A capacity planning tool for PEPA

Christopher D Williams

Master of Informatics

School of Informatics
University of Edinburgh

2014

Abstract

Capacity planning is the process of determining the fewest number of resources required in a system whilst guaranteeing some performance target. This is significant when planning a system as it reduces waste and minimises cost. Performance Evaluation Process Algebra (PEPA) is an algebraic language which is used to build, and analyse the performance of, models of systems. This paper presents the capacity planning tool, an extension of the PEPA Eclipse plugin project, which uses meta-heuristics to find an optimal number of resources, or optimal model configuration, that meets some user defined performance requirements.

Acknowledgements

I would like to express my appreciation and gratitude to Dr Jane Hillston, whose mentoring and guidance have been paramount to the success of this dissertation.

Table of Contents

Chapter 1

Introduction

1.1 Capacity planning

We model so that we can evaluate and determine a system's behaviour without needing to resort to expensive and possibly infeasible experimentation. Using modelling we can simulate a system and inexpensively analyse characteristics of that system under different parameters.

A modeller may be interested in finding an *optimal model configuration*; that is they may be interested in searching for the best configuration of parameters of a system, where both the configuration and the system need to satisfy some requirements.

Capacity planning is an example of this kind of search. An optimal configuration in capacity planning is one that strikes a good[1] balance between system performance and use of resources. In this paper capacity planning has two resource parameters:

- The mixture of resources. Each component in a model serves a purpose, and will perform certain actions, at a given rate, to fulfil its purpose. Rates of actions are one kind of model parameter. We may be able to change the rate of actions by using different components in our model, faster rates over all actions should mean a faster service. However having faster rates usually involves some cost, as it could mean buying more expensive equipment, requiring to pay for more cooling, needing more floor space, etc. Further it may be wasteful to purchase a faster component because if there is not enough demand, or if there is a bottleneck in the system, the component could be idle for long periods of time. An optimal configuration will have the best mixture of components, where the best mixture is a trade off between some notion of cost and some user defined performance requirement.
- The number of resources used. The number of resources used in the model is another example of a model parameter. Ideally the model will have enough resources for the expected load. An optimal configuration will have the fewest

[1]The balance is contingent on the user defined requirements, therefore how 'good' a balance is depends on what the modeller is looking for.

number of resources required, *whilst still satisfying some user defined performance requirement.*

1.2 An example of capacity planning

As an example imagine a company that sells tickets on-line where they have three types of server networked in their infrastructure; front-end web servers, back-end application servers, and database servers (Fig 1.1).

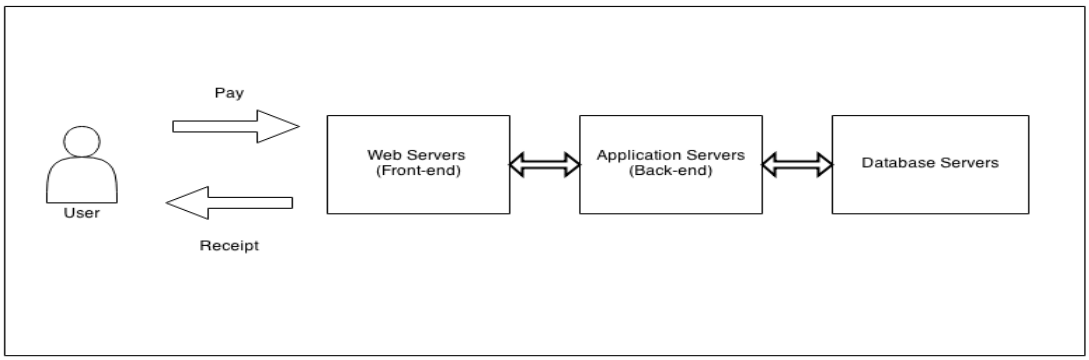


Figure 1.1: A web service infrastructure, user requests are processed by a chain of servers.

Our company has an expectation of the number of clients it wants to serve, and a *performance measure* that is important to it. For example this company will require the *response time* of returning the receipt of a ticket to a user to be within some amount of time under the *expected load*. In this case our imaginary company cares most about customer satisfaction, and therefore the timeliness of their service. Thus our company has a web service infrastructure as a model, *response time* as a performance requirement of the system, and various servers as resources.

Figure 1.2: A PEPA model of the example web service

1. Rates:

$payRate = 0.08$
$receiveRate = 5.0$
$confirmRate = 3.0$
$authoriseRate = 5.0$
$readRate = 10.0$
$updateRate = 5.0$
$writeRate = 5.0$

2. Agents/Components:

$User_0 \stackrel{def}{=} (pay, payRate).User_1$
$User_1 \stackrel{def}{=} (receive, receiveRate).User_0$

$WebServer_0 \stackrel{def}{=} (pay, payRate).WebServer_1$
$WebServer_1 \stackrel{def}{=} (confirm, confirmRate).WebServer_2$
$WebServer_2 \stackrel{def}{=} (authorise, authoriseRate).WebServer_3$
$WebServer_3 \stackrel{def}{=} (receive, receiveRate).WebServer_0$

$ApplicationServer_0 \stackrel{def}{=} (confirm, confirmRate).ApplicationServer_1$
$ApplicationServer_1 \stackrel{def}{=} (read, readRate).ApplicationServer_2$
$ApplicationServer_2 \stackrel{def}{=} (update, updateRate).ApplicationServer_3$
$ApplicationServer_3 \stackrel{def}{=} (authorise, authoriseRate).ApplicationServer_4$
$ApplicationServer_4 \stackrel{def}{=} (write, writeRate).ApplicationServer_0$

$DatabaseServer_0 \stackrel{def}{=} (read, readRate).DatabaseServer_0$
$+(write, writeRate).DatabaseServer_0$

3. System Equation:

$User_0[500.0] \underset{S_1}{\bowtie} WebServer_0[1.0]$
$\underset{S_2}{\bowtie} ApplicationServer_0[1.0] \underset{S_3}{\bowtie} DatabaseServer_0[1.0]$

4. Co-operation language sets:

$S_1 = \{pay, receive\}$
$S_2 = \{confirm, authorise\}$
$S_3 = \{read, write\}$

Performance Evaluation Process Algebra (PEPA) is an example of a modelling language and is supported by the PEPA Eclipse Plug-in project [9]. Figure 1.2 is an example of the ticketing web service in Figure 1.1 modelled in PEPA and is going to be used as a demonstration for the rest of this chapter.

The Rates section is a declaration of the rates of actions; we see in the example the `pay` action for a user is relatively slower than the rate of any other action. If a single user were using the system then we could believe that the system will be under utilised, the system could respond faster than the user can pay. This would lead to components being idle and thus would be wasteful. There must be an upper limit on the number of users that can use this system before saturation happens, where a component becomes a bottleneck and we see performance degradation. At this point the components will be at their highest utilisation for that model configuration.

The Agents/Components section is a mapping of the company's four components (Users, Web servers, Application servers, and Database servers) into PEPA agents. A user of the web service is abstracted as the two agents $User_0$ and $User_1$, as they are either making a payment or receiving a receipt. The web server component has four agents $WebServer_0$ to $WebServer3$, two of which are used for interacting with user agents, and a further two for interfacing with the application servers. The application servers are abstracted into 5 agents; $ApplicationServer_0$ to $ApplicationServer_4$, two are for interacting with the web servers, two are for interacting with database servers, and finally a fifth state $ApplicationServer_2$ where the application server is assumed to be updating bank details. The database servers have one agent $DatabaseServer_0$, where they are either reading or writing database entries.

The System Equation shows the number of components being modelled (the number of resources in the configuration), and the actions the components interface with each other on. The number of components, or *population*, is seen in the square brackets next to the component name in the system equation: in Figure 1.2 there are 500 user components $User_0[500.0]$ and one of each of the other components $WebServer_0[1.0]$, $ApplicationServer[1.0]$, and $DatabaseServer[1.0]$. The actions the components interface on are labelled as sets S_1, S_2, and S_3.

The Co-operation language sets are the actions the components interface or co-operate on; for example in this case S_1 shows that a $User$ and a $WebServer$ co-operate on the actions `pay` and `receive`. These actions can only happen in co-operation, so a $User_0$ must wait until there is a $WebServer$ ready to do a `pay` action before it can move into its next state $User_1$.

Now we have a model of the infrastructure in a modelling language we can start investigating what effect different populations have on performance. Using the simplest case we are going to say that the company has these requirements:

1. The *response time* of the system must be on or below a required value, in this case we are interested in the time it takes for $User_0$ to return to $User_1$.
2. The least number of servers are used.

1.3 Doing a manual search

We can treat the system equation in this model as a vector in some 4 dimensional space:

$$User_0[x_1] \underset{s_1}{\bowtie} WebServer_0[x_2] \underset{s_2}{\bowtie} ApplicationServer_0[x_3] \underset{s_3}{\bowtie} DatabaseServer_0[x_4]$$
where $x_i \in \mathbb{R}_{>0}$

This allows us to consider the search problem. In order to find the optimal configuration for the simplest case the modeller would need to use the following algorithm:

1. Manually change the population of some component(s) $x_i \in (x_1, x_2, x_3, x_4)$ in the system equation using the PEPA Eclipse Plug-in editor

2. Run the relevant PEPA Eclipse Plug-in performance evaluation tool to return the new performance value

3. Check if the performance value and population are satisfactory if not: Go back to 1. Repeat until the lowest population satisfying the performance requirement is found

Table 1.1: How a human might search for an optimum configuration

A person could probably make an intuitive guess as to what populations they should start with, but it is clearly time consuming to manually search through different combinations of components. This gets worse as a system equation's dimensions become larger.

There is likely to be dependence between the components, we may see very little performance increases when raising the population of some components and large increases in performance when raising another.

Using our example system we can fix $x_4 = 10$, $x_1 = 500$ and randomly sample 1000 points in $[x_2, x_3]$ (where $x_2, x_3 \in [1, 1000]$). Then we can evaluate the performance or *fitness* of each sample point. Figures 1.3, 1.4, 1.5 are three dimensional graphs of the results. The higher the fitness (the z-axis) the closer the model configuration is to our performance measure. In both Figure 1.4 and 1.5 we see a steep increase in fitness as x_2 or x_3 increases respectively, however the fitness soon plateaus and we no longer see any fitness increase for any larger populations whilst x_4 is fixed. Until the number of database servers is increased we will not see any further increase in performance.

Thus there must be a 'sweet spot' in the plateau where we have a minimal number of web and application servers but a maximum performance value for this configuration. This kind of dependence is probably easy to spot in lower dimensionality system equations, but much harder to visualise in larger models.

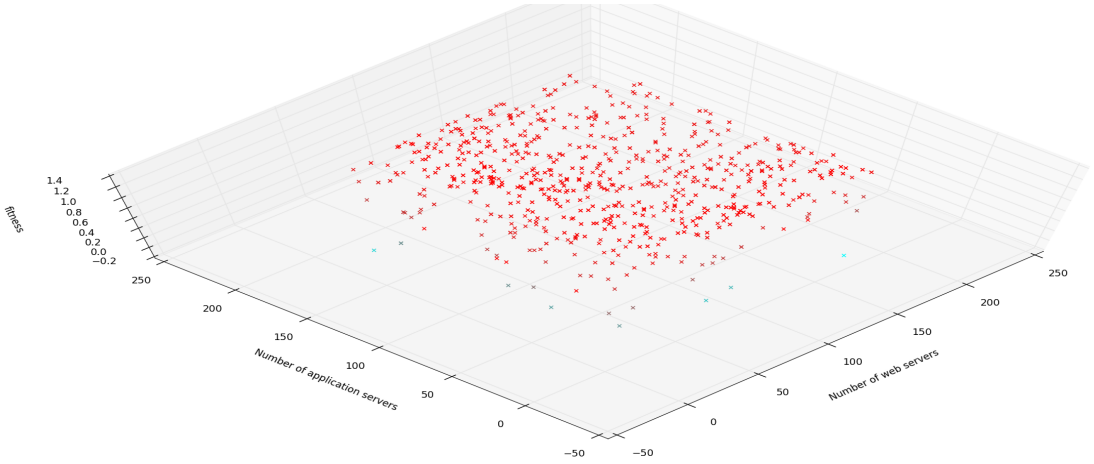


Figure 1.3: This graph shows a top view of the search space and candidate system equations sampled from 1000 points.

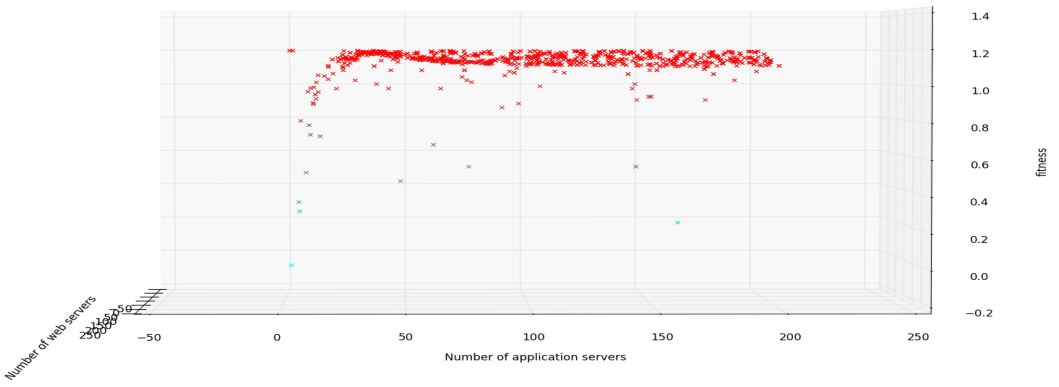


Figure 1.4: This graph shows the search space from the side of the application servers. This clearly shows a rise from the left into a solid plateau at the top.

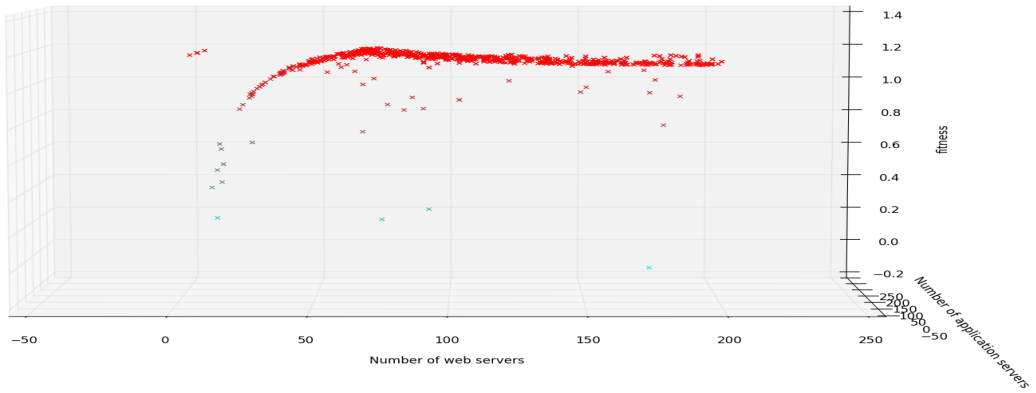


Figure 1.5: This graph shows the search space from the side of the web servers. This clearly shows a rise from the left into a solid plateau at the top, there is a small rise at around 75 web servers where we see a possible 'sweet spot'.

What if we expand our imaginary company's requirements so they care slightly more about performance than population?

1. The *response time* of the system must be on or below a required value, in this case we are interested in the time it takes for $User_0$ to return to $User_1$.
2. The smallest number of servers are used.
3. Minimising the distance from the performance target is twice as important as expenditure on components.

Our modeller is going to need to run their algorithm over many model configurations so that they can compare performance gains over different populations.

There is some further tuning the company is interested in. Our imaginary company does not care as much about spending on database servers as they do application servers, so what if our company cares more about the cost of one component and not another? The modeller would now need to search for an optimal configuration with looser restrictions on the population of one or more components. This is a subtle extension to the search problem, the modeller would need to run their algorithm many more times so that they have a set of data with which they can best determine the optimal population ratio.

What if our company has a choice of servers it can use? This is when we start to see resource mixing as described at the start of this chapter. What if there are two types of Application server available, one having a superior processor which means it can perform its duties faster than the other. Now our requirements are:

1. The *response time* of the system must be on or below a required value, in this case we are interested in the time it takes for $User_0$ to return to $User_1$.
2. The smallest number of servers are used.
3. Minimising the distance from the performance target is twice as important as

expenditure on components.

4. Database servers are half the cost of Application servers.
5. There are three different types of Application server.

In these cases the modeller would need to create models for *every posssible combination* of Application server, examine many system configuration combinations, and then compare the distance from the performance target against population. This is time consuming and does not scale well as the number of models required grows exponentially with the number of component types to test:

$$modelsRequired = (numberOfComponentType)^2 - 1$$

It is clear then that manual capacity planning in PEPA is time consuming, but also perhaps error-prone as creating a new model or manually comparing results will involve a large set of data.

1.4 My contribution

My contribution is a capacity planning tool; an extension to the PEPA Eclipse plugin project that allows for automated capacity planning of PEPA models.

I have tested Hill Climbing (HC), Genetic Algorithms (GA), and Particle Swarm Optimisation (PSO) meta heuristics as capacity planning optimisation algorithms and after evaluation found that a PSO is the best meta heuristic of the three for capacity planning. This kind of capacity planning is called a *single search*, as only one meta heuristic is used to find an optimal model configuration.

Further in order to minimise the user interface and to provide a more accurate search mechanism I have investigated using one meta heuristic to drive the parameters of a second meta heuristic. This is called a *driven search*, and after evaluating three different combinations of meta heuristics (an HC driving an HC, an HC driving a GA and an HC driving a PSO) I have found that an HC driving a PSO is the best driven search.

A driven search however takes considerably longer than a single search, so I have concluded that it is best to offer both types of search to the modeller, and for the driven search to not only output the top model configurations, but to also output the meta heuristic parameters so that the modeller can use this information to perform single searches in the future.

The capacity planning plugin extension:

- Provides an Eclipse wizard for users to enter search parameters.
- Offers ten optimum model configurations as output in an Eclipse view pane.
- Allows for a search on one of two user defined performance measures: average response time and throughput.

- Finds an optimal configuration that has a minimal population whilst having a performance measure that is close to the user defined measure.
- Allows the user to a define weighting between population and performance so that they may optionally decide which is more important to them.
- Allows the user to weight each component in the system equation so that they may assign a notion of cost to each component.
- Allows the user to enter components with 0 population, and therefore allow searching on mixes of components in one editor.

1.5 Previous work

In MPP1 I had a capacity planning extension that could search on models using either a Hill Climbing (HC) algorithm or a Genetic Algorithm (GA). The fitness function was not as complex as the one offered in MPP2; it did not offer fine tuning on the component weighting, it could not handle having a population of zero for any component (the importance of this is discussed in section 4.4). MPP1 has a user interface, so there also existed a Wizard and the supporting WizardPages for the user to run a search, this interface has been refactored and improved in MPP2. Lastly MPP1 did not offer chaining as a possible type of search, this is something only seen in MPP2.

1.6 Structure of document

The following chapters are structured as follows:

- Chapter 2 provides some background to the third meta heuristic used in this tool, and some context on how I have used existing PEPA code to implement my project. Lastly this chapter discusses and gives a critical evaluation of related work.
- Chapter 3 gives an overview of the capacity planning tool. This chapter is necessary as it gives some context and perspective for the remaining chapters. This chapter discusses the final architecture, and leaves the discussion of work not found in the final capacity planning tool to later chapters.
- Chapter 4 discusses how I developed the tool. If the previous chapter gives an impression of what the tool looks like, then this chapter is a discussion of the work I did to complete it, and why I decided to implement it in this way.
- Chapter 5 is the evaluation and analysis of the three meta heuristics and chaining technique. It provides evidence supporting my decisions about the final version of the capacity planning tool.
- Chapter 6 is the evaluation of the user interface developed to use the capacity planning tool in PEPA. In this chapter I discuss how I evaluated the user

interface, and provide some critical evaluation on how the interface could be improved.

- Chapter 7 is the conclusion of this document, where I provide some critical evaluation of the project, a discussion of future work, and my evaluation of the success of the project.

Chapter 2

Background

The objective of the first part of this chapter is to introduce the Particle Swarm Optimisation (PSO), the third algorithm evaluated as a possible candidate search algorithm for this project. The purpose of the second part of this chapter is to provide some context to the techniques I have used in order to develop the capacity planning tool; the ODE function which is used to evaluate a model, and the different stages a model goes through in order to be used by an ODE function. The third part of this chapter is a discussion of related work, and a critical evaluation of related work.

2.1 Particle Swarm Optimisation (PSO)

Particle Swarm Optimisation (PSO) is a stochastic optimisation algorithm modelled after flocking or swarming agents [8]. A PSO works by scattering a number of candidates (or particles) in some search space and providing them a random velocity. Each generation, or iteration of the optimisation method, each candidate moves depending on its velocity. Then the best candidate of that iteration, called the *global best*, is found and its position is given to all other candidates. Each agent then uses a portion of this global best, a portion of its own velocity, and also a portion of its own best, *local best*, to create a new velocity vector which it uses on the next step. After a number of iterations the PSO should converge on an optimum position in the search space. This is better described in pseudocode inspired by Luke [6] as follows:

There are a number of parameters for a PSO;

- Iterations - `generation`: The number of iterations the algorithm will run for before stopping.
- The candidate population - `candidatePopulation`: This defines how many candidates will be used at once in the search space.
- Local best position - `localBest`: This defines what portion of the candidates known best position should be used in the new velocity.

