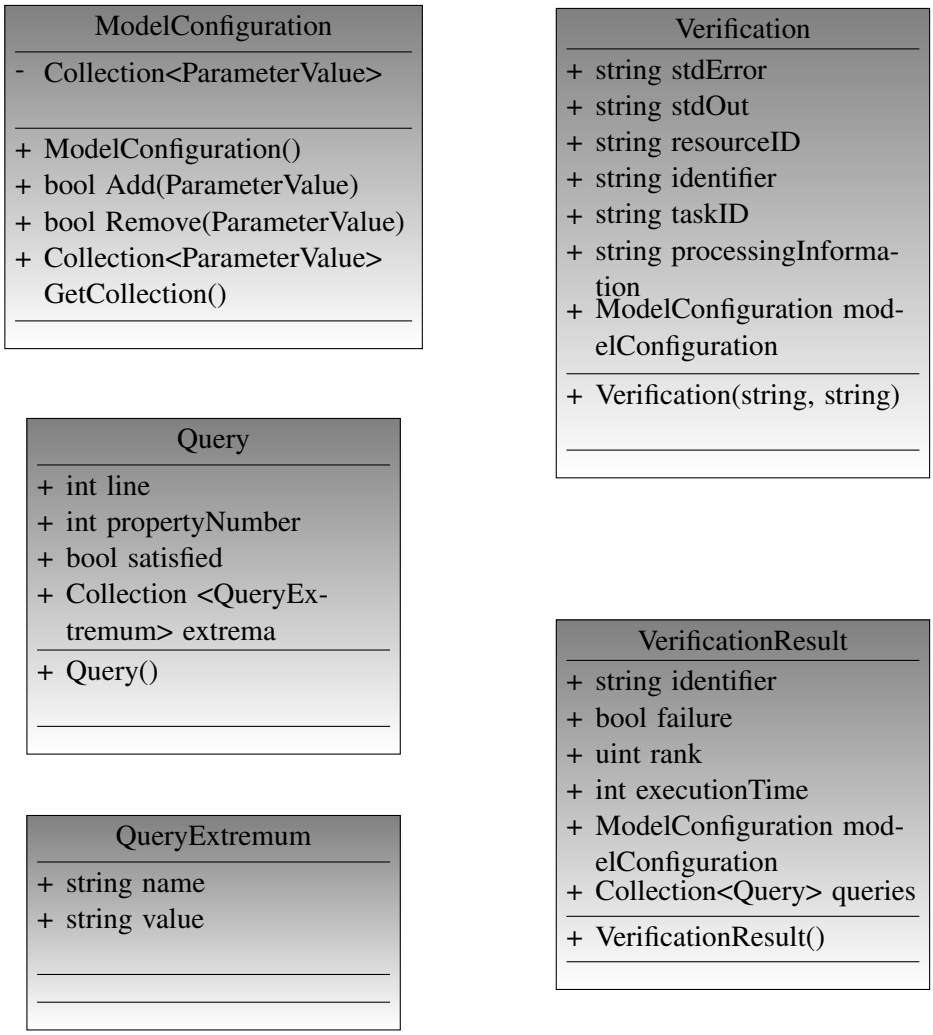

ModelConfiguration
- Collection<ParameterValue>
+ ModelConfiguration()
+ bool Add(ParameterValue)
+ bool Remove(ParameterValue)
+ Collection<ParameterValue> GetCollection()
Verification
+ string stdError
+ string stdOut
+ string resourceID
+ string identifier
+ string taskID
+ string processingInformation
+ ModelConfiguration modelConfiguration
+ Verification(string, string)
Query
+ int line
+ int propertyNumber
+ bool satisfied
+ Collection <QueryExtremum> extrema
+ Query()
VerificationResult
+ string identifier
+ bool failure
+ uint rank
+ int executionTime
+ ModelConfiguration modelConfiguration
+ Collection<Query> queries
+ VerificationResult()
QueryExtremum
+ string name
+ string value

Figure 8.5

Acronyms

ANTLR	ANother Tool for Language Recognition	79
API	Application Programming Interface	61
APST	AppLes Parameter Sweep Template	9
CLI	Common Language Infrastructure	57
DNS	Domain Name System	76
ERM	Entity-Relationship Model	27
FIFO	First-In First-Out	12
GA	Genetic Algorithm	38
GUI	Graphical User Interface	13
IP	Internet Protocol	76
NTA	Network of Timed Automata	3
OGE	Oracle Grid Engine	11
PAES	Pareto Archived Evolution Strategy	38
PSA	Parameter Sweep Application	1
RDBMS	Relational Database Management System	69
SA	Simulated Annealing	45
SFTP	SSH File Transfer Protocol	79
SLURM	Simple Linux Utility for Resource Management	11
SOA	Service-Oriented Architecture	16
SOAP	Simple Object Access Protocol	70
SSH	Secure Shell	76
TA	Timed Automaton	3
TGS	Task Graph Scheduler	4
TORQUE	Terascale Open-Source Resource and QUEue Manager	11
WSDL	Web Services Description Language	70

Bibliography

[1] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, pages 1–6, 13–17, 54–55, 57–65, 77–79. John Wiley & Sons, 1989.

[2] AAU Grid. The fyrkat wiki. Wiki, 2011. URL `https://fyrkat.grid.aau.dk/wiki/`. (Login required) Accessed: May 16, 2011.

[3] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Pearson, third edition, 2001.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. A system of patterns: Pattern-oriented software architecture. Wiley, 1996.