- Global best position - `globalBest`: This defines what portion of the best known candidates position should be used in the new velocity.
- Original velocity - `originalVelocity`: This defines how much of the original velocity should be seen in the new velocity.

Initiation of variables

```
generation = user defined value
candidatePopulation = user defined value
localBest = user defined value
globalBest = user defined value
originalVelocity = user defined value
```

Scattering of candidates

```
bestCandidate = null
arrayOfCandidates = []

for candidatePopulation do:
    newCandidate = (new candidate) //candidate random position and velocity
    arrayOfCandidates = arrayOfCandidates ∪ newCandidate
```

Table 2.1: PSO initiation

A Particle Swarm Optimisation was implemented in MPP2 as an extension of the selection of meta heuristics implemented in MPP1. The implementation of the PSO for MPP2 is seen in Chapter 4.

Iterative search

```
for generation do:

    //find fittest
    for each candidate in arrayOfCandidates do:
       fitnessFunction(candidate) // use system equation fitness function to assess fitness
       if(candidate 'is better than' bestCandidate) do:
          bestCandidate = candidate

    //update each candidate
    for each candidate in arrayOfCandidates do:
       currentPosition ← candidate's position // the system equation
       previousVelocity ← candidate's velocity
       localBestPosition ← candidate's previous best position
       globalBestPosition ← bestCandidate's best position
       newVelocity = (originalVelocity * previousVelocity) +
          localBest * (localBestPosition - currentPosition) +
          globalBest * (globalBestPosition - currentPosition)
       candidate's velocity ← newVelocity  // the candidate gets a new velocity vector

    //move each candidate
    for each candidate in arrayOfCandidates do:
       candidate's current position ←
          floor(candidate's current position + candidate's velocity)
```

Table 2.2: PSO search

2.2 PEPA

The purpose of this section is introduce the reason behind using Ordinary Differential Equations (ODEs) as performance evaluation tools and to provide some context on how a PEPA model is evaluated. This will be necessary when in later sections I describe using Abstract Syntax Tree (AST) handling and ODE evaluation.

2.2.1 ODE

Originally Performance Evaluation Process Algebra (PEPA) had been designed to take advantage of Continuous-Time Markov Chains (CTMCs). However CTMCs can suffer from state-space explosion; the larger the population of agents in the model, the larger the state space required in order to solve the steady state probability. This means that PEPA needs an increasing amount of processing power and memory in order to evaluate larger model populations when using CTMCs.

Ordinary Differential Equations (ODEs) are a new solution technique in model evaluation, they provide faster analysis than CTMCs [3]. ODEs can take seconds to analyse

a model with agent populations in the hundreds or thousands. Using the ODE technique to evaluate models makes it feasible to use a search algorithm to find an optimum model configuration.

2.2.2 Models

In order to evaluate a PEPA Model using ODEs the PEPA Eclipse plug-in must first convert the String model class (PepaModel) into a Graph model class (Graph). Once the model is a graph, the ODEs can evaluate, returning performance measure results as an array of floats. Fig 2.1 shows the PEPA Eclipse plug-in methods used by the capacity tool, and the life cycle of a model.

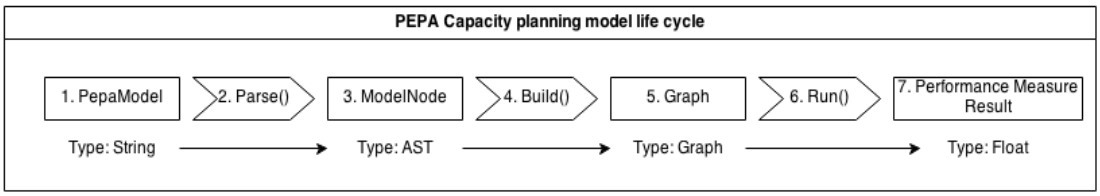


Figure 2.1: The model life cycle in the capacity planning tool. The square boxes represent classes of models, the arrowed boxes represent a conversion method.

1. PepaModel (a Java class); this is the collection of strings entered into the Model Editor by the modeller.
2. The PepaModel is parsed into an AST. A model can only be parsed if the model checking and parsing were successful, thus we always have a valid model beyond this stage.
3. Abstract Syntax Tree (AST); the model is now represented by an AST, which is a class called ModelNode.
4. The AST is built into a Graph. The modeller will have chosen some ODE parameters during the previous Wizard stage of the capacity planning tool, these parameters are used here to build the graph.
5. The model is now represented as an ODE graph.
6. Performance evaluation runs.
7. Performance measure results are returned as an array of floats.

In the life cycle of a model in Figure 2.1 the capacity planning tool manipulates model configurations during stage 3. Using a Visitor pattern and a Java class called ASTHandling, the capacity planning tool can operate on models, and change the population value of some component. This updated AST is then built into a Graph (5) and evaluated using ODEs.

2.2.3 Identical interfaces

In Chapter 4 I introduce the idea of *mixed agent type models*, these are contingent on the understanding of what an *identical interface* is. An agent's interface is the total set of actions it exposes, or co-operates on, to use with other agents. If we take the example system from the introduction (Figure 2.2)) we can say that the ApplicationServer's interface is the set of actions `confirm`, `authorise`, `read`, and `write`.

$$User_0[X_1] \underset{S_1}{\bowtie} WebServer_0[x_2] \underset{S_2}{\bowtie} ApplicationServer_0[x_3] \underset{S_3}{\bowtie} DatabaseServer_0[x_4]$$

where

$x_1, x_2, x_3, x_4 \in \mathbb{R}_{>0}$

and

$S_1 = \{pay, receive\}$

$S_2 = \{confirm, authorise\}$

$S_3 = \{read, write\}$

Figure 2.2: System equation example

In order for an agent to have an identical interface, it must expose the same set of actions. Let us assume that there are two more agents added into the model, shown in Figure 2.3.

$$\begin{aligned}
ApplicationServer2_0 &\overset{def}{=} (confirm, confirmRate).ApplicationServer2_1 \\
ApplicationServer2_1 &\overset{def}{=} (read, readRate).ApplicationServer2_2 \\
ApplicationServer2_2 &\overset{def}{=} (update, updateRate2).ApplicationServer2_3 \\
ApplicationServer2_3 &\overset{def}{=} (authorise, authoriseRate).ApplicationServer2_4 \\
ApplicationServer2_4 &\overset{def}{=} (write, writeRate).ApplicationServer2_0 \\
\\
ApplicationServer3_0 &\overset{def}{=} (confirm, confirmRate).ApplicationServer3_1 \\
ApplicationServer3_1 &\overset{def}{=} (read, readRate).ApplicationServer3_2 \\
ApplicationServer3_2 &\overset{def}{=} (update, updateRate2).ApplicationServer3_3 \\
ApplicationServer3_3 &\overset{def}{=} (authorise, authoriseRate).ApplicationServer3_4 \\
ApplicationServer3_4 &\overset{def}{=} (write2, writeRate).ApplicationServer3_0
\end{aligned}$$

Figure 2.3: Two extra application servers are added to the example system model.

ApplicationServer2 has a different rate for its `update` action, however it has an identical interface to ApplicationServer. On the other hand ApplicationServer3 has identical rates as ApplicationServer, but it has the action `write2`, which means it can not offer the same interface as ApplicationServer, and therefore is not identical.

$$User_0[X_1] \underset{S_1}{\bowtie} WebServer_0[x_2] \underset{S_2}{\bowtie} (ApplicationServer_0[x_3] || ApplicationServer2_0[x_4]) \underset{S_3}{\bowtie} DatabaseServer_0[x_5]$$
where
$x_1, x_2, x_3, x_4, x_5 \in \mathbb{R}_{>0}$
and
$S_1 = \{pay, receive\}$
$S_2 = \{confirm, authorise\}$
$S_3 = \{read, write\}$

Figure 2.4: System equation example, with identical interface agents.

If we have two components that have identical interfaces, and identical rates, we could argue that if two identical components are in parallel, the parallel unit is equivalent to one agent having the sum of the population of the identical agents (Figure 2.5).

$$ApplicationServer_0[x_1] \equiv (ApplicationServer_0[x_2] || ApplicationServer2_0[x_3]) \equiv ApplicationServer2_0[x_4]$$
where
$x_1, x_2, x_3, x_4 \in \mathbb{R}_{>0}$
and
$x_1 = x_4 = x_2 + x_3$

Figure 2.5: Identical interface agents, with the same rates, are equal in parallel to their single counter part.

2.3 Related work

Using ODEs to make a feasible capacity planning tool in PEPA is first mentioned in the work by Hillston, Tribastone and Gilmore, 2012 [3]. The paper draws on an example case called the e-University model, and has what I believe to be a manual capacity planning experiment. The capacity planning experiment fixes the population of one component, and seeks to find an adequate average response time, whilst reducing the population of all other components. I could not replicate the experiment and return the same average response time of 3.686 as found in the paper. This I believe will be due to how the ODE has been set up in the experiment. However using the same model values, and using my own ODE settings, I found e-University model with the population of $N_P = 70$, $N_D = 20$, $N_L = 20$, $N_{PS} = 40$, $N_{PD} = 40$ (with a total population of 190) gave an average response time of 1.4. Using my capacity planning tool, I have managed to find a model configuration of $N_P = 9$, $N_D = 25$, $N_L = 62$, $N_{PS} = 14$, $N_{PD} = 28$ (a total population of 171) which gives an average response time of 0.48, this means the capacity planning tool has managed to find a model configuration which is both faster, *and* uses less components overall.

Meta heuristics, specifically Genetic Algorithms (GA) [4] [1], have been used with Process Algebra in at least two pieces of work. Marco et al [7], are using GAs in a

similar manner to this dissertation; they use an existing PEPA model and search for a set of parameters in order to find an optimal model configuration. The key difference is the parameters being searched on; Marco et al. are looking for a *set of rates* to fit a definition of an optimal model, whereas in this paper we are looking for a *component population* that fits some definition of optimal. Karaman et al. [5] conversely are using GAs to build a process algebra model; using GAs they create models which can satisfy some path optimisation. They have chosen process algebra because it can express a broad class of complex problems.

Chapter 3

Architecture Design

3.1 Overview

This chapter aims to describe the architecture of the capacity planning tool and to discuss all the shared mechanisms in the extension, such as the two different fitness functions which can be used by any meta heuristic. Eclipse plugins are written in Java, therefore this extension is also written in Java. Some of the following will be described as abstractions of Java classes, that is some of the following will be made of many literal Java classes, but will be discussed here as if they are a single class in order to make discussion less complicated.

The extension is the sum of three parts, a Wizard, a Job and a Viewer. Fig 3.1 is a high level work flow diagram of how these parts interact:

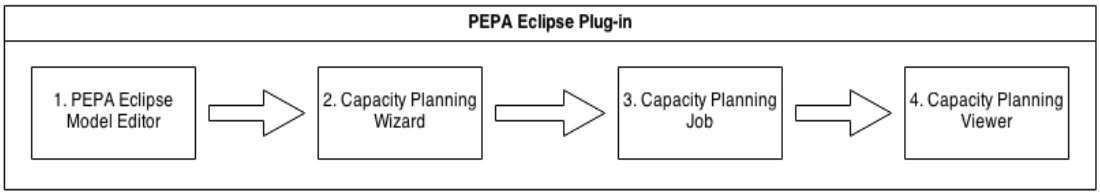


Figure 3.1: A high level picture of the capacity planning tool. The input is a PEPA model written in the PEPA Eclipse plug-in, the output is displayed on a Viewer inside Eclipse.

1. (input) : The modeller would create a model in the PEPA Eclipse model editor.
2. (settings/input) : The capacity planning Wizard is selected by the modeller which guides the modeller into entering the search parameters, e.g. the range of populations, the performance requirement, the meta-heuristic parameters, etc.
3. (processing) : The capacity planning Job, which is the search engine, runs once the modeller has completed the Wizard.
4. (output) : Finally the results are displayed in a PEPA Eclipse Viewer.

3.1.1 Capacity planning Wizard

The user follows an Eclipse wizard in order to set up and start a capacity planning job. A Wizard is a Java class and is used for controlling a logical ordering of more Java classes called WizardPages. WizardPages are used to guide the user in entering the parameters, the input and settings, required for a capacity planning search.

Figure 3.2 below shows the logical steps of the capacity planning wizard pages. More details on the User Interface are in Chapter 6 where we also see an evaluation of the interface.

1. The user creates a model in the PEPA editor.
2. The user starts the capacity planning tool by selecting the action in the PEPA menu, the wizard picks up the PEPA model from the editor.
3. The user sets the type of search, driven or single (explained in the following section), and the kind of evaluation they would like to perform: either response time or throughput.
4. The path splits into either a driven or single search depending on which algorithm was selected on the previous page. Here the user can change the number of experiments (explained in the following section) and algorithm parameters.
5. This page is for the selection of the performance targets, either actions or agents, and the parameters for the ODE function.
6. This page determines the balance between population vs performance in the fitness function.
7. This page is for the user to determine the performance target values, and the weighting of the performance targets.
8. This page is for the user to determine the possible population ranges, and the weighting for each component.
9. The data is handed over to the capacity planning job, which starts the search.

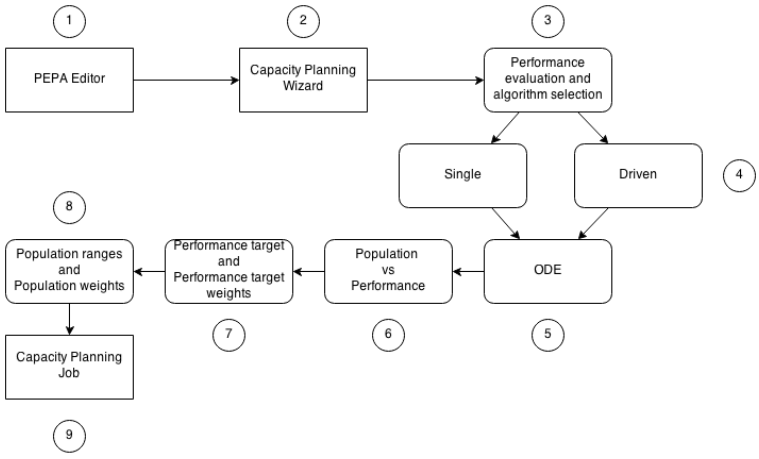


Figure 3.2: A high level picture of the capacity planning wizard.

3.1.2 Capacity planning Job

3.1.2.1 General design

The capacity planning Wizard passes all data to the capacity planning Job, and then starts the search. A capacity planning job is created as a separate thread and so runs separately from Eclipse.

Figure 3.3 is an abstraction of a capacity planning job; a capacity planning job creates a Lab, a Lab runs a number of *experiments*. As the meta heuristics are stochastic the extension has been designed to increase the likelihood of finding a good result; a number of searches can run serially and the top results from all experiments are returned as output. Each run is called an experiment, an experiment is simply a new meta heuristic with a randomly seeded *candidate* population.

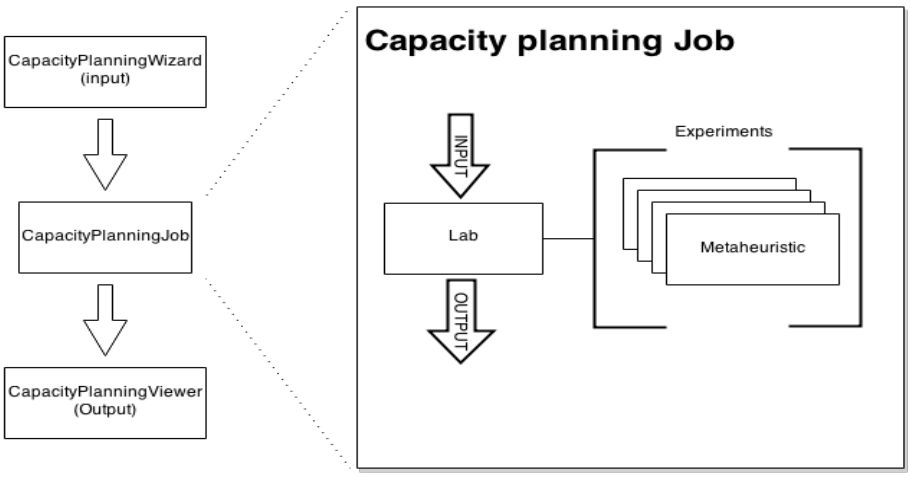


Figure 3.3: High level diagram of a capacity planning job

A meta heuristic has its optimisation method, and a number of *candidates* called the *population* (Figure 3.4). A candidate is a possible solution to the optimisation or search. Each candidate has its own *FitnessFunction* class, the type of FitnessFunction depends on the type of search: there can be two types of candidate and therefore there are two types of FitnessFunction.

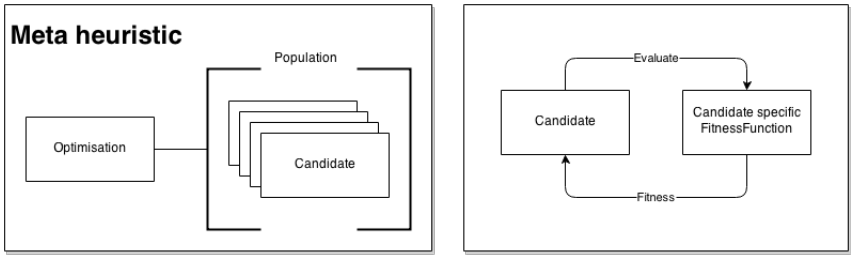


Figure 3.4: High level diagram of a capacity planning job

The architecture of the capacity planning job has been designed so that a meta heuristic can run on any type of candidate, the type of candidate used depends on whether a

single or driven search configuration is being used. The reasons for using different types of searches is seen in Chapter 4.

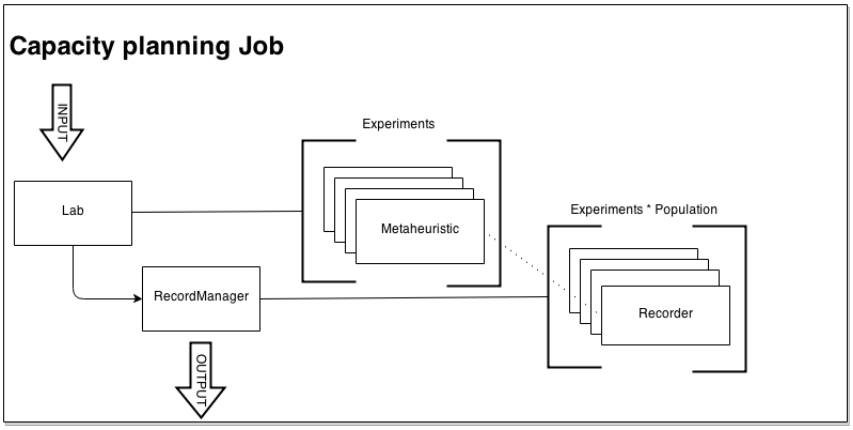


Figure 3.5: Recording output

In order to keep track of all the experiments and candidates we use a RecorderManager. A RecorderManager is created with every Lab, and for every candidate there exists a Recorder (therefore there are $|population| * |experiment|$). It is the Recorder's duty to track the progress of an experiment, and it is the RecorderManager's task to collect all Recorders and pass the results to the capacity planning Viewer.

3.1.2.2 Single searches

Single searches are when only one meta heuristic is used in the search process. In the final capacity planning version only one single meta heuristic option is given, the PSO, and therefore the name 'single' is dropped.

In MPP1 there was a choice of Hill Climbing (HC) or Genetic Algorithms (GA), MPP2 introduces Particle Swarm Optimisation (PSO) as a third choice. The implementation and evaluation for these algorithms, and the evidence supporting the final choice of meta heuristic is seen in Chapters 4 and 5.

The candidate in a single search is a model configuration, specifically an abstraction of the system equation from a PEPA model. This type of candidate uses the system equation fitness function for evaluation (described in a later section).

Figure 3.6 shows the layout of a single search. As before a Job creates a Lab, and a Lab can have a number of meta heuristics as experiments. Figure 3.7 is an expansion of the system equation candidate and fitness function work flow:

1. The optimisation method calls a request on the system equation candidate, which is passed to the system equation fitness function.

2. The system equation fitness function assesses the population of the model configuration, and makes a call to the AST handling.

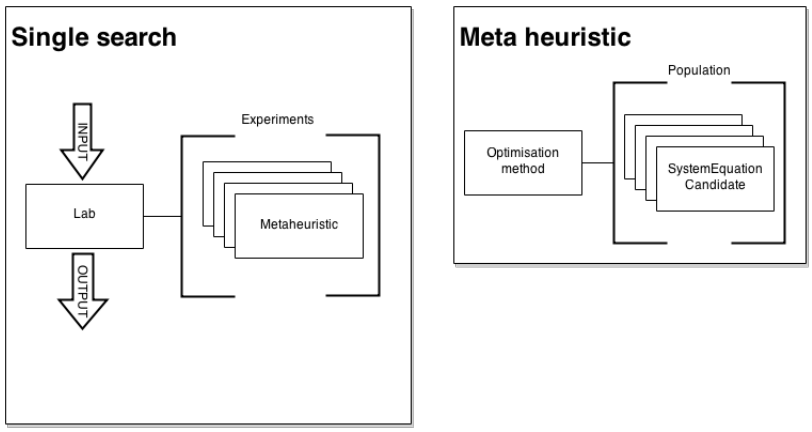


Figure 3.6: High level diagram of a single search job

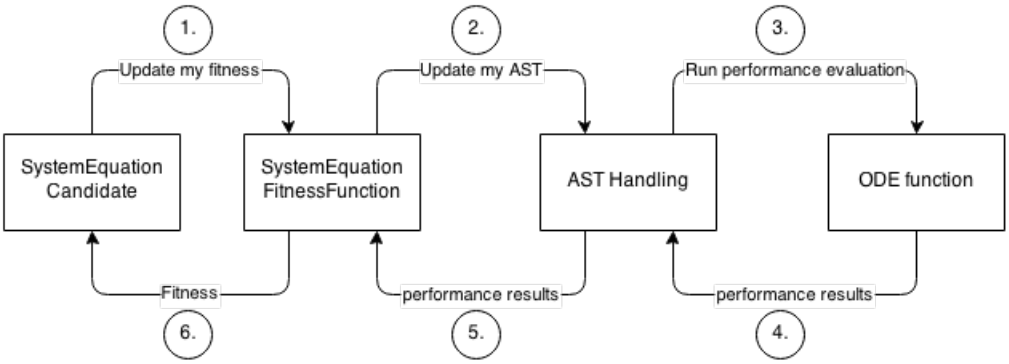


Figure 3.7: More detail behind the system equation candidate

3. The AST Handling then updates the underlying PEPA AST, and creates an ODE graph from the updated AST. This graph is sent to the ODE function.

4. The ODE function evaluates the graph with respect to the user defined performance target and evaluation choice (throughput or average response time). The evaluation results are passed back up to the AST Handling.

5. The AST Handling passes the results back up to the system equation fitness function.

6. The fitness function now uses the performance results, and model configuration population to create a fitness measure, and passes the result to the candidate.

This architecture is different from MPP1 because now the AST handling and ODE function calls are 'underneath' the fitness function. This design is superior to the one used in MPP1 because it allows the possibility of having the performance evaluation happen in a separate thread and therefore be run in parallel.

3.1.2.3 Driven searches

Driven searches are a type of chaining (discussed in detail in Section 4.5), and are an extension of single searches. A driven search is where a *driving* meta heuristic is used to find the best settings for a second *driven* meta heuristic.

In the final capacity planning tool a driven search is specifically where an HC algorithm finds the best settings for a PSO algorithm, the justification for this choice of configuration is discussed in Chapter 5.

The candidate for a driven search is a Single search Lab (above) and is called a Lab candidate. The fitness function for a driven search is a Lab fitness function and is discussed in a later section.

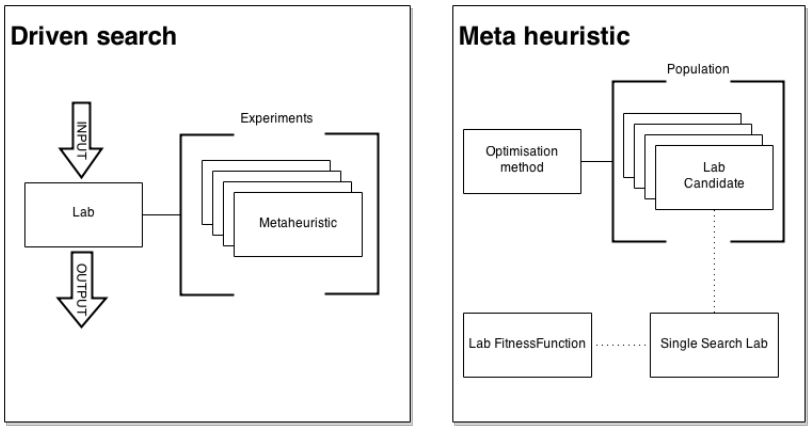


Figure 3.8: High level diagram of a driven search job

3.1.2.4 System Equation Fitness Function

In this subsection we discuss how a model configuration is evaluated during a single search. In the Introduction we saw an example of what a modeller would be searching for; an optimal model configuration, with the optimisation having specific requirements. Here I expand and define the search concisely, a search for an optimal model configuration involves:

- **Minimising performance target distance:** The modeller will have a desired performance target (of either throughput or average response time). The modeller is looking for a model configuration where the distance between its performance measure and the performance target will be as small as possible.
- **Minimising component population:** The modeller will want to reduce the number of resources used in a model, therefore they are looking for a model configuration with the lowest component population whilst still satisfying the above.
- **Balancing performance targets and component population:** The modeller may consider the distance to the performance target to be more important to them than keeping a minimum component population, or the reverse, they may have

restrictions on the number of components they can use and care less about the distance from the performance target. It is required therefore that there is some ability to be able to balance performance distance against component population.

- **Component weighting:** The modeller may want to reduce, or penalise, the use of one component in a candidate model configuration. This introduces a notion of relative cost between components, there is a requirement to offer some method of assigning importance or cost to components.
- **Performance target weighting:** The modeller may be interested in several performance targets, but they may consider some targets more important than others. There is a requirement here to allow a user to define some notion of importance to performance targets individually.
- **Component choice:** The modeller may have some choice in what components to use, there may be a choice of different types of component which have different costs and rates. There is a requirement to be able to search on models that offer a choice of resource.

Balancing population and performance is a coarse way to allow a modeller to consider cost. We have two values, the distance a model is from a performance target, and the total component population used in a model. If the modeller is only interested in reducing the total component population, and has no interest in the distance from a performance target, then we can say the distance to the performance target will cost the modeller nothing. On the other hand, if the modeller is only interested in the performance target distance, then we can say the component population will cost the modeller nothing. This gives us a notion of *user defined cost*, providing a balance lets the modeller determine their own notion of cost, the cost of distance against the cost of the population.

Performance and component weighting are finer controls of *user defined cost*. Providing performance target weights are similar to balancing the performance and population, when more weight is added to a target we are effectively saying we consider the distance from that performance target more important than another. Or another way of saying it, the distance from that target costs more than another. With component weighting it is more obvious; there maybe different reasons for needing to assign value to a component, but if we weight components relative to each other then we have introduced relative cost using weighting. The effect of weighting is seen later in this subsection.

Therefore optimal model configurations are configurations that minimise performance target distance and component population, whilst satisfying the *user defined cost*.

A system equation fitness function evaluates a candidate model configuration to give a *fitness* value. Let us assume there is an optimal configuration; an impossible configuration which has a total population of zero but also a distance of zero from the performance target. The system equation fitness function evaluates a candidate configuration based the distance from the target and component population, whilst satisfying user defined costs. The lower a fitness value, the better it is, for this impossible configuration the fitness value will be zero, the best possible fitness value. Optimal model

configurations then, are the set of models that have the lowest fitness value.

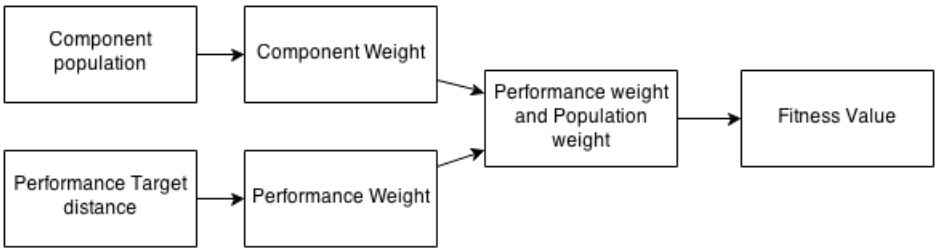


Figure 3.9: System equation fitness function

A system equation fitness function can be broken into five parts directly related to five out of the six points above (with component choice being discussed in a later chapter):

1. The result of the performance evaluations from the ODE function (Performance target distance)
2. The number of components in a system equation (Component population)
3. The weighting of components in terms of population (Component weight)
4. The weighting of performance targets across all selected performance targets (Performance weight)
5. The balance between performance targets and population size (Performance versus Population weight)

These five parts collectively build up a fitness value (Figure 3.9) and it is this value which determines how good one system equation candidate is relative to another. The fitness is better the *lower* it is.

Lets take the equation in Table 3.1 as an example system equation, and look at throughput on the actions `pay` and `receive` as performance targets.

$$User_0[X_1] \underset{S_1}{\bowtie} WebServer_0[x_2] \underset{S_2}{\bowtie} ApplicationServer_0[x_3] \underset{S_3}{\bowtie} DatabaseServer_0[x_4]$$
where
$x_1, x_2, x_3, x_4 \in \mathbb{R}_{>0}$
and
$S_1 = \{pay, receive\}$
$S_2 = \{confirm, authorise\}$
$S_3 = \{read, write\}$

Table 3.1: System equation example

1. The result of the performance evaluations from the ODE function

The performance results, $ODEResult_i$, is returned by the ODE after it has run a performance evaluation. There will be one $ODEResult_i$ for each user defined performance

target, so then in our example case there will be two performance targets as we are looking at throughput on the actions `pay` and `receive`. The user will have given a performance target, $performanceTarget_i$, for each performance measure in the capacity planning Wizard. A $scaledPerformanceValue$ is created so that it can be used later in the fitness function:

$$scaledPerformanceValue_i = |(100 - ((ODEResult_i / performanceTarget_i) * 100)|$$

2. The number of components in a system equation

The modeller will have defined in the Wizard the minimum and maximum population for each component that the extension can search through: $minPopulation_i$ and $maxPopulation_i$.

A range is calculated as: $populationRange_i = maxPopulation_i - minPopulation_i$.

Each system equation candidate then gives the population of each component $componentPopulation_i$. Then a value for population is created, $scaledPopulationValue_i$:

$$scaledPopulationValue_i = |((componentPopulation_i / populationRange_i) * 100)|$$

3. The weighting of components in terms of population

The modeller will also have decided upon weights for each component, $componentWeight_i$, which gives the fitness function some notion of cost per component:

$$totalWeight_i = \sum_0^n (componentWeight_i)$$

$$weightedPopulationResult = \sum_0^n (scaledPopulationValue_i * (componentWeight_i / totalWeight_i))$$

A larger weighting of one component produces a larger fitness value relative to the other components, thus giving some notion of cost at a component level.

4. The weighting of performance targets across all selected performance targets

Similarly the performance targets can also be weighted:

$$totalWeight_i = \sum_0^n (performanceTargetWeight_i)$$

$$weightedPerformanceResult = \sum_0^n (scaledPerformanceValue_i * \frac{performanceTargetWeight_i}{totalWeight_i})$$

This allows the user to put more importance on finding one performance target over any others.

5. The balance between performance targets and population size

Finally the modeller will have entered a population and performance balance $populationWeight$ and $performanceWeight$. These are used when the modeller would like to make either performance or resource count as a more important factor in finding an optimal model configuration.

$$totalWeight = populationweight + performanceWeight$$

$$\begin{aligned} fitnessValue = {} & weightedPopulationResult * (populationWeight/totalWeight) \\ & + performanceWeight * (performanceWeight/totalWeight) \end{aligned}$$

In summary all of the above creates a fitness value where the smaller the value the lower the number of components used, and the closer the performance evaluation will be to the user defined target.

3.1.2.5 Lab Fitness Function

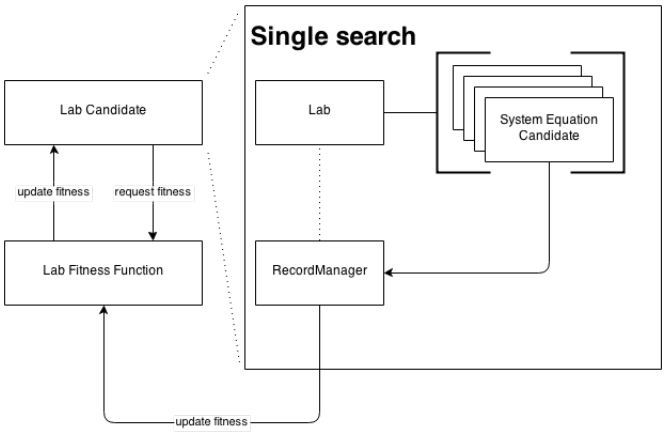


Figure 3.10: Lab candidate calling a fitness update

Recall that a Lab candidate is a single search. A Lab fitness function calls the Record-Manager from the underlying single search (Figure 3.10) to return 4 values;

- the top fitness from the underlying experiments $topFitness$
- the mean fitness of all experiments $meanFitness$
- the standard deviation of the experiments $standardDeviation$
- the average response time of the underlying experiments $averageResponseTime$. How long an experiment took to run, not to be confused with the average response time performance measure of the model.

The lower a lab fitness the better the underlying single search will be. Getting the best fitness has the highest priority and is reflected in the fitness function as having a weight of 0.6, next priority is finding a single search that on average returns a high value and therefore it uses a weight of 0.2, in order to break any ties between candidates accuracy (standard deviation) and response time are used with 0.1 weights:

$$fitnessValue = (topFitness * 0.6) + (meanFitness * 0.2) + (standardDeviation * 0.1) + (averageResponseTime * 0.1) \quad (3.1)$$

Figure 3.11: Lab fitness function

3.1.3 Capacity planning Viewer

The Capacity planning viewer is a viewer pane in Eclipse. It is passed the top ten fittest candidates from either type of search and displays them in order in the view pane. A review and evaluation is seen in chapter 6.

This chapter has provided an overall picture of the capacity planning tool, in the next chapter I discuss how the capacity planning tool was implemented, and the design decisions that dictated the outcome of final version.

Chapter 4

Discussion of MPP2 Implementation

4.1 Introduction

At the end of MPP1 I had an extension which gave the option of using either Hill Climbing (HC) or Genetic Algorithms (GA) to find an optimum model configuration.

The GA, when using the 'default'[1] settings, was found to be no better than the HC Algorithm. In MPP1 I proposed that a Particle Swarm Optimisation (PSO) algorithm may have better success because it has the potential to accidentally 'wander' through good positions as opposed to the GA's more 'jumpy' behaviour due to using the crossover function.

Further I proposed another reason for the GA under performing; the GA may have had a disadvantage in the MPP1 evaluation *because of* the 'default' settings. The GA's settings may have been incorrect or sub optimal for the search space. The GA may have performed better if it has been given better settings, or a smaller search space. This presents a problem: to use the meta heuristics as effectively as possible the modeller will need optimal meta heuristic configuration settings, therefore the modeller needs to understand the meta heuristic and its settings. I proposed in MPP1 the idea of *chaining*; either using *driven* or *pipelined* meta heuristics. Chaining is where we use a second meta heuristic to either narrow down the search space (pipelined), or find the settings for the meta heuristic doing a model configuration search (driven), this will be discussed in a later section.

Separately, the fitness function in MPP1 was quite coarse, and the capacity planning tool could not yet handle *mixed agent type models* (explained in Section 4.4). In order to be able to provide a more flexible tool to the modeller I also had to extend MPP1's fitness function, this is explained in detail in Section 3.1.2.4.

In summary, this chapter discusses the implementation of the PSO, the implementation of chaining and the extension of the MPP1 fitness function. In order to realise these implementations I needed to refactor the existing project, which I discuss in the following section.

[1] The default settings were defined in MPP1

4.2 Refactoring

The program code in MPP1 was inflexible, I decided that in order to implement the PSO and chaining I should first restructure the capacity planning tool's architecture.

4.2.1 Extending MPP1 meta heuristic candidates

The MPP1 project code only had a system equation as a meta heuristic candidate (called a ModelObject in MPP1) which meant the meta heuristics could only operate on one kind of problem. It was necessary to create a super class of type Candidate and to allow the meta heuristics to operate on any type of Candidate. In the refactoring I created subclasses of type Candidate, SystemEquationCandidate and LabCandidate, and then shifted all ModelObject functionality into the SystemEquationCandidate. This gave the architecture a degree of flexibility as now the meta heuristics could operate on any problem that could be sub-classed as a Candidate, the reason for which becomes clear in later sections.

Next I separated out the AST functions from the SystemEquationCandidate object and moved the ODE evaluation methods from a common shared object, as previously the ODE performance evaluation could only happen in serial (Figure 4.1). Now each candidate has access to its own ODE evaluator (Figure 4.2). The intention here was to allow parallel Candidate evaluation in future versions of PEPA when the ODE evaluation function call is made thread safe.

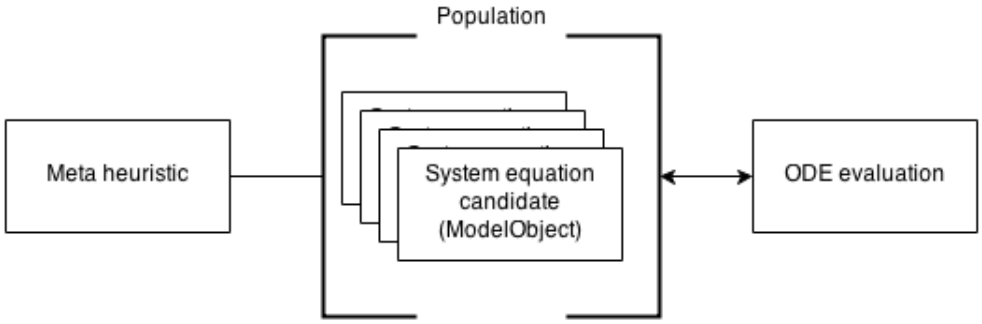


Figure 4.1: In MPP1 there could only be one type of candidate, previously called a ModelObject, there was also only one ODE evaluator for the entire candidate population. This diagram shows the meta heuristic, a single evaluator and a population of ModelObjects.

A super class of type FitnessFunction was also designed, with its subclasses of SystemEquationFitnessFunction and LabCandidateFitnessFunction. Each Candidate has a related FitnessFunction object, it is this object that is called on by a Meta heuristic when evaluating a Candidate. The AST handling separated previously from the SystemEquationCandidate is now found in a new class type ASTHandling, which belongs to the SystemEquationFitnessFunction (4.3). Finally the ODE evaluator belongs the ASTHandling.

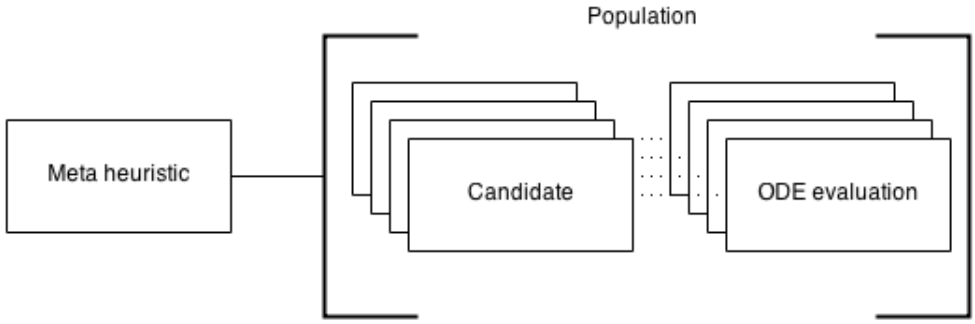


Figure 4.2: In MPP2 after refactoring a meta heuristic could take any Candidate of which a SystemEquationCandidate is a subclass, and any system equation has access to its own ODE evaluator. This diagram shows a meta heuristic with a system equation candidate population each with their own ODE evaluators.

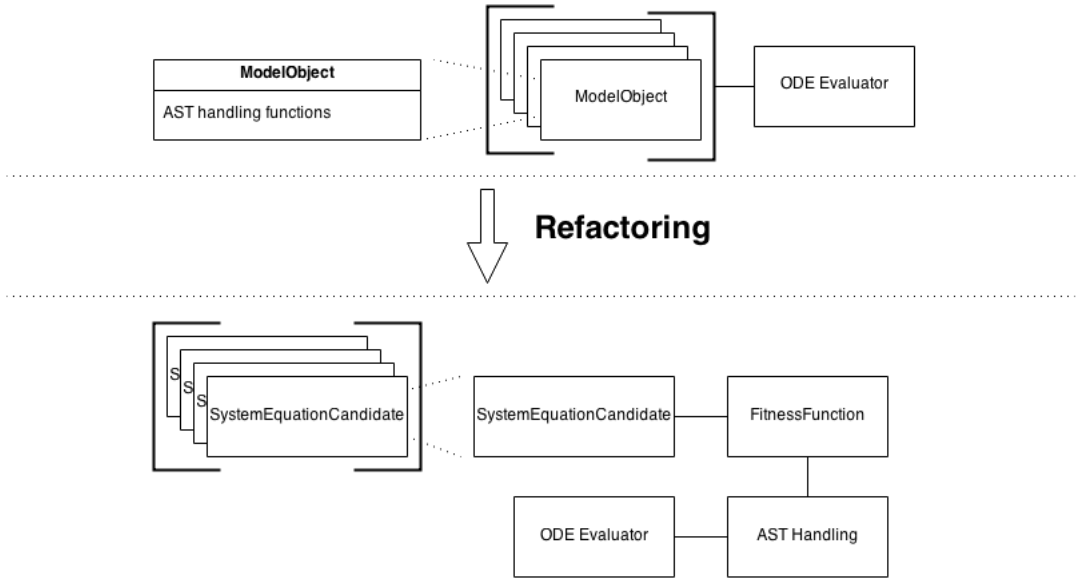


Figure 4.3: Redesigning the ModelObject as a SystemEquationCandidate; the AST handling is now found in a new class, and each SystemEquationCandidate has access to its own ODE Evaluator

4.2.2 Feedback

Learning from the mistakes I made in MPP1, I designed the classes RecordManager and Recorder, and redesigned the framework to allow for experiments (allowing the modeller to run multiple meta heuristics in one search as discussed in Chapter 3). The RecordManager and Recorders are feedback classes, they are used when providing output to the user, but are also used as temporary storage in between experiments. A RecordManager oversees many experiments and temporarily keeps track of all results. The capacity planning Job calls a RecordManager, which collects the results from all experiments, sorts the results in order of fitness, and finally outputs the results for the modeller. It is also the RecordManager that is called by chaining in order to determine

Lab fitness or obtain Candidates.