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

[6] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *9th IEEE Heterogeneous Computing Workshop (HCW)*, pages 349–363, 2000.

[7] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*, chapter 1,2,15, pages 14–15,19–20,53–57,605. Addison-Wesley Longman, fourth edition, 2005.

[8] C. Darwin and J. Carroll. *On the origin of species*. Broadview Press, 2003. ISBN 1551113376.

[9] C. V. Deutsch and X. H. Wen. An improved perturbation mechanism for simulated annealing simulation. *Mathematical Geology*, 30:801–816, 1998.

[10] I. Foster. Service-oriented science. *Science*, 308(5723):814–817, 2005.

[11] G. F. Franklin, J. D. Powell, and M. L. W. (Author). *Feedback Control of Dynamic Systems*, chapter 1, pages 1–7. Prentice Hall, third edition, 1997.

[12] T. Gal. Sharpssh - a secure shell (ssh) library for .net, 2007. URL `http://www.tamirgal.com/blog/page/SharpSSH.aspx`. Accessed: May 20, 2011.

[13] N. A. B. Gray. Comparison of web services, java-rmi, and corba service implementations. In *The Fifth Australasian Workshop on Software and System Architectures (AWSA 2004)*, page 52, 2004.

[14] J. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor, 1975.

[15] ISO. *ISO/IEC 23271:2006: Information technology — Common Language Infrastructure*. International Organization for Standardization, Geneva, Switzerland, 2010. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=42927.

[16] S. Kirkpatrick, C. G. Jr., and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[17] J. Knowles and D. Corne. The pareto archived evolution strategy: A new baseline algorithm for pareto multiobjective optimisation. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 1. IEEE, 1999. ISBN 0780355369.

[18] J. Knowles, R. Watson, and D. Corne. Reducing local optima in single-objective problems by multi-objectivization. In *Evolutionary Multi-Criterion Optimization*, pages 269–283. Springer, 2001.

[19] J. M. Krag, P. S. Freiberg, and B. Villumsen. Distributed parameter sweep for uppaal models. Technical report, Computer Science, Aalborg University, 2010.

[20] Lawrence Livermore National Laboratory. Slurm documentation, August 2010. URL https://computing.llnl.gov/linux/slurm/documentation.html. Accessed: May 27, 2011.

[21] K. G. L. Luca Aceto, Anna Ingólfsdóttir and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[22] S. Luke. *Essentials of Metaheuristics*. Lulu, 2009. URL http://cs.gmu.edu/~sean/book/metaheuristics/. Accessed: March 5, 2011.

[23] L. Mathiassen, A. Munk-Madsen, P. Nielsen, and J. Stage. *Objektorienteret analyse & design*. Marko, 2001.

[24] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21: 1087–1092, 1953.

[25] Microsoft. Activator class, 2010. URL http://msdn.microsoft.com/en-us/library/system.activator.aspx. Accessed May 11, 2011.

[26] Microsoft. System.reflection namespace, 2010. URL http://msdn.microsoft.com/en-us/library/136wx94f.aspx. Accessed May 11, 2011.

[27] Microsoft. Load and unload assemblies, 2010. URL http://msdn.microsoft.com/en-us/library/ms173101.aspx. Accessed May 11, 2011.

[28] Microsoft. Run partially trusted code in a sandbox, 2010. URL http://msdn.microsoft.com/en-us/library/bb763046.aspx. Accessed 17.05.2011.

[29] Microsoft. Ado.net, 2011. URL http://msdn.microsoft.com/en-us/library/aa286484.aspx. Accessed May 21, 2011.

[30] N. Milanovic and M. Malek. Current solutions for web service composition. *Internet Computing, IEEE*, 8(6):51–59, 2004.

[31] MySQL. Download connector/net, 2011. URL http://dev.mysql.com/doc/refman/5.5/en/introduction.html. Accessed: May 21, 2011.

[32] MySQL. Mysql 5.5 manual: 1. general information, 2011. URL http://dev.mysql.com/doc/refman/5.5/en/introduction.html. Accessed: May 10, 2011.

[33] Popular Science Publishing Company. Mechanics & Handicraft, July 1939. URL http://blog.modernmechanix.com/issue/?magname=PopularScience&magdate=7-1939. Accessed: May 5, 2011.

[34] I. Rechenberg. *Evolutionsstrategie–Optimierung technisher Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, Technical University of Berlin, 1973.

[35] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*, chapter 1, 2, 6, pages 1–5, 9,37–45, 235–236. McGraw Hill, fifth edition, 2006.

[36] UPPAAL Team. UPPAAL 4.0 : Small tutorial, November 2009. URL http://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf. Accessed: December 10, 2010.

[37] UPPAAL Team. UPPAAL help file, September 2010. UPPAAL v. 4.1.3 (rev. 4577).

[38] The ANTLR developers. Antlr, 2011. URL http://www.antlr.org/. Accessed: May 21, 2011.

[39] The SSH.NET developers. SSH.NET Library, 2011. URL http://sshnet.codeplex.com/. Accessed: May 21, 2011.

[40] W3C. Web services description language (wsdl) 1.1, March 2001. URL http://www.w3.org/TR/wsdl. Accessed 18.05.2011.

[41] W3C. Soap version 1.2 part 1: Messaging framework (second edition), 2010. URL http://www.w3.org/TR/2007/REC-soap12-part1-20070427/. Accessed: December 15, 2010.

[42] C. Youn and T. Kaiser. Management of a parameter sweep for scientific applications on cluster environments. *Concurrency and Computation: Practice & Experience*, 22 (18):2381–2400, 2010.