Once the refactoring was completed the project code was in a more legible form, and made it easier to implement the PSO and any chaining. The new objects meant roles were separated out in a manner which in turn made new program code was easier to implement. In other words, there was proper demarcation of object roles. There was further refactoring involving the user interface which I will discuss in Chapter 6.

4.3 The meta heuristics

A major component of the project was to find a good algorithm for the search optimisation problem. In MPP1, I decided that Hill Climbing (HC) was the simplest and closest to what a human might do, and therefore started with this as a baseline. A Genetic Algorithm (GA) can be seen as an extended HC algorithm; it has a larger candidate population (an HC only has a single Candidate) a *crossover* function, and uses *tournament selection*, but otherwise it is similar to an HC algorithm. The GA has the advantage that it could potentially take advantage of threading; because it has a Candidate population larger than one the candidates could be evaluated in parallel.

In MPP1 I proposed using a Particle Swarm Optimisation (PSO) algorithm. A GA will 'hop' between positions in the search space. If we consider two GA candidates in fairly good positions, they are likely to *crossover* (swap elements in candidate) and therefore hop between positions they already know. A PSO on the other hand has agents that communicate with each other, two candidates in fairly good positions will have the effect of 'pulling' each other, which leads to the possibility of dragging each other through better positions.

4.3.1 Hill Climbing

The Hill Climbing algorithm used is as seen in the Minf Project Proposal and Minf Project Phase 1. There has been one change to the algorithm; an extension to its architecture so it can be used as the driving algorithm in any chaining search. This modification only meant allowing the HC algorithm to accept a Lab candidate and a Lab fitness function which had no effect on how the algorithm worked.

4.3.2 Genetic Algorithms

The Genetic Algorithm used is also as seen in the Minf Project Proposal and Minf Project Phase 1; there has been no change to this algorithm.

4.3.3 Particle Swarm Optimisation

In Chapter 2 I provided a generalised algorithm for a PSO, this subsection's objective is to provide a description of how this meta heuristic is implemented and executed for MPP2.

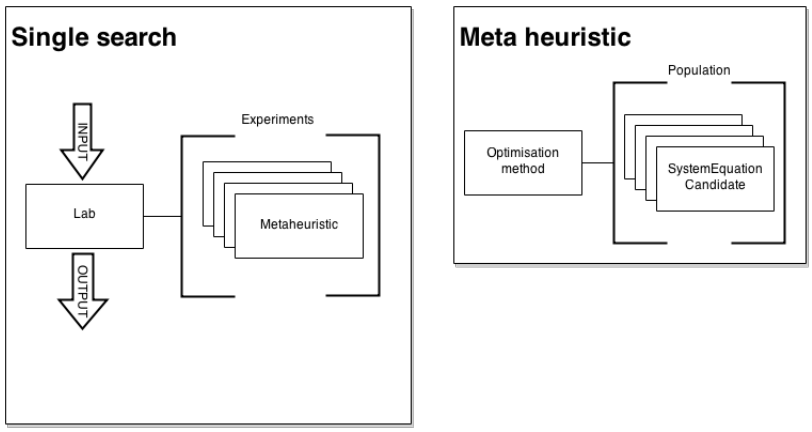


Figure 4.4: High level diagram of a single search job

A PSO is a single search (Figure 4.4). Single searches are identically implemented with the exception of the *optimisation method*. The optimisation method is the core implementation of the meta heuristic. As stated before the HC algorithm and GA have the same implementation as in MPP1, this is a discussion of the PSO.

A SystemEquationCandidate, from the perspective of a PSO, can be seen as a particle with a position in vector space. The position vector is a representation of a system equation, where each dimension represents a component (Figure 4.5).

$$A[x_1] \underset{s_2}{\bowtie} B[x_2] \underset{s_3}{\bowtie} C[x_3] \equiv (x_1, x_2, x_3)$$
where $x_i \in \mathbb{R}_{>0}$

Figure 4.5: A system equation seen as a vector

A PSO has a number of particles in a search space, each particle has its own position. These particles also have their own *velocity*. Each iteration of a PSO the particles move to another position depending on their velocity (Figure 4.6).

$$A[x_1] \underset{s_2}{\bowtie} B[x_2] \underset{s_3}{\bowtie} C[x_3] \equiv (x_1, x_2, x_3) = \bar{p}_t$$
Velocity:
$$A[x_1] \underset{s_2}{\bowtie} B[x_2] \underset{s_3}{\bowtie} C[x_3] \equiv (v_1, v_2, v_3)$$
$$\bar{v_{t+1}} = (v_1, v_2, v_3)$$
Next position:
$$\bar{p_{t+1}} = \bar{p}_t + \bar{v_{t+1}}$$

Figure 4.6: A particle moving to the next position $\bar{p_{t+1}}$

Each iteration a new velocity is created from a weighted sum of three vectors; a global-best, a local-best and the previous velocity of the particle. These three vectors are

weighted by α, β, σ (Figure 4.7). The global-best is the position of the fittest particle in the swarm so far, local-best is the best position that a particle has found so far.

$\bar{g} \leftarrow$ global-best
$\bar{l} \leftarrow$ local-best
$\bar{v_t}$ the previous velocity of the particle
$\bar{v_{t+1}} = (\alpha * \bar{v_t}) + (\beta * (\bar{g} - \bar{p})) + (\sigma * (\bar{l} - \bar{p}))$

Figure 4.7: Creating a new velocity from the position vector, local-best, global-best and velocity of the previous iteration

α, β, σ are the PSO settings, these are the settings the modeller can change:

- α determines how much a candidate should carry on with its previous heading.
- β determines how strongly a candidate should head toward the global-best position, setting this high causes the particles to rapidly converge on the best position.
- σ determines how strongly a candidate should favour its own best position, setting this high breaks the swarm into individual HC particles.

It should be easy to see then that if the position vector is the representation of the system equation, then the velocity is the mutation of, or change to, the system equation. The execution of the PSO is as follows

1) Initialisation. As PSO is a population based meta heuristic (or swarm) a number of SystemEquationCandidate particles are created. These particles are given an initial random velocity, a random starting position within the range specified by the user, and a fitness of 10,000.
2) Fitness test. As each particle is a SystemEquationCandidate it also has an underlying ASTHandler (see Section 3.1.2.2), when a particle is evaluated for fitness, first the ASTHandler updates the underlying AST. This then creates an ODE Graph ready for evaluation. Each particle is then evaluated for fitness, as per the SystemEquationFitnessFunction described in Section 3.1.2.4.
3) Update local/global. The population is then iterated through to find the global best particle, and to update each particle's local best position.
4) The velocity of each particle is updated as per Figure 4.7.
5) Each particle now 'moves': the new velocity is added to the position vector.
6) If we still have iterations left, go to 2) otherwise complete.

It is worth noting some extra implementation required in order use a PSO with system equations:

- Using a weighted sum to create new velocities creates non-integer values, I have put in an extra function that rounds new position vectors to the nearest integer before the new position is passed to the ASTHandler.

- A modeller will set ranges on the search space by defining the minimum or maximum population of each component. To keep the PSO inside of these ranges I needed to decide how to handle the edges of the search space, I decided to use a 'wrapping' function where a particle will re-appear on the other side of a search space when leaving the range.

4.3.4 Meta heuristic settings

In order to save confusion, I refer to meta heuristics as having settings instead of parameters and only ever use parameters when discussing the models. These settings effect the behaviour of the meta heuristics, and only effect the efficacy of the search. Table 4.1 is a 'feature' table of the different meta heuristic settings, these are the values the modeller can set when performing a search.

Setting	**HC**	**GA**	**PSO**
Experiments	x	x	x
Generations	x	x	x
Population		x	x
Mutation	x	x	
Crossover		x	
Original Velocity			x
Local Best			x
Global Best			x
Total	**3**	**5**	**6**

Table 4.1: A 'feature' list of the settings of each meta heuristic

I consider an HC algorithm to have one setting, but in the feature list above we see it has three. The experiment and generation settings are general search settings, they are not specific to any one meta heuristic. They are also reasonably clear in their effect; the number of experiments is the number of times we would like to run a search (see Chapter 3 for clarification on experiments), the generation count is the number of iterations we will allow each experiment to run for. Increasing either always has the effect of making the search take longer, but also increases the chance of finding the best candidate. Therefore although the capacity planning tool offers three settings for an HC algorithm, the HC has only one setting *specific* to it.

4.4 Improving the fitness function

The system equation fitness function, discussed in Chapter 3, is an improved version of the one offered in MPP1. These changes affected the user interface, the core PEPA code and the system equation fitness function. Any work on the user interface is seen in a later chapter.

Recall from Chapter 3 that the fitness function has component weighting and component ranges. These are two significant updates to the system equation fitness function:

- The component population weighting has been added to allow for finer tuning of component cost. A modeller can now say they care about the cost of one component more than another.
- The component population ranges have changed to allow fixed populations or to allow some components to be zero. The modeller can then search a space where it is possible for a component to be ignored completely.

4.4.1 Component population weights

As stated in Chapter 3 the weighting of components is now relative, if we have N components in the system equation then each individual component weight becomes:

$relativeComponentWeight_i = componentWeight_i / (\sum_{N=0}^{N} componentWeight_j)$

Now the user can fine tune the cost of each component relative to the others, and can even remove cost from a component entirely by using a weight of 0.

4.4.2 Component population ranges

In Subsection 3.1.2.4 I discussed **Component choice**; one of the requirements of this project was to enable the modeller to be able to search for an optimal configuration when there is a choice of resources. In Chapter 1 there was a demonstration of this problem where we had a choice of Application server to use in the example system. I stated that without a tool the modeller would need to create three models to solve this problem, one for each type of Application server, and one with both Application servers in parallel. This is due to the PEPA parser not allowing a zero population in a system configuration, the modeller cannot evaluate a model where one component has no population.

PEPA has an operator called choice, so to save from confusion I will refer to any model that offers a choice of resource as a *mixed agent type model* (Figure 4.9), mixed because the system equation could have a mix of agent types. These mixed agent types will always have an identical interface, but will have a different rate on some action, and possibly some difference in cost. Any model then that is not a mixed agent type model will be referred to as a *single agent type model*. In these models each component role within the model is supported by a single component type.

There are two updates to the population ranges in MPP2:

- The minimum and maximum range are now allowed to be equal, so that the user can fix the population of one component. This is a minimal change.
- The minimum range can now be zero, allowing components to have no population in the system equation. This means we can provide a more flexible fitness

test when we have mixed agent type models. When searching on mixed agents the modeller may be interested to know if one type of agent is required at all in the system equation. This was an involved piece of work and is discussed in the following sub-section.

4.4.2.1 Setting a population to zero

In order to be able to have a population of zero in any component I had to modify the core PEPA code and test for any unwanted behaviour.

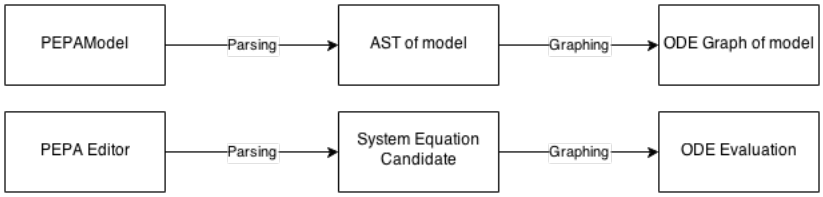


Figure 4.8: **Top:** The stages of a PEPA model from left to right: the user creates a PEPA model in the editor, an AST is created of the PEPA model, the AST is passed through a graphing function to create the ODE graph from which we get the performance results. **Bottom:** How the capacity planning tool uses the above stages: The user enters the model into the PEPA editor, the capacity planning tool uses ASTs to manipulate system equations, ASTs are passed to the ODE Evaluator to obtain the performance results.

Figure 4.8 shows the stages of a PEPA model when being evaluated using ODEs. I had to choose at which point the capacity planning tool would change the component populations of the underlying model. The AST was the best choice as this is beyond the parsing tool (which rejects populations of zero) and is where any system equation manipulation happens anyway.

Next I had to identify any existing core code which would prevent the ODE from being able to accept a population of zero. I found there were two equality checks in the core PEPA code that tested that if populations were above zero.

I changed these equality tests so that they would accept a population of zero, and then created two tests to ensure that there was no unexpected behaviour due to this change.

I created six models; three were single agent type models, the other three were copies of those single agent type models turned into mixed agent type models by duplicating one component. The duplicated resource then was given a different rate on one action. Figure 4.9 is an example of the system equations from two of the models.

For the first test (Figure 4.10), let us assume that the Application servers have a different rate on the action `update`, with `updateRate1` for ApplicationServer and `updateRate2` for ApplicationServer2, where `updateRate1` $!=$ `updateRate2`. The single agent type model only has ApplicationServer, whereas the mixed agent type model uses both resources in parallel as seen in Figure 4.9.

An example single agent type model system equation:

$$User_0[x_1] \underset{s_1}{\bowtie} WebServer_0[x_2] \underset{s_2}{\bowtie} ApplicationServer_0[x_3] \underset{s_3}{\bowtie} DatabaseServer_0[x_4]$$

An example mixed agent type model system equation:

$$User_0[x_1] \underset{s_1}{\bowtie} WebServer_0[x_2] \underset{s_2}{\bowtie} (ApplicationServer_0[x_3] || ApplicationServer2_0[x_4]) \underset{s_3}{\bowtie} DatabaseServer_0[x_5]$$

Figure 4.9: The system equations of the two different types of model, the top model has single type of ApplicationServer and therefore is a single type model, the bottom model has ApplicationServer and ApplicationServer2 in parallel making it a mixed type model, it is understood that these two agents have the same interface, but a different rate on some action. From the perspective of the capacity planning tool the bottom equation is offering a choice of Application server, not to be confused with the PEPA choice operator.

Then we can say, if the mixed agent type model has the same population as the single agent type model in all components and ApplicationServer2 has a population of zero. We define unwanted behaviour to be when any performance evaluation of the mixed agent type has a relative difference of over 1% when compared to the performance evaluation of the single agent type for any population.

For the second test (Figure 4.11), let us assume that the Application servers have the *same* rate on the action `update`, with `updateRate1` for ApplicationServer and `updateRate2` for ApplicationServer2, where `updateRate1` $==$ `updateRate2`.

Then we can say, for the mixed agent type model, for any population of ApplicationServer and ApplicationServer2 where the sum of their population is equal to the population of the ApplicationServer component in the single agent type model, and all other populations are equal. We define unwanted behaviour to be when any performance evaluation of the mixed agent type has a relative difference of over 1% when compared to the performance evaluation of the single agent type for any rate and population.

In summary, test 1 ensures that no unwanted behaviour happens when we introduce a zero population *anywhere* in the system equation, whereas test 2 ensures that the sum of components in parallel produce the same result as an individual component. In short; can we use zero populations? and, can we use mixed agents in parallel?

For brevity I include the results from only one model, the 'Brewery' model which I have used all the way through testing the extension. The Brewery model can have two different Brewery components `Brewery` and `Brewery2`, these both have an action `produce`. `Brewery` has a rate of `p1` on `produce`, `Brewery2` has a rate of `p2`.

If:

$\texttt{Single} = User_0[x_1] \underset{s_1}{\bowtie} WebServer_0[x_2] \underset{s_2}{\bowtie} ApplicationServer_0[x_3]$

$\underset{s_3}{\bowtie} DatabaseServer_0[x_4]$

And:

$\texttt{Mixed} = User_0[x_4] \underset{s_1}{\bowtie} WebServer_0[x_5] \underset{s_2}{\bowtie}$

$(ApplicationServer_0[x_6] || ApplicationServer2_0[x_7]) \underset{s_3}{\bowtie} DatabaseServer_0[x_8]$

Where:

For all components in `Single` that are also in `Mixed` the rates are the same

For all components not in `Single` the rates are different

And:

$x_i \in \mathbb{R}_{>0}$

$x_1 \equiv x_4, x_2 \equiv x_5, x_3 \equiv x_6, x_4 \equiv x_8,$ and $x_7 = 0$

Then:

$$\frac{f(\texttt{Single})}{f(\texttt{Mixed})} < 0.01$$

Where:

$f() =$ any performance evaluation

Figure 4.10: Example of ”Test 1” on the Example system from Chapter 1

Tables 4.2 and 4.3 show the populations of components used in the models and then what rates were used during testing, notice the spread of population and type of rate determines the type of test. Tables 4.4 and 4.5 are the performance evaluation results for every action in this model. These tables only show the results for throughput; average response time was also tested. Finally Table 4.6 is the average difference between performance results, case 6 and case 8 are expected to show a difference of over 1%.

In total I evaluated these tests on 6 models, in conclusion the tests gave evidence to suggest that there was never any unwanted behaviour in either using zero populations in ODE functions, or when using components in parallel (this data is too large and out of scope of this document and so I have not included it, however the results support my claim as they are similar). However this has allowed me to use this method to implement zero behaviour and therefore a more flexible fitness function.

If:

$\texttt{Single} = User_0[x_1] \underset{s_1}{\bowtie} WebServer_0[x_2] \underset{s_2}{\bowtie} ApplicationServer_0[x_3]$

$\underset{s_3}{\bowtie} DatabaseServer_0[x_4]$

And:

$\texttt{Mixed} = User_0[x_4] \underset{s_1}{\bowtie} WebServer_0[x_5] \underset{s_2}{\bowtie}$

$(ApplicationServer_0[x_6] || ApplicationServer2_0[x_7]) \underset{s_3}{\bowtie} DatabaseServer_0[x_8]$

Where:

For all components in `Single` that are also in `Mixed` the rates are the same

For all components not in `Single` the rates in `Mixed` are the same as their duplicate

And:

$x_i \in \mathbb{R}_{>0}$

$x_1 \equiv x_4, x_2 \equiv x_5, x_3 = (x_6 + x_7), x_4 \equiv x_8$

Then:

$$\frac{f(\texttt{Single})}{f(\texttt{Mixed})} < 0.01$$

Where:

$f() =$ any performance evaluation

Figure 4.11: Example of ”Test 2” on the Example system from Chapter 1

Case	Model type	System Equation population			
		Well	Farm	Brewery	Brewery2
0	Single baseline	100	100	40	n/a
1	Mixed	100	20	100	20
2	Mixed	100	0	100	40
3	Mixed	100	40	100	0
4	Mixed	100	10	100	30
5	Mixed	100	30	100	10
6	Mixed	100	20	100	20
7	Mixed	100	0	100	40
8	Mixed	100	40	100	0
9	Single	100	100	40	n/a

Table 4.2: This table shows the populations of each component used in each case

4.5 Further Implementation: Chaining

In this section I present the implementation required in order to build some further implementation called *chaining*.

Case	Model type	Rates		Test type
		p1	p2	
0	Single baseline	100	n/a	n/a
1	Mixed	100	100	2
2	Mixed	100	100	1 and 2
3	Mixed	100	100	1 and 2
4	Mixed	100	100	2
5	Mixed	100	100	2
6	Mixed	100	1000	2
7	Mixed	100	1000	1
8	Mixed	100	1000	1
9	Single	1000	n/a	n/a

Table 4.3: An example of the 9 tests run on 2 models, the test type relates to figures 4.10 and 4.11, which are the two different kinds of test that have been discussed.

Case	APV (Throughput)						
	sow	store_w	store_f	tend	sink	reap	produce
0	16.62	16.62	16.62	16.62	16.62	16.62	16.62
1	16.62	16.62	16.62	16.62	16.62	16.62	16.62
2	16.62	16.62	16.62	16.62	16.62	16.62	16.62
3	16.62	16.62	16.62	16.62	16.62	16.62	16.62
4	16.62	16.62	16.62	16.62	16.62	16.62	16.62
5	16.62	16.62	16.62	16.62	16.62	16.62	16.62
6	16.65	16.65	16.65	16.65	16.65	16.66	16.65
7	16.62	16.62	16.62	16.62	16.62	16.62	16.62
8	16.68	16.68	16.68	16.68	16.68	16.69	16.68
9	16.68	16.68	16.68	16.68	16.68	16.69	16.68

Table 4.4: These are the performance evaluation results of each action in each case in table 4.3. APV stands for Absolute Performance Value, the figures are shown with four significant figures.

Case	RPD						
	sow	store_w	store_f	tend	sink	reap	produce
0	0.00E+000	0.00E+000	0.00E+000	0.00E+000	0.00E+000	0.00E+000	0.00E+000
1	3.35E-006	1.22E-004	2.20E-004	5.23E-006	1.48E-005	2.83E-005	2.42E-002
2	5.23E-006	3.42E-006	9.84E-006	4.12E-005	1.19E-004	2.21E-004	8.53E-004
3	5.23E-006	3.42E-006	9.84E-006	4.12E-005	1.19E-004	2.21E-004	8.53E-004
4	1.39E-005	1.94E-005	2.57E-005	1.05E-004	2.96E-004	5.61E-004	3.42E-003
5	1.39E-005	1.94E-005	2.57E-005	1.05E-004	2.96E-004	5.61E-004	3.42E-003
6	1.90E-001	1.87E-001	1.88E-001	2.03E-001	1.69E-001	2.57E-001	2.10E-001
7	5.23E-006	3.42E-006	9.84E-006	4.12E-005	1.19E-004	2.21E-004	8.53E-004
8	3.78E-001	3.75E-001	3.76E-001	3.90E-001	3.56E-001	4.44E-001	4.03E-001

Table 4.5: These are the relative performance difference (RPD) between cases 1-8 and case 0 from Table 4.4, for throughput of all actions in the models.

Case	Average	As %
0	0.00E+00	0
1	3.52E-03	0.35
2	1.79E-04	0.02
3	1.79E-04	0.02
4	6.34E-04	0.06
5	6.34E-04	0.06
6	2.01E-01	20.07
7	1.79E-04	0.02
8	3.89E-01	38.89

Table 4.6: This table shows the average result of each performance value of each test case taken from table 4.5, and finally the percentage difference. Case 6 and 8 should have a value of above 1%: these are the cases when the rates are in favour of p2 and not p1.

4.5.1 Chaining

The Genetic Algorithm (GA) was shown to be under performing when compared to a Hill Climbing (HC) algorithm in MPP1:

Algorithm	Average Fitness	Run time	Best found time
HC	17	20 seconds	6 seconds
GA	28	2 seconds	0.85 seconds

Table 4.7: Results from MPP1, comparing a GA to an HC algorithm, the 'best time found' is the average time it took to find the top fitness, the run time is how long the algorithms took to complete

Table 4.7 shows us that, on average, the HC algorithm finds a fitter system equation candidate than the GA. However the GA does converge and complete faster than the HC.

In MPP1 the HC was implemented so that it would reject any candidate that had a worse fitness than the current candidate and attempt a new mutation. This rejection meant that the generation count was not being increased and therefore the HC was turned into a greedy and brutal search; it was allowed to keep searching the search space until it had found a better candidate without penalty. If we is count rejected mutations, the HC completes in a quarter of the time a GA takes (within 50 milliseconds), with the top fitness on average found within 25 milliseconds.

I conjectured that the GA may have been under performing because it had not been given the correct settings, and this problem would also be seen with the Particle Swarm Optimisation.

Further, during the development it became clear that the requirement for the modeller to have to understand the meta heuristics well enough in order to be able to define the settings might be infeasible.

Therefore I proposed two further implementations for the capacity planning tool where the meta heuristics could be used in conjunction. I called such combinations of meta heuristics 'chaining' and considered two particular forms: pipe-lining and driven searches.

4.5.2 Pipe-line search

A pipe-line search was to be a configuration of meta heuristics where one 'fast' meta heuristic would find a reduced but optimal section of the search space, and pass this reduced search space to a second more accurate meta heuristic.

Although the framework exists for pipe-lining and the architecture is ready for this type of chaining, I did not have enough time to implement, test and evaluate this kind of search. I decided that out of the two chaining techniques pipe-lining was probably the weakest, as it still left the modeller with the need to define the meta heuristic settings. In fact it would have made this problem worse as the modeller would need to choose the settings for two meta heuristics instead of one.

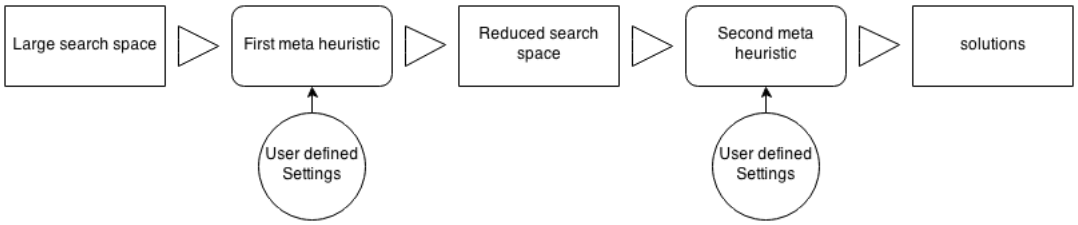


Figure 4.12: Pipe-lining meta heuristics, the first meta heuristic reduces the larger search space for a second more accurate meta heuristic. Notice there are two user defined settings, one for each meta heuristic.

4.5.3 Driven search

A driven search uses one meta heuristic to find the settings of a second meta heuristic; The purpose of this is to use a meta heuristic to fine tune the settings of a second meta heuristic, and to relieve the modeller of requiring to understand the purpose of all the meta heuristic settings.

Supporting a driven search involved some further work on the core implementation. There are three kinds of driven search, a driven HC, a driven GA and a driven PSO. The HC algorithm is the logical choice for the *driving* heuristic, as the modeller needs to enter only one setting, therefore in all three cases there is an HC algorithm driving the settings for the second algorithm.

I designed and implemented the LabCandidate class, which is a 'wrapper' for a single search as seen in Chapter 3. An HC algorithm can evaluate the efficiency of a particular configuration of meta heuristic settings by evaluating a LabCandidate.

The driving meta heuristic determines the settings of the driven meta heuristic; I needed to decide what range the driving meta heuristic could use for experiments, generations

and populations for the driven meta heuristic. Allowing too large a number on any of these would make the search infeasible as the run time would be too long. I elected to allow a range of 1 to 10 for all, except in the case where an HC drives an HC, in this case the population would be 1, and therefore the generations could be 100.

Otherwise using an HC algorithm to search on meta heuristic settings is straightforward, if we treat the driven meta heuristic settings as a vector the HC algorithm works on a candidate vector of settings with its mutation function.

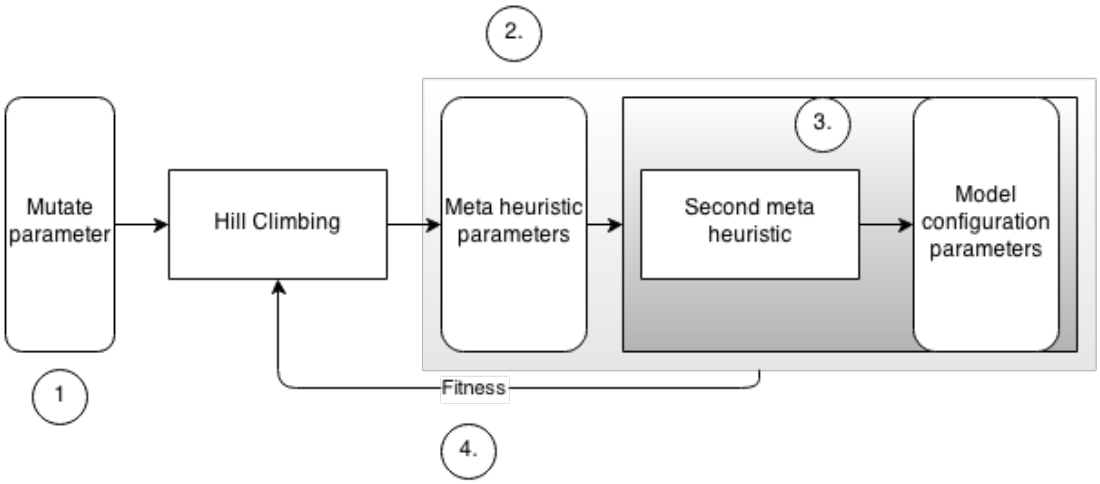


Figure 4.13: Driven search

Figure 4.13 is a diagram showing the work flow of a driven search for one LabCandidate, for one generation iteration:

1. The user sets a mutation rate for the Hill Climbing algorithm.
2. The Hill Climbing algorithm creates meta heuristic parameters suitable for the second meta heuristic.
3. The second meta heuristic attempts a search for an optimal model configuration using its given parameters.
4. The fitness of this meta heuristic configuration, as described in chapter 3, is passed back to the Hill Climbing algorithm.

I also updated the RecordManager class so it could be evaluated by the LabFitnessFunction. For a single search the RecordManager simply outputs the top results to the capacity planning Viewer, for a driven search the RecordManager passes the results to the LabFitnessFunction in order for the fitness to be evaluated (Figure 4.14).

A considerable portion of the work on driven searches was in refactoring the framework (seen in Section 4.2) so that a meta heuristic could run on any type of Candidate and testing. Testing was particularly difficult as I had to make sure the results from a stochastic search on a stochastic search were sound. This meant running many test models.

I had originally intended to test using a GA or PSO as a driving algorithm as well. Again the framework exists to make this possible, but the complexity of testing the

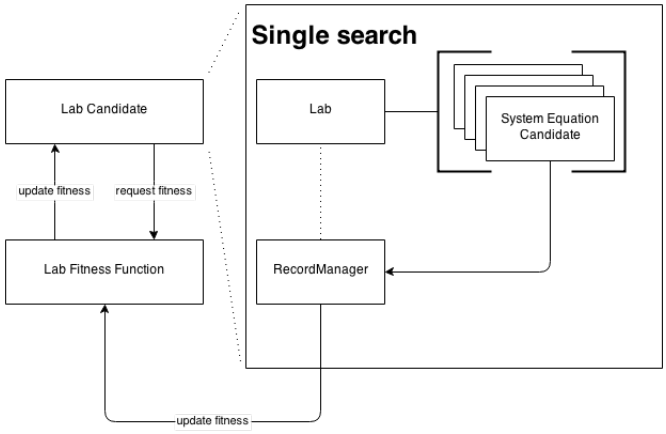


Figure 4.14: Lab candidate calling a fitness update

configuration with just an HC algorithm meant that I took longer in testing and implementation than I would have liked. I believe however that the advantage of using the HC with only one setting is a strong enough reason to choose the HC algorithm over any other. In this chapter we have seen the work and design decisions of MPP2, in the following chapter we look at the evaluation of the implementation, which provides evidence towards which choice of meta heuristics to use in the final capacity planning tool version.

Chapter 5

Evaluation

5.1 Introduction

There are four kinds of evaluation necessary for this project, three of which are related to the internal mechanics of the extension and are going to be discussed in this chapter. The fourth relates to the user interface, and this will be discussed in chapter 6.

Single meta heuristic evaluation: is used to determine which of the three algorithms will be the best to use in the final capacity planning version. The evaluation looks at the best returned solution, the mean fitness and standard deviation of the top solutions across experiments and the response time of the algorithm.

Driven versus single evaluation: In previous chapters I made the claim that a driven meta heuristic will have more success than a single meta heuristic because of the tuning of the second meta heuristic to the search problem. Evaluation is carried out here to provide evidence that the driven search is indeed a superior search.

Driven evaluation: Finally this evaluation determines which of the combinations of meta heuristic is going to be best for a driven search. This dictates the choice of the driven search offered to the modeller in the final plugin extension.

5.2 The evaluation method

Eight different models were used in the evaluation (Appendix A). There are five single agent type models, and three 'extended' models, where three of the single agent type models were extended into mixed agent type models.

Each algorithm ran for 1000 experiments on each model. In order to make the evaluation fairer, the Hill Climbing (HC) was given 100 generations, whereas the Genetic

Algorithm (GA) and Particle Swarm Optimisation (PSO) had 10 candidates and 10 generations. This is the equivalent of having 100 chances to find a top candidate solution.

Six of the models had throughput as a performance target, and the remaining three were looking for an average response time.

5.3 Single meta heuristic evaluation

For the single meta heuristic evaluation the HC and GA were given the same default parameters as in MPP1, the PSO was given an equal split between its parameters; `localBest = originalVelocity = globalBest = 1/3`. This means that none of the algorithms has been tuned for a particular search space.

Each model was evaluated on four evaluation criteria;

- Top fitness: This is the best fitness value returned by the algorithm after running 1000 experiments.
- Mean fitness: This is the mean fitness of all 1000 experiments, and can be seen as how well the algorithm did overall.
- Standard deviation: This is the standard deviation from the mean of 1000 experiments; this can be seen as how accurate each algorithm is.
- Average response time of the algorithm: The average response time of running one experiment, how long the algorithm took to complete each experiment.

5.4 Single meta heuristic evaluation results

If we take HC as a baseline, as it is probably the most similar algorithm to the one a modeller might do by hand, we can show the relative difference between the algorithms.

Tables 5.1 and 5.3 are the results found from running each meta heuristic over the eight models. Table 5.1 shows the top fitness value (as returned by the fitness function), and the average length of time the meta heuristic took to complete for that model. It is worth noting that the 'E University' model took considerably longer than any other model to complete. This is due to the order of magnitude difference within the rates for that model making the ODEs stiff. Table 5.3 shows the average fitness of each experiment, and its standard deviation.

Meta heuristic	Model	Top fitness	Response Time (ms)
GA	Brewery	21.1459190469	31.4875
HC	Brewery	23.7709845399	46.15
PSO	Brewery	22.6989297536	29.075
GA	Brewery ab	18.8108704311	220.38875
HC	Brewery ab	16.3774794548	217.1575
PSO	Brewery ab	14.1821291257	208.0525
GA	E University	11.8022549847	1723.17
HC	E University	10.5644015294	1327.025
PSO	E University	5.221605961	1653.355
GA	Example System	8.2600534551	46.135
HC	Example System	7.3310791863	46.75
PSO	Example System	3.9171013554	43.975
GA	Example System ab	6.5884332866	75.75
HC	Example System ab	7.6583992229	75.67
PSO	Example System ab	1.75360953	71.2
GA	Large-t	2	120.71
HC	Large-t	2	122.68
PSO	Large-t	2	119.95
GA	Simple	8.0049356892	29.17
HC	Simple	7.5036812611	33.025
PSO	Simple	7.2500827539	30.095
GA	Simple ab	9.0345147758	33.82
HC	Simple ab	11.0039640775	28.45
PSO	Simple ab	4.8659176832	26.7
GA	Traffic	34.2665069405	33.05
HC	Traffic	34.4215069375	33.32
PSO	Traffic	32.9015069647	30.87

Table 5.1: Evaluation criteria results for Top fitness and Response time (ms)

Table 5.2

Meta heuristic	Model	Mean fitness	Standard Deviation
GA	Brewery	28.6941628835	4.735574615
HC	Brewery	28.3048428707	2.4917109254
PSO	Brewery	25.9490962347	3.3529018951
GA	Brewery ab	26.3514942917	5.4802970356
HC	Brewery ab	21.8615543701	2.8833072566
PSO	Brewery ab	17.8455207333	3.4298099258
GA	E University	25.6979984392	7.6439259233
HC	E University	22.5450728521	5.0469960427
PSO	E University	12.001243269	4.1554127297
GA	Example System	23.1396448038	12.1868790895
HC	Example System	17.1573938933	4.9289361721
PSO	Example System	8.4170294212	2.9652322935
GA	Example System ab	28.1695903265	11.7339905487
HC	Example System ab	19.6831779352	5.8050984466
PSO	Example System ab	7.4273299426	3.2678241622
GA	Large-t	108.68	85.4796911553
HC	Large-t	36.31	32.0075600445
PSO	Large-t	6.8375	23.3519639163
GA	Simple	20.0596667468	8.6051191398
HC	Simple	13.83626092	3.8211821347
PSO	Simple	9.818544418	3.2378342572
GA	Simple ab	19.3404495631	7.5977886355
HC	Simple ab	29.0989416307	18.8160176814
PSO	Simple ab	8.9800196614	3.3855659563
GA	Traffic	38.1797069441	2.2437837278
HC	Traffic	36.6633569439	1.2023504978
PSO	Traffic	33.1361069596	0.5031935185

Table 5.3: Evaluation criteria results for mean fitness and st. deviation

In all four evaluation criteria the smaller the value the better the algorithm has run for that evaluation criteria. By dividing each evaluation criteria of the HC by the PSO or the GA, in every model, we can see how the algorithms are performing relative to each other. The larger the value the better the search is performing.

Tables 5.4 and 5.5 show on average, relative to the HC, how well the GA and PSO are performing respectively. For example, the top fitness for the Brewery model, in table 5.4 shows that the GA finds a top fitness 12% 'better' over 1000 experiments than the HC, where as in table 5.5 the same position shows the PSO only finds a 5% improvement.

The last row shows the mean values. From this row we can see in Table 5.4 the GA performs equally, or worse than an HC on average, whereas in Table 5.5 on average the PSO performs nearly 2 times better than the HC on nearly every performance measure.

Model	Fitness ratio	Mean ratio	St. Dev ratio	Resp Time ratio
Brewery	1.12	0.99	0.53	1.47
Brewery ab	0.87	0.83	0.53	0.99
E University	0.90	0.88	0.66	0.77
Example System	0.89	0.74	0.40	1.01
Example System ab	1.16	0.70	0.49	1.00
Large-t	1.00	0.33	0.37	1.02
Simple	0.94	0.69	0.44	1.13
Simple ab	1.22	1.50	2.48	0.84
Traffic	1.00	0.96	0.54	1.01
Mean:	1.01	0.85	0.72	1.03

Table 5.4: Genetic Algorithm performance divided by Hill Climbing performance

Model	Fitness ratio	Mean ratio	St. Dev ratio	Resp Time ratio
Brewery	1.05	1.09	0.74	1.59
Brewery ab	1.15	1.23	0.84	1.04
E University	2.02	1.88	1.21	0.80
Example System	1.87	2.04	1.66	1.06
Example System ab	4.37	2.65	1.78	1.06
Large-t	1.00	5.31	1.37	1.02
Simple	1.03	1.41	1.18	1.10
Simple ab	2.26	3.24	5.56	1.07
Traffic	1.05	1.11	2.39	1.08
Mean:	1.76	2.22	1.86	1.09

Table 5.5: Genetic Algorithm performance divided by Particle Swarm Optimisation performance

5.4.1 Single meta heuristic evaluation conclusion

The Particle Swarm Optimisation (PSO) algorithm is clearly the best algorithm for this kind of search. Although it can take 10% longer to run than the Hill Climbing

(HC) algorithm it is on average twice as good as the HC algorithm in returning a fit candidate. The reason for this comes from the optimisation method. An HC algorithm could be seen as too random. It has no memory and only uses a single agent. The PSO on the other hand does have memory; the agents retain some knowledge of a previous better position, and the agents broadcast fit positions. Both of these factors limit the randomisation in movement. The Hill Climbing sees a speed boost because it is literally a simpler method to run; a Particle Swarm has more nested loops than the Hill Climbing algorithm.

This can be shown visually, using the example system again from the introduction:

$$User_0[x_1] \underset{s_1}{\bowtie} WebServer_0[x_2] \underset{s_2}{\bowtie} ApplicationServer_0[x_3] \underset{s_3}{\bowtie} DatabaseServer_0[x_4]$$
where $x_i \in \mathbb{R}_{>0}$

Figure 5.1: The example system's system equation.

If we fix $x_4 = 500$, and allow the PSO and HC to search for an optimal configuration on some arbitrary performance measure we can plot each generation iteration on a three dimensional graph. The number of Application servers is the x-axis, the number of Web servers is the y-axis and the number of Database servers is the z-axis. Using colour to show fitness, a scale from black to bright red gives an indication of the fitness at that point, bright red being the fittest (a black arrow indicates the fittest found area).

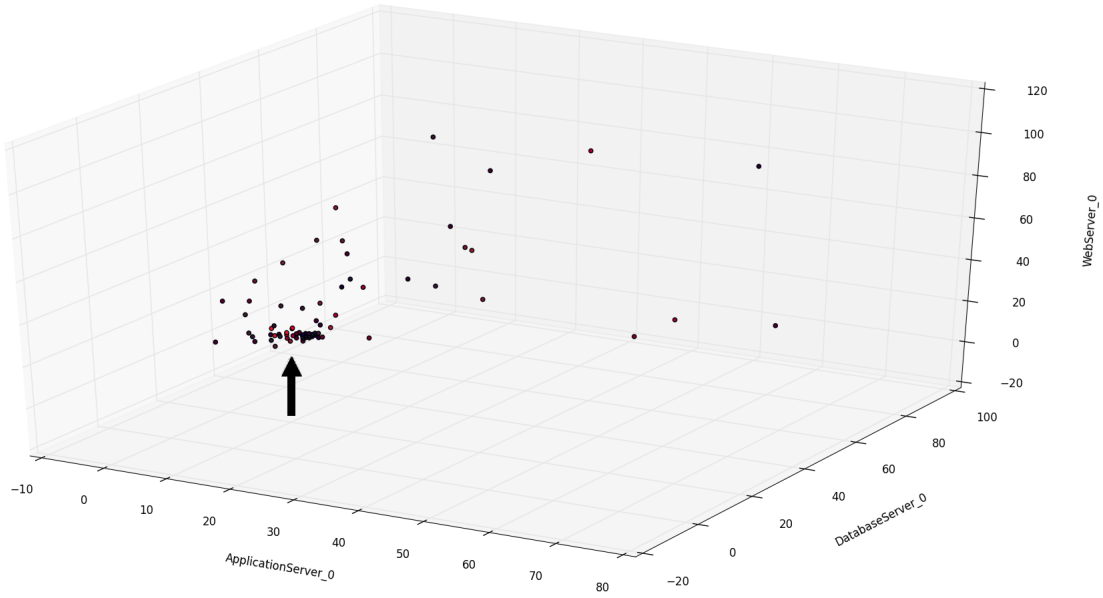


Figure 5.2: This graph shows the candidate swarm of a PSO search. The points clearly show some spread candidates, and also some clustering in the bottom left corner. This demonstrates the converging of candidates.

Figure 5.2 captures the PSO agents moving through space as they are searching for an optimum candidate; there is a tight grouping of agents in the lowest left end of the graph. This is where the agents have converged onto, the end position of the algorithm.

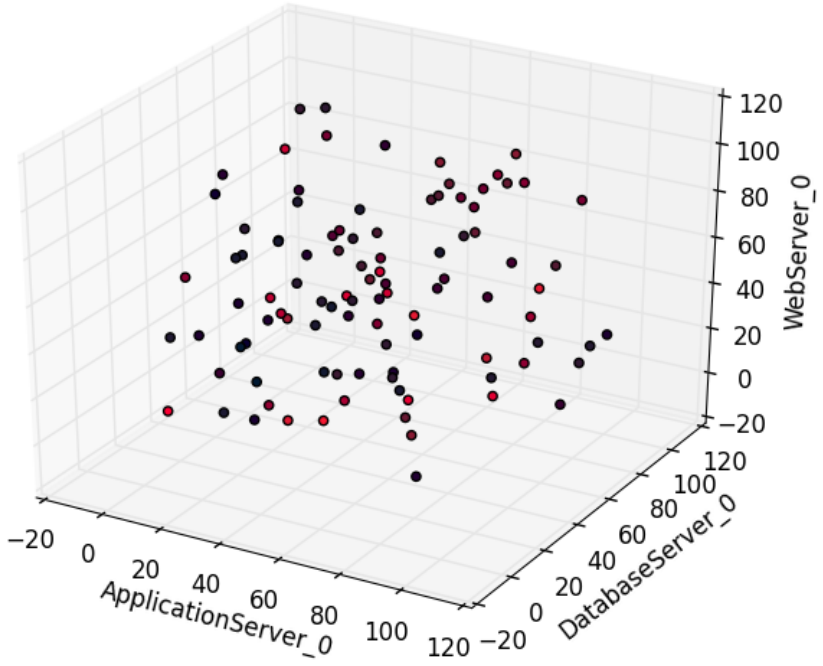


Figure 5.3: This graph shows the randomised search method of the HC algorithm. In contrast to the PSO the candidates are spread around the entire search space.

The agents widely scattered about the center of the graph are actually the agent starting positions. I believe it is relatively easy to see that there is a funnel shape: this is a pattern created from the agents communicating with each other, eventually all the agents are being pulled into the lower left of this search space.

Figure 5.3 captures the more random nature of the HC algorithm, the agent is dotted randomly about the search space, and comparing this against Figure 5.2 it is clear now that the PSO has some emergent behaviour.

Further insight is given when we look at what happens with fitness over generations. By running each algorithm 1000 times on the same model we can find the average fitness of each generation; we would expect a good algorithm to have the average fitness getting better (decreasing) as the generation iteration increases.

Figure 5.4 shows the average fitness and the variance of fitness over generations of the PSO on our example model. It shows that on average the PSO has converged after 8 generations because we see no real improvement in average fitness after that. It also shows the algorithm consistently finding a better candidate over time. In contrast to figure 5.5 we see randomised behaviour: on average the HC has no more success over time in finding a better candidate. Conversely we can see the Genetic Algorithm (GA) performing badly in Figure 5.6. The agents are on average converging on the 5th

generation, but we can see that there is a widely spread variance, which means there are better positions found *but not always being found*. Worse still the generations after the 5th are still occasionally finding better candidates. The width of the variance shows the GA to have quite different behaviour each run; compared to Figure 5.4 where the end generations have a much tighter variance. This shows that the PSO on average has the same behaviour independent of how the algorithm was started, it is more predictable, and also has a better success rate than either the HC or GA.

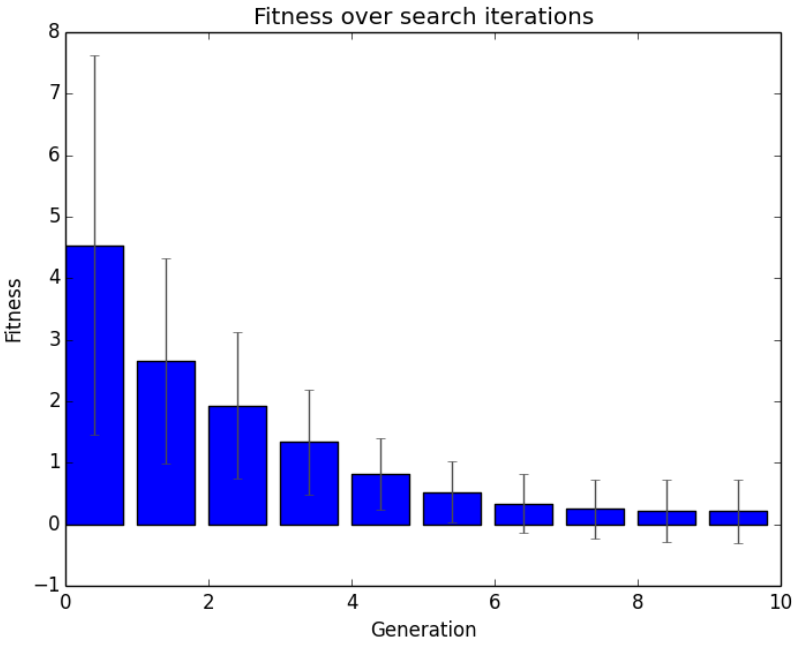


Figure 5.4: This graph shows the average fitness and variance of fitness over iterations of generations of a Particle Swarm Optimisation.

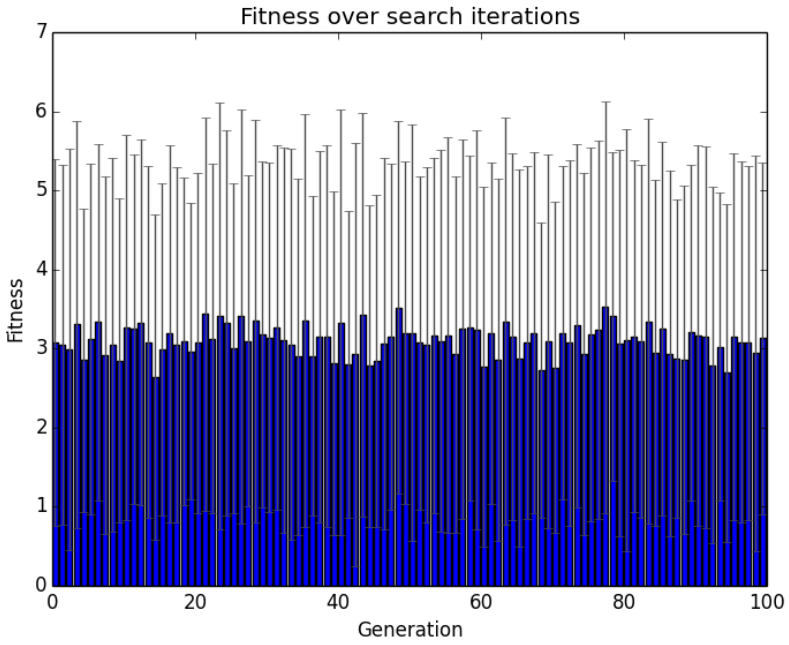


Figure 5.5: This graph shows the average fitness and variance of fitness over iterations of generations of a Hill Climbing algorithm.

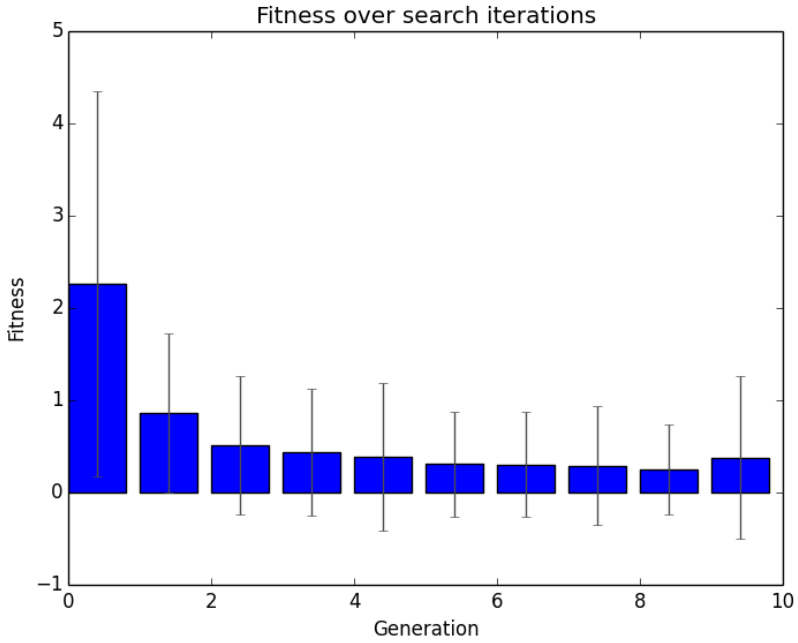


Figure 5.6: This graph shows the average fitness and variance of fitness over iterations of generations of a Genetic Algorithm.

5.5 Further Evaluation

5.5.1 The evaluation method revisited

The next steps in the evaluation are to determine if a driven search has any benefits at all over a single search, and if it does then which kind of driven search is the best. A

driven search generates huge amounts of data per run, with each run creating in the order of 15Gb of data. This made it infeasible for me to be able to do the same kind of analysis as I have done with single searches. Therefore I elected to look at using top fitness as the only evaluation criteria when comparing driven searches. I think this can be somewhat justified when we consider the lab fitness function:

$$\begin{aligned} fitnessValue = &(topFitness * 0.6) + (meanFitness * 0.2) \\ &+ (standardDeviation * 0.1) + (averageResponseTime * 0.1) \end{aligned} \quad (5.1)$$

Figure 5.7: Lab fitness function

The lab fitness function itself uses the same evaluation criteria as being used in the single search evaluation. I conjecture that when we see a driven search performing well on top fitness we can also make an assumption that the mean and standard deviation of that driven search is performing at least as well as a comparable single search. That is we could expect a top fitness candidate to have a good mean, standard deviation and average response time.

The evaluation method otherwise is the same as for the single search evaluation, it was run over the same set of models, but now with the evaluation criteria reduced to the top fitness found.

5.5.2 Driven evaluation results

Model	Search	fitness
Brewery a	Driven	23.56
Brewery a	Single	24.65
Brewery ab	Driven	15.85
Brewery ab	Single	18.51
E University	Driven	12.87
E University	Single	12.82
Example system a	Driven	4.48
Example system a	Single	8.48
Example system ab	Driven	3.98
Example system ab	Single	14.34
Large-t	Driven	2.00
Large-t	Single	2.00
Simple a	Driven	7.25
Simple a	Single	10.75
Simple ab	Driven	5.18
Simple ab	Single	9.10

Table 5.6: The fitness results of the driven Hill Climbing algorithm against the single Hill Climbing algorithm.

Model	Search	Fitness
Brewery a	Driven	23.5281446663
Brewery a	Single	24.7544292047
Brewery ab	Driven	18.7876607527
Brewery ab	Single	23.5009997145
E University	Driven	8.5128012326
E University	Single	18.3208776397
Example system a	Driven	4.3097801751
Example system a	Single	10.2452560245
Example system ab	Driven	2.7553149057
Example system ab	Single	12.6729421658
Large-t	Driven	2
Large-t	Single	2.82
Simple a	Driven	7.2500827539
Simple a	Single	8.2500827702
Simple ab	Driven	5.1487848267
Simple ab	Single	10.8015858398

Table 5.7: The fitness results of the driven Genetic Algorithm against the single Genetic Algorithm.

Model	Search	fitness
Simple ab	Driven	5.0052345479
Simple a	Driven	7.2500827539
Brewery ab	Driven	18.6423571384
Brewery a	Driven	22.3579375684
E University	Driven	5.117539601
Example system a	Driven	3.8240994735
Example system ab	Driven	2.7833476478
Large-t	Driven	2
Simple ab	Single	6.3042747912
Simple a	Single	7.2500827539
Brewery ab	Single	15.1921460447
Brewery a	Single	23.1876468559
E University	Single	6.0621683007
Example system a	Single	5.215208333
Example system ab	Single	4.9097955332
Large-t	Single	2

Table 5.8: The fitness results of the driven Particle Swarm Optimisation against the single Particle Swarm Optimisation.

5.5.3 Driven versus single search evaluation

By comparing each driven meta heuristic with its single analogue we can see if there is any improvement in candidate fitness. Simply by dividing the top fitness of the single search by the top fitness of the driven search we can test for any improvement. Table 5.9 shows the improvement by using a driven search by model and finally the average improvement across all models.

Model	HC Fitness ratio	GA Fitness ratio	PSO Fitness ratio
Simple ab	1.7578063972	2.0978903185	1.2595363376
Simple a	1.4824195635	1.1379294624	1
Brewery ab	1.1682106228	1.2508741787	0.814926242
Brewery a	1.0461987174	1.0521198996	1.0371102784
E University	0.9963825196	2.1521561633	1.1845864953
Example system a	1.8935419852	2.3772108108	1.3637742347
Example system ab	3.6002243735	4.5994532747	1.7639893231
Large-t	1	1.41	1
mean	1.6180980224	2.0097042635	1.1779903639

Table 5.9: A Table to show single top fitness divided by driven top fitness and the mean of the results.

Nearly all of the meta heuristics are improved by using a driven search. Therefore a driven search is certainly worthwhile to have as an option for model configuration searches. The GA sees the most improvement from the driven technique as it is, rel-

atively, two times better at finding a fit model than it's single counterpart. This is evidence towards my MPP1 conjecture that the GA was under performing due to algorithm settings.

The fact that the HC sees such a big improvement is actually surprising, the driven algorithm nearly always favoured a mutation rate of around 90%, an almost completely random search of the space on every dimension. This result comes from the fact that the search space is actually quite small, so the higher the rate that a candidate mutated meant the candidate had more chance to land on a fitter candidate position.

The PSO only sees a 17% improvement on average, but this small increase is still going to be an improvement. The PSO has the most amount of parameters to set so the driven search does still have the benefit of reducing the number of parameters the modeller has to consider.

5.5.4 Final driven evaluation

Finally which combination of driven meta heuristic is the best? HC driving an HC (HC-HC), HC driving a GA (HC-GA), or HC driving a PSO (HC-PSO)? If we take the same approach as the single search evaluation, we can compare the evaluation criteria against a baseline algorithm. We can compare the HC-GA and HC-PSO against the HC-HC algorithm. Table 5.10 shows the relative performance criteria of the HC-GA and HC-PSO against the HC-HC. The third column is the comparison of the HC-PSO against the HC-GA, that is the comparison of the relative improvements of the HC-PSO and HC-GA against the HC-HC.

Model	**GA**	**PSO**	**PSO/GA**
Simple ab	1.01	1.03	1.0286800304
Simple a	1.00	1.00	1
Brewery ab	0.84	0.85	1.0077942726
Brewery a	1.00	1.05	1.0523396711
E University	1.51	2.51	1.6634558589
Example system a	1.04	1.17	1.1270052479
Example system ab	1.45	1.43	0.9899284079
Large-t	1.00	1.00	1
mean	1.11	1.26	1.11

Table 5.10: Table to show driven HC divided by driven GA and PSO. With the relative difference between shown in the third column.

We see that the HC-PSO is 11% 'better' on average than the HC-GA is for finding on average the fittest candidate, so I conclude that the HC-PSO has to be the best driven combination for a driven search.

5.6 Conclusion

I started this project with the goal of finding a good meta heuristic for a model configuration search. I have tried three different kinds of meta heuristic, the Hill Climbing (HC) algorithm, the Genetic Algorithm (GA) and the Particle Swarm Optimisation (PSO) algorithm. After testing and evaluation I have found that the PSO algorithm is the most suitable algorithm for this search. I believe it is unnecessary for all three meta heuristics to be offered in the final capacity planning tool, and therefore I decided that the tool will only be using the PSO algorithm.

Further, I have found with some further experimentation that all three algorithms perform better when given settings appropriate for the search space. This is demonstrated by using chaining, or specifically a driven configuration. The evidence suggests that a PSO will perform 11% better when used in a driven configuration, driven by an HC.

However, naturally the driven configuration takes considerably longer to complete than the single search. A driven search is made of many single searches running in serial. The length of time the driven configuration takes to complete is dependent on the number of experiments and generations chosen for the driving Hill Climbing algorithm, but also on the number of experiments, the size of the candidate population, and the generation length of the driven meta heuristic.

The driven meta heuristic reduces the amount of input the modeller needs to enter, and evidence suggests it increases the chance of finding an optimal model, however it takes longer to return a solution. The single meta heuristic requires more modeller input, but runs quickly. Therefore I decided that the final capacity planning tool should offer both search methods to the user. I extended the output of the driven search to include the algorithm settings used on the driven meta heuristic. Therefore the driven search offers an optimal model configuration but also the optimal PSO algorithm settings, which can then be used to do faster single searches.

Chapter 6

User Interface

6.1 Introduction

The primary goal for this dissertation project has been to build a good search mechanism. The implementation of the user interface was a secondary concern and I have not had enough time to spend on interface design. My design philosophy for the user interface has been simple: provide 'a minimal but sufficient' interface to the user to demonstrate the primary goal; the search mechanism. The interface design and implementation turned out to be time consuming, especially as I had no experience in Eclipse plugin design previously and I have had to learn standard practices as I have been developing. Nevertheless I wanted to provide an interface that is stable and can be used in the future by users of PEPA. Furthermore with the guidance collected from the user evaluation some further development of the user interface has the potential to be much more supportive.

I had originally designed the user interface to allow the modeller to select any meta heuristic using any of three searches: single, pipeline or driven. This meant creating a complicated logical path for the Wizard. In the final capacity planning tool I elected to provide only two different types of search, the single PSO search and the driven PSO search, which reduced the Wizard drastically. If I were to start this project again, knowing what meta heuristics I was finally going to use, I would have more time to spend as I would only be designing two wizard paths instead of the nine originally developed.

I have spent some time refactoring the user interface during MPP2, which has been briefly mentioned in Chapter 4. In MPP1 the user interface code was untidy and unnecessarily hard coded in some places. This made introducing the new aspects of the project, such as chaining and the PSO, unnecessarily complicated. I elected to start using a Model-View-Controller framework instead, which enabled me to again separate out the tasks and roles of objects into proper classes.

The user interface can be broken into, and discussed as, two parts:

- The input: the capacity planning Wizard and its WizardPages.

- The output: the capacity planning Viewer.

6.1.1 Input

In order to run the capacity planning tool we need to enter some parameters; to input these parameters we use the capacity planning Wizard and its WizardPages. In Chapter 3 we saw the logic of the wizard pages (repeated here in Figure 6.1). In this section we step through screen shots of the WizardPages in the same order, explaining the role of the system.

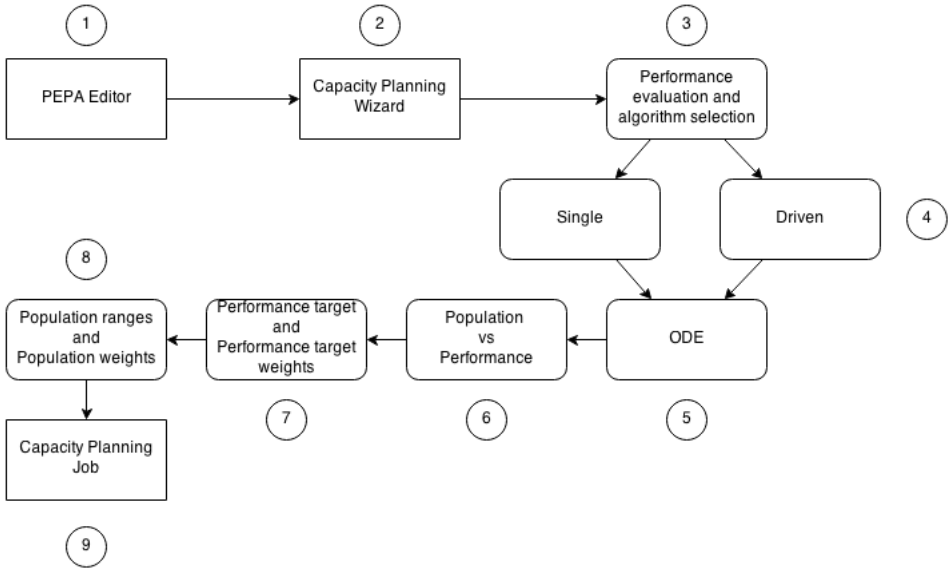


Figure 6.1: A high level picture of the capacity planning wizard.

Figures 6.2 to 6.6 correspond to the steps 1-8 in the high level picture presented in Figure 6.1. This logical path has been designed with the goal of minimising the number of forks (step 4 in the above figure), but also to provide a consistent and uncluttered interface for the user.

The user interface is used to define some settings for the fitness function defined in Subsection 3.1.2.4. I will be using the following list, and the letters **A-G** to demonstrate where the modeller can define these parameters:

- **Minimising performance target distance:** The modeller will have a desired performance target. Therefore there are the requirements to allow the modeller to define a performance target **[A]**, and an evaluation type **[B]** (of either throughput or average response time).
- **Minimising component population:** The modeller will want to reduce the number of resources used in a model, therefore they are looking for a model configuration with the lowest component population whilst still satisfying the above. The user interface allows the modeller to set a minimum and maximum population for components **[C]**. This enables the modeller to be able to set the search space to a specific domain.

- **Balancing performance targets and component population:** The modeller may consider the distance to the performance target to be more important to them than keeping a minimum component population, or the reverse, they may have restrictions on the number of components they can use and care less about the distance from the performance target. It is required therefore that there is some ability to be able to balance performance distance against component population **[D]**.
- **Component weighting:** The modeller may want to reduce, or penalise, the use of one component in a candidate model configuration. This introduces a notion of relative cost between components, there is a requirement to offer some method of assigning importance or cost to components **[E]**.
- **Performance target weighting:** The modeller may be interested in several performance targets, but they may consider some targets more important than others. There is a requirement here to allow a user to define some notion of importance to performance targets individually **[F]**.
- **Component choice:** The modeller may have some choice in what components to use, there maybe a choice of different types of component which have different costs and rates. There is a requirement to be able to search on models that offer a choice of resource **[G]**.

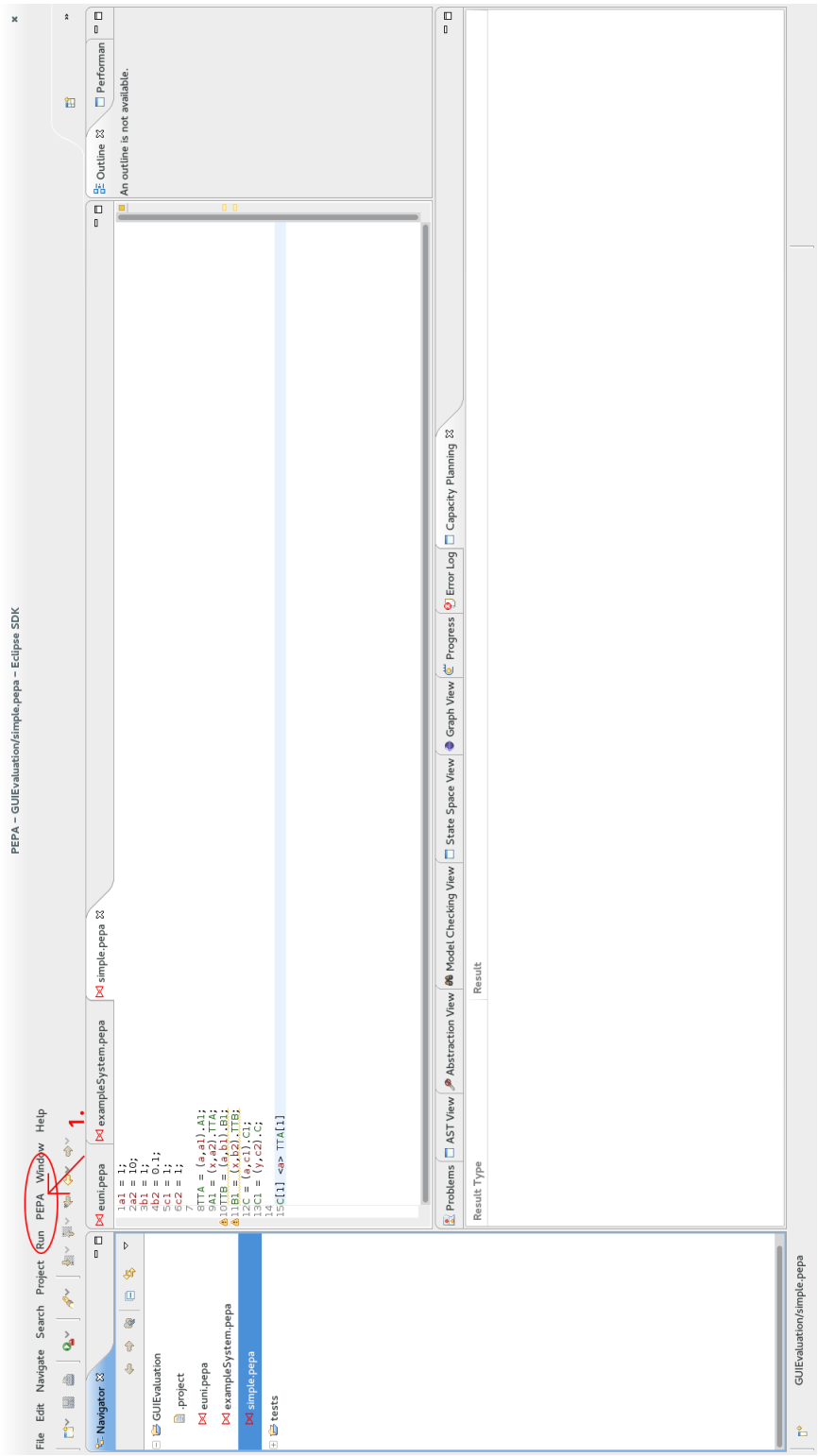
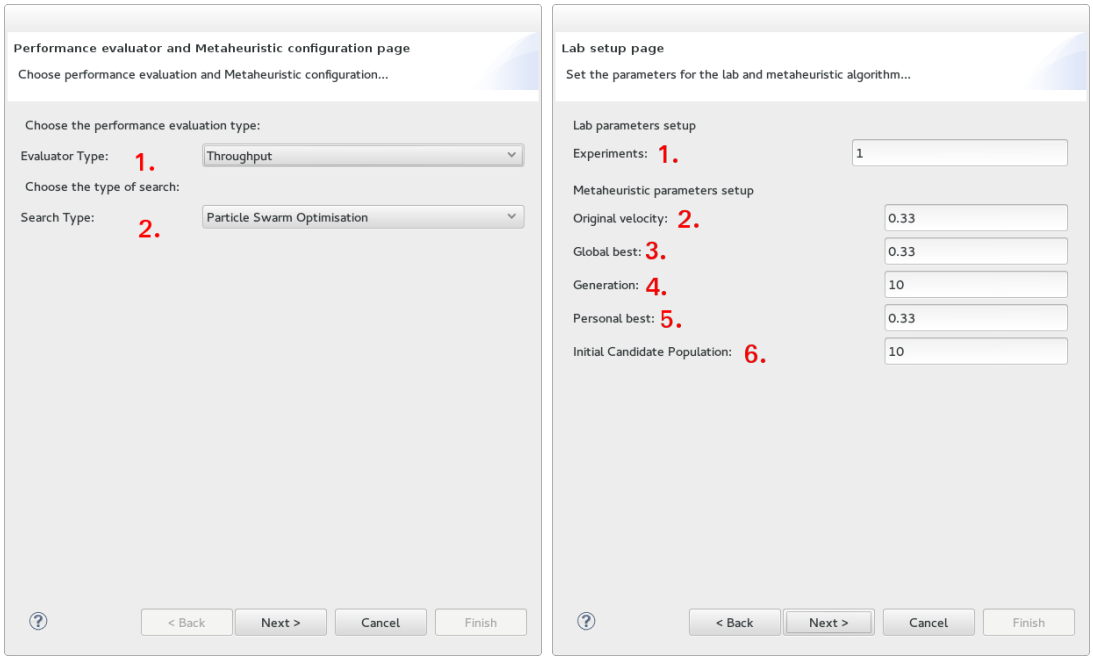


Figure 6.2: PEPA Eclipse menu.

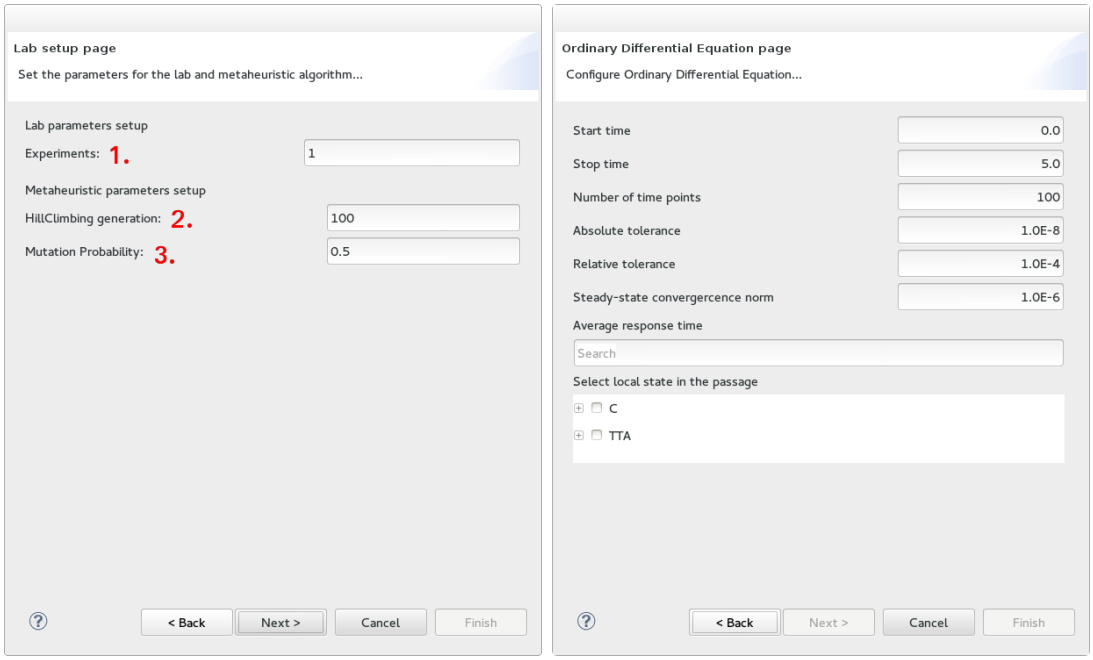
The first step is for the user to select the capacity planning tool found in the PEPA menu, under ”Scalable analysis”. This is a logical place for the capacity planning wizard as this is also where the other ODE based tools are found.



(a) The second step in the Wizard path logic. (b) The third step in the Wizard path logic.

Figure 6.3

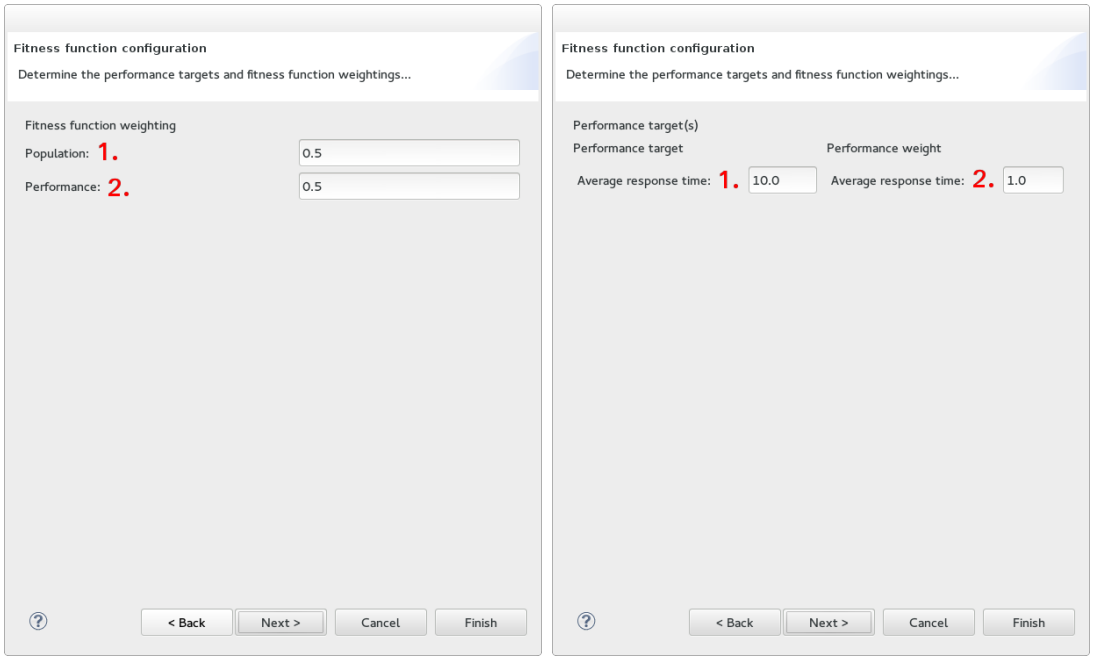
Figure 6.3 is comprised of two diagrams: (a) is the second step in the Wizard logic, here we see requirement **[B]** being satisfied: the modeller can determine evaluation type. The search type is used to determine whether to use the fast PSO single search, or slower driven PSO search. (b) is the third step in the Wizard logic if a single search has been selected, this page is for defining the meta heuristic settings as seen in Section 4.3.4.



(a) The third step in the Wizard path logic. (b) The fourth step in the Wizard path logic.

Figure 6.4

Figure 6.4 is comprised of two diagrams: (a) is the third step in the Wizard logic if a driven search has been selected, this page is for defining the meta heuristic settings. (b) this is the ODE setup Wizard page. I was able to reuse code already in PEPA in order to create this Wizard page. This page satisfies part of the requirement of **[A]** as this page is used to select the performance target.



(a) The fifth step in the Wizard path logic. (b) The sixth step in the Wizard path logic.

Figure 6.5

Figure 6.5 is comprised of two diagrams: (a) is the fifth step in the Wizard page logic. The user can define a balance between population and performance, which satisfies requirement **[D]** (b) this is sixth step of the Wizard page logic, here the user gives a value for the performance target which completes requirement **[A]**, and further the modeller can define performance weighting which satisfies requirement **[F]**.

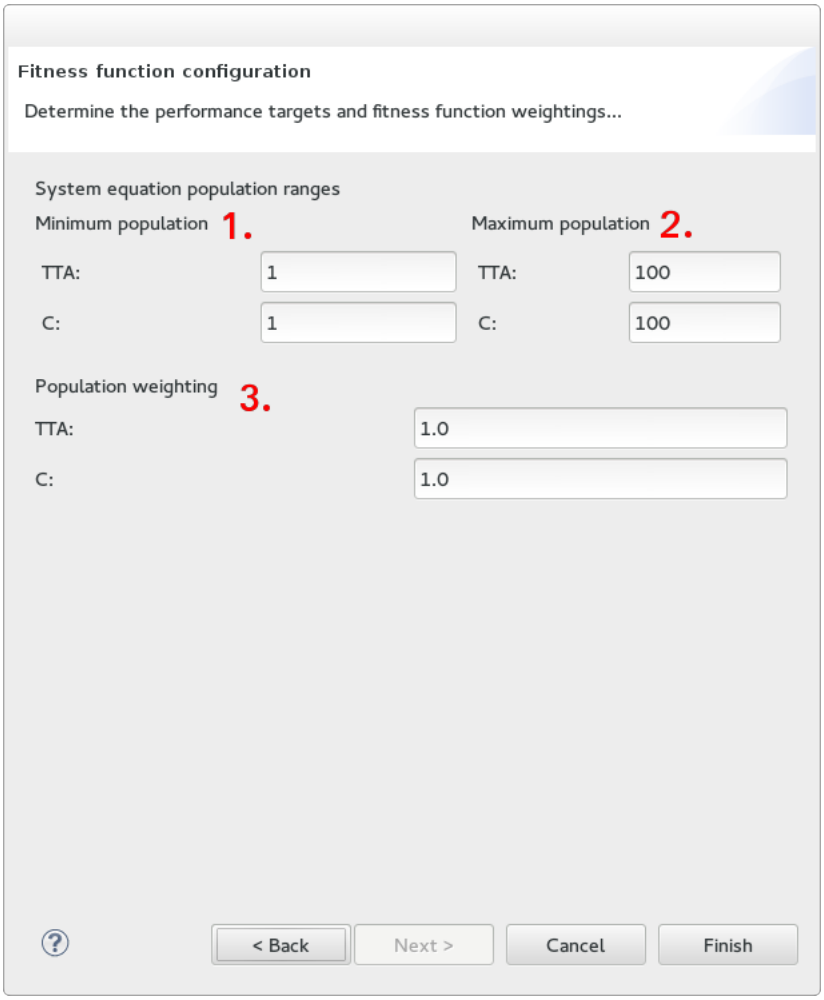


Figure 6.6: The last page in the Wizard logic.

Figure 6.6; The shows the last step in the logical path: The modeller can determine a minimum and maximum population, which satisfies requirement **[C]**, and because the minimum population can be set to zero, this page also satisfies requirement **[G]**. This page also enables the modeller to set component weighting, which satisfies the last requirement **[E]**.

6.1.2 Output

When the capacity planning job has completed, the RecordManager updates the capacity planning View with the results from the search. In the case of a single search (Figure 6.7) the view provides the top ten system equations found across all experiments. A driven search (Figure 6.8) also displays the top ten, but also returns the parameters of the best PSO search so that the modeller can run the fast single searches later.

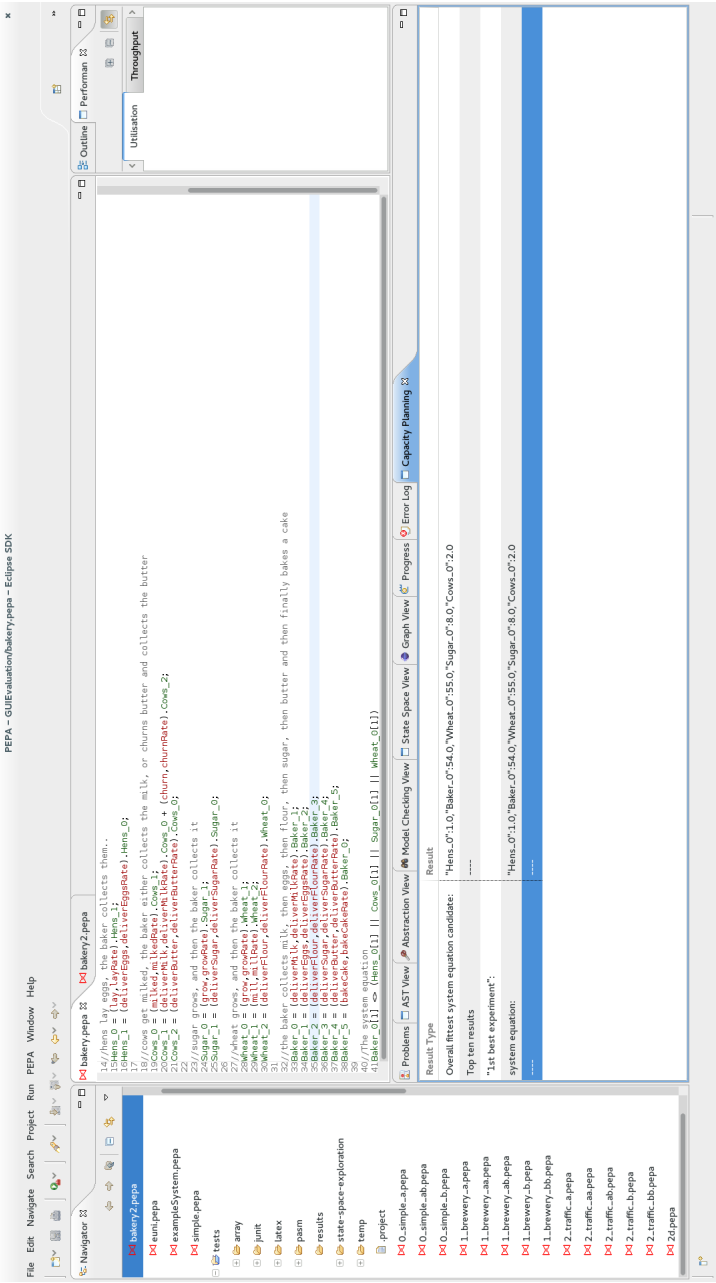


Figure 6.7: The output from a single search, the bottom 'capacity planning pane' shows the top ten candidate solutions

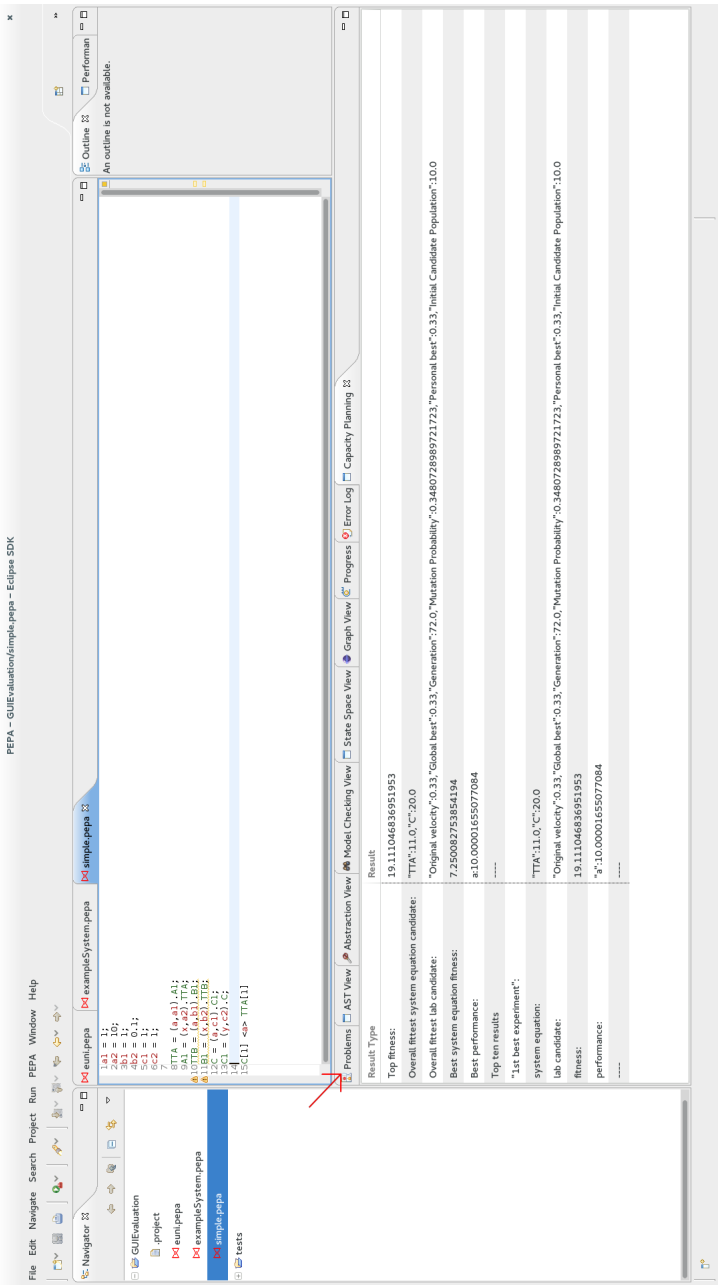


Figure 6.8: The output from a driven search, the bottom 'capacity planning pane' shows the top ten candidate solutions with the possible parameters for a single search

6.2 Evaluation

For the evaluation of the user interface I demonstrated the capacity planning tool to 8 PEPA modellers, and then asked them to answer some written questions and make notes on printouts of the same screen shots seen in the previous section. My primary goal was to find out if the tool was useful to them and if it was something they would like to use in the future. Secondly I asked if there were any improvements or recommendations the interface would need in order to be usable.

As I did not have enough time to write 'tool tips' or documentation inside the plugin, I decided that a walk-through demonstration on two models would be an effective way of guiding the modellers through the capabilities of the tool, rather than asking them to try the tool without any assistance. Therefore I would be on hand to explain what each of the settings did in lieu of having documentation.

In summary I wanted to know:

- If the tool was useful.
- If the tool was intuitive; if the PSO or the driven search was easy to understand.
- If the input was clear.
- If the modellers would be interested in only using either the single or driven searches or using both.
- Given a driven search, if the meta heuristic settings as output were useful.
- If the output was clear.
- If the number of returned items was enough.
- If there were any recommendations they could suggest to make the interface better

6.3 Results

The evaluation was extremely useful. On the whole I discovered that the tool would be useful and there were two evaluators who had models on which they would like to use the tool already.

The overall impression was the driven and single search would be more understandable if there was some documentation explaining how each of the settings affected the search, and that they would not be put-off from needing to understand the meta heuristic better.

On the whole they felt the input was not clear enough, and in some cases using sliders instead of entering numbers would make the experience more intuitive, particularly in the PSO settings setup (local best, original velocity, and global best), and with performance versus population.

Everyone who answered felt they would use the single search option, but felt the driven search was not transparent enough, and therefore my impression is I had not been able to explain the driven search clearly enough.

There was mixed content in the feedback about using the output meta heuristic settings. I believe that I had not been clear enough when explaining the driven search, because those who had understood the driven search were positive about using the output parameters in further single searches, whereas those who did not did not understand what they would be using them for.

Mostly the evaluators were asking for more output, with the possibility of having graphs and also the ability to save the results to a file. Further it was pointed out that they would like the previous settings to remain in memory so that they need not change the settingss again on the next run.

The evaluators said they would like to have more control over the number of output items.

There were many more recommendations and suggestions. These are listed verbatim below:

- The evaluators believe I should have separated the performance target selection from the ODE graph parameters.
- There was too much free space on each page.
- It would be useful to see the total population, and performance result in the output.
- It would be useful to have the output in columns.
- The order of the PSO settings is wrong, it would make sense to have the non-specific settings (generation, experiments and candidate population) first, and then the PSO specific settings.
- The component weights and population should be in columns.
- If there is a lot of variation in the results, then some feedback to reflect this in the output (perhaps with shading) would be useful.
- Some further explanation of what fitness means.
- Some more intuitive explanation of the PSO settings is required (perhaps using illustrations).
- Make it clearer that the driven search can provide details for a single search.
- Provide automatic feedback, the next single search uses the driven search parameters.
- Further explanation of the term 'Lab'.
- Make it clearer where a 0 can be used in a parameter.
- Provide the facility to create graphs from the results.

6.4 Conclusion

Overall the evaluators had said they thought the tool was useful, so my primary evaluation goal was a success. I had expected a lot of feedback from this session about the design of the interface, but I had not considered a lot of the comments made about the output.

I needed to have completed the implementation of MPP2 in order to be able to demonstrate the tool. Had I managed this in good time then I would have liked to have had this interface evaluation a lot earlier in the project. It is feasible to fulfil many of the requests from the evaluators with the programming that exists now, it would require extending the output mechanism to offer more analysis. This is, however, time consuming as this involves working with the Eclipse plugin development kit, and testing the interface.

If I also had more time I would also have liked to have had follow up sessions, where I could have used the feedback to create documentation. The idea of using diagrams to show how the single and driven search work I believe would have been very useful in teaching how to use the tool.

Chapter 7

Conclusion

The final version of the capacity planning tool offers two different kinds of search: a fast single search with the requirement that the modeller understands the heuristic, and the slower driven search with less user involvement required. I have chosen what I believe to be the best heuristics, or combination of heuristics, for both kinds of search based on empirical evidence.

The heuristic's optimisation method manages the candidate search, how the candidates communicate, how the candidates are created and mutated, but it is really the fitness function which defines the search. In Chapter 3 we described how the fitness function is constructed, and defined what a search for an optimal model configuration involves:

- **Minimising performance target distance:** The modeller will have a desired performance target (of either throughput or average response time). The modeller is looking for a model configuration where the distance between its performance measure and the performance target will be as small as possible.
- **Minimising component population:** The modeller will want to reduce the number of resources used in a model, therefore they are looking for a model configuration with the lowest component population whilst still satisfying the above.
- **Balancing performance targets and component population:** The modeller may consider the distance to the performance target to be more important to them than keeping a minimum component population, or the reverse, they may have restrictions on the number of components they can use and care less about the distance from the performance target. It is required therefore that there is some ability to be able to balance performance distance against component population.
- **Component weighting:** The modeller may want to reduce, or penalise, the use of one component in a candidate model configuration. This introduces a notion of relative cost between components, there is a requirement to offer some method of assigning importance or cost to components.
- **Performance target weighting:** The modeller may be interested in several performance targets, but they may consider some targets more important than others. There is a requirement here to allow a user to define some notion of impor-

tance to performance targets individually.

- **Component choice:** The modeller may have some choice in what components to use, there may be a choice of different types of component which have different costs and rates. There is a requirement to be able to search on models that offer a choice of resource.

MPP1 offered two different heuristics, with the choice of two different evaluation types (average response time and throughput), and only coarse tuning with regards to the weights above. In MPP2 I implemented the PSO, the framework for chaining, the driven search, more flexible searching, and the individual component weighting. The only part of this extension that I did not write was the ODE function and the ODE WizardPage, both of which were part of the existing Eclipse plugin tool and which I have treated as a black box.

7.1 Critical evaluation of implementation

Here I offer some critical evaluation of the project and possible future work as improvements for the capacity planning extension. My final version is stable, but it offers a minimal user interface, and might need some extension with regards to the system equation fitness function.

7.1.1 Improvements to the system equation fitness function

The system equation fitness function considers distance to performance target. This distance is an absolute value; a model with performance that is less than the target is treated the same as a model with a performance value higher than the target. In practise a modeller might consider an average response time less than the performance target, or a throughput higher than the performance target as beneficial. Currently the fitness function penalises any distance, so some future work into making this distance measure more dynamic, so that it recognises better performance would be beneficial.

The system does not behave very well if there is no possible performance target inside the population ranges given. That is if a modeller is looking for performance target x and this is impossible with the given range of resources, the optimisation method (regardless of which heuristic) will start to give erroneous results: results that have high populations or distant performance values. The plugin extension currently cannot warn the user of this effect, and this could lead to misleading results. This has the effect of the fitness function flipping between trying to reduce the resource count, and then trying to reduce the performance target distance and oscillating in a grey area. Future work could provide some further analysis of the results, and provide some indication to the modeller that their resource space is not large enough to return a stable result.

7.1.2 Programming

I had decided in MPP1 that the development of the heuristics was simple and interesting enough to not need to use some external library like the Evolutionary Computation and Genetic Programming System (ECJ). However at that point I had not appreciated the complexity of developing a driven meta heuristic, a stochastic search on a stochastic search, and further the difficulty in debugging. If I were to develop this again I would certainly use more supporting libraries.

7.2 Critical evaluation of user interface

The user interface has always been a secondary goal, with the development of the algorithms, and the evaluation of their efficacy as the primary goal. With that in mind, the user interface is stable, can be used for a search and does output useful results, but potentially it could be a lot more useful with the following improvements:

- Better code: I have attempted to keep as close as possible to a Model-View-Controller (MVC) framework, however there is a correct Eclipse plugin development philosophy which should be better exploited. That means future work would consist of redesigning the user interface as a 'proper' Eclipse plugin. Further, at the start of the project I was developing three different algorithms, with the potential for each to be used in different chaining configurations. Coming back to this project in the future, knowing that I would be using only a driven search (Hill Climbing on Particle Swarm Optimisation) and a single Particle Swarm Optimisation algorithm, I would definitely design the Wizard and output differently.
- Saving files: Currently the output is displayed to the Viewer at the bottom of the Eclipse application. It has become clear that it would also be useful to be able to save this output to a file for later reading.
- Results: The Viewer could be a lot more intelligent in what it returns as output; model configurations that are similar could be shaded similarly, total population and the performance evaluation could be displayed, offering the facility to graph fitness with total population and performance evaluation results would give the modeller a feel as to what is going on with the search space.

In summary I believe that the plugin extension has successfully fulfilled its requirements, it provides a stable experience and demonstrates a good capacity planning process for the modeller.

Bibliography

[1] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning / David E. Goldberg.* Reading, Mass. ; Wokingham : Addison-Wesley, [1989], 1989., 1989.

[2] Jane Hillston, Mirco Tribastone, and Stephen Gilmore. *Stochastic Process Algebras: From Individuals to Populations*. 2011.

[3] Jane Hillston, Mirco Tribastone, and Stephen Gilmore. Stochastic Process Algebras: From Individuals to Populations. *Computer Journal*, 55(7):866 – 881, 2012.

[4] John H. Holland. *Adaptation in natural and artificial systems [electronic resource] : an introductory analysis with applications to biology, control, and artificial intelligence / John H. Holland* . Complex adaptive systems. Cambridge, Mass. : MIT Press, 1992., 1992.

[5] Sertac Karaman, Tal Shima, and Emilio Frazzoli. A process algebra genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 16(4):489 – 503, 2012.

[6] S. Luke. Essentials of metaheuristics. Lulu, 2011.

[7] David Marco, David Cairns, and Carron Shankland. *Optimisation of Process Algebra Models Using Evolutionary Computation*. 2011.

[8] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization; an overview. *Swarm Intelligence*, (1):33, 2007.

[9] Mirco Tribastone, Adam Duguid, and Stephen Gilmore. The PEPA eclipse plugin. *Performance Evaluation Review*, 36(4):28, 2009.

Appendix A

The following in alphabetical order is the collection of eight models which were used in the evaluation of the single and driven meta heuristics:

A.1 Brewery model

$$
\begin{aligned}
f1 &= 1.2 \\
f2 &= 0.8 \\
f3 &= 0.8 \\
s2 &= 0.9 \\
w1 &= 0.4 \\
s1 &= 0.5 \\
p1 &= 0.8 \\
p2 &= 0.8
\end{aligned}
$$

$$
\begin{aligned}
Well &\stackrel{def}{=} (sink, w1).(store_w, s1).Well \\
Farm &\stackrel{def}{=} (sow, f1).(tend, f2).(reap, f3).(store_f, s2).Farm \\
Brewery &\stackrel{def}{=} (store_w, s1).BreweryW + (store_f, s2).BreweryF \\
BreweryW &\stackrel{def}{=} (store_f, s2).BreweryP \\
BreweryF &\stackrel{def}{=} (store_w, s1).BreweryP \\
BreweryP &\stackrel{def}{=} (produce, p1).Brewery \\
Brewery2 &\stackrel{def}{=} (store_w, s1).BreweryW2 + (store_f, s2).BreweryF2 \\
BreweryW2 &\stackrel{def}{=} (store_f, s2).BreweryP2 \\
BreweryF2 &\stackrel{def}{=} (store_w, s1).BreweryP2 \\
BreweryP2 &\stackrel{def}{=} (produce, p2).Brewery2
\end{aligned}
$$

$$
Well[1.0] \underset{*}{\bowtie} Brewery[1.0] \underset{*}{\bowtie} Farm[1.0]
$$

A.2 Brewery ab model

$$
\begin{aligned}
f1 &= 1.2 \\
f2 &= 0.8 \\
f3 &= 0.8 \\
s2 &= 0.9 \\
w1 &= 0.4 \\
s1 &= 0.5 \\
p1 &= 0.8 \\
p2 &= 80.0
\end{aligned}
$$

$$
\begin{aligned}
Well &\stackrel{def}{=} (sink, w1).(store_w, s1).Well \\
Farm &\stackrel{def}{=} (sow, f1).(tend, f2).(reap, f3).(store_f, s2).Farm \\
Brewery &\stackrel{def}{=} (store_w, s1).BreweryW + (store_f, s2).BreweryF \\
BreweryW &\stackrel{def}{=} (store_f, s2).BreweryP \\
BreweryF &\stackrel{def}{=} (store_w, s1).BreweryP \\
BreweryP &\stackrel{def}{=} (produce, p1).Brewery \\
Brewery2 &\stackrel{def}{=} (store_w, s1).BreweryW2 + (store_f, s2).BreweryF2 \\
BreweryW2 &\stackrel{def}{=} (store_f, s2).BreweryP2 \\
BreweryF2 &\stackrel{def}{=} (store_w, s1).BreweryP2 \\
BreweryP2 &\stackrel{def}{=} (produce, p2).Brewery2
\end{aligned}
$$

$$
Well[50.0] \underset{*}{\bowtie} Brewery[50.0] \parallel Brewery2[50.0] \underset{*}{\bowtie} Farm[50.0]
$$

A.3 E University model

$$
\begin{aligned}
nu &= 50.0 \\
r_cache &= 20.0 \\
r_int &= 3.0 \\
r_ext &= 4.0 \\
r_prep &= 5.0 \\
r_disp &= 8.0 \\
r_uni &= 5.0 \\
r_cur &= 4.0 \\
r_con &= 4.0 \\
r_reg &= 3.5 \\
r_read &= 5.0 \\
r_write &= 3.0 \\
r_read1 &= 5.0 \\
r_write1 &= 3.0 \\
r_read2 &= 10.0 \\
r_write2 &= 6.0 \\
r_lgc &= 3.0 \\
r_lgd &= 3.5 \\
r_think &= 0.08 \\
p &= 0.95 \\
n_s &= 1.0 \\
n_p &= 1.0 \\
n_d &= 1.0 \\
n_l &= 1.0 \\
n_ps &= 1.0 \\
n_pd &= 1.0
\end{aligned}
$$

$$
\begin{aligned}
StdThink &\stackrel{def}{=} (think, r_think).StdBrowse \\
StdBrowse &\stackrel{def}{=} (request_student_browse, nu).(reply_student_browse, nu).StdSelect \\
StdSelect &\stackrel{def}{=} (request_student_elect, nu).(reply_student_elect, nu).StdConfirm \\
StdConfirm &\stackrel{def}{=} (request_student_confirm, nu).(reply_student_confirm, nu).StdRegister \\
StdRegister &\stackrel{def}{=} (request_student_register, nu).(replystudentregister, nu).StdThink \\
StdThinkTag &\stackrel{def}{=} (think, r_think).StdBrowseTag \\
StdBrowseTag &\stackrel{def}{=} (request_student_browse, nu).(reply_student_browse, nu).StdSelectTag \\
StdSelectTag &\stackrel{def}{=} (request_student_elect, nu).(reply_student_elect, nu).StdConfirmTag \\
StdConfirmTag &\stackrel{def}{=} (request_student_confirm, nu).(reply_student_confirm, nu).StdRegisterTag
\end{aligned}
$$

$$
\begin{aligned}
StdRegisterTag &\stackrel{def}{=} (request_student_register, nu).(replystudentregister, nu).StdThinkTag \\
Portal &\stackrel{def}{=} (request_student_browse, nu).Browse + \\
&\quad (request_student_select, nu).Select + (request_student_confirm, nu). \\
&\quad Confirm + (request_student_register, nu).Register \\
Browse &\stackrel{def}{=} (acquire_ps, nu).Cache \\
Cache &\stackrel{def}{=} (cache, p * r_cache).Internal + (cache, 1.0 - p * r_cache).External \\
Internal &\stackrel{def}{=} (acquire_ps, nu).(internal, r_int).BrowseRep \\
External &\stackrel{def}{=} (request_external_read, nu).(reply_external_read, \\
&\quad nu).(acquire_ps, nu).(external, r_ext).BrowseRep \\
BrowseRep &\stackrel{def}{=} (reply_student_browse, nu).Portal \\
Select &\stackrel{def}{=} (acquire_ps, nu).(prepare, r_prep).ForkPrepare \\
ForkPrepare &\stackrel{def}{=} (fork, nu).JoinPrepare \\
JoinPrepare &\stackrel{def}{=} (join, nu).Display \\
Display &\stackrel{def}{=} (acquire_ps, nu).(display, r_disp).SelectRep \\
SelectRep &\stackrel{def}{=} (reply_student_select, nu).Portal \\
ValUni &\stackrel{def}{=} (fork, nu).(acquire_ps, nu).(validate_uni, r_uni).(join, nu).ValUni \\
ValCur &\stackrel{def}{=} (fork, nu).(acquire_ps, nu).(validate_cur, r_cur).(join, nu).ValCur \\
Confirm &\stackrel{def}{=} (acquire_ps, nu).(confirm, r_con).LogStudent \\
LogStudent &\stackrel{def}{=} (request_confirm_log, nu).(reply_confirm_log, nu).(reply_student_confirm, nu).Portal \\
Register &\stackrel{def}{=} (acquire_ps, nu).(register, r_reg).Store \\
Store &\stackrel{def}{=} (request_register_write, nu).(reply_register_write, nu).(replystudentregister, nu).Portal \\
Database &\stackrel{def}{=} (request_external_read, nu).Read + (request_register_write, nu).Write \\
Read &\stackrel{def}{=} (acquire_pd, nu).(read, r_read).ReadReply \\
ReadReply &\stackrel{def}{=} (reply_external_read, nu).Database \\
Write &\stackrel{def}{=} (acquire_pd, nu).(write, r_write).LogWrite \\
LogWrite &\stackrel{def}{=} (request_database_log, nu).(reply_database_log, nu).WriteReply \\
WriteReply &\stackrel{def}{=} (reply_register_write, nu).Database \\
Logger &\stackrel{def}{=} (request_confirm_log, nu).LogConfirm + (request_database_log, nu).LogDatabase \\
LogConfirm &\stackrel{def}{=} (acquire_pd, nu).(log_conf, r_lgc).ReplyConfirm \\
ReplyConfirm &\stackrel{def}{=} (reply_confirm_log, nu).Logger \\
LogDatabase &\stackrel{def}{=} (acquire_pd, nu).(log_db, r_lgd).ReplyDatabase \\
ReplyDatabase &\stackrel{def}{=} (reply_database_log, nu).Logger \\
PD_1 &\stackrel{def}{=} (acquire_pd, nu).PD_2 \\
PD_2 &\stackrel{def}{=} (read, r_read1).PD_1 + (write, r_write1).PD_1 + (log_conf, r_lgc).PD_1 + (log_db, r_lgd).PD_1 \\
PD2_1 &\stackrel{def}{=} (acquire_pd, nu).PD2_2 \\
PD2_2 &\stackrel{def}{=} (read, r_read2).PD2_1 + (write, r_write2).PD2_1 + (log_conf, r_lgc).PD2_1 + (log_db, r_lgd).PD \\
PS_1 &\stackrel{def}{=} (acquire_ps, nu).PS_2 \\
PS_2 &\stackrel{def}{=} (cache, r_cache).PS_1 + (internal, r_int).PS_1 + (external, r_ext).PS_1 + (prepare, r_prep).PS_1 +
\end{aligned}
$$

$$
StdThinkTag[600.0] \underset{*}{\bowtie} Portal[80.0] \underset{S_5}{\bowtie} ValUni[80.0] \underset{S_6}{\bowtie} ValCur[80.0]
$$

$$
\underset{S_7}{\bowtie} Database[80.0] \underset{S_8}{\bowtie} Logger[80.0] \underset{*}{\bowtie} PS_1[40.0] \parallel TTPD_1[40.0]
$$

$$
\begin{aligned}
[2.0ex] S_1 &= \{fork, join\} \\
S_2 &= \{fork, join\} \\
S_3 &= \{request_external_read, reply_external_read, request_register_write, reply_register_write\} \\
S_4 &= \{request_confirm_log, reply_confirm_log, request_database_log, reply_database_log\} \\
S_5 &= \{fork, join\} \\
S_6 &= \{fork, join\} \\
S_7 &= \{request_external_read, reply_external_read, request_register_write, reply_register_write\} \\
S_8 &= \{request_confirm_log, reply_confirm_log, request_database_log, reply_database_log\}
\end{aligned}
$$

A.4 E University ab model

$$
\begin{aligned}
nu &= 50.0 \\
r_cache &= 20.0 \\
r_int &= 3.0 \\
r_ext &= 4.0 \\
r_prep &= 5.0 \\
r_disp &= 8.0 \\
r_uni &= 5.0 \\
r_cur &= 4.0 \\
r_con &= 4.0 \\
r_reg &= 3.5 \\
r_read &= 5.0 \\
r_write &= 3.0 \\
r_read1 &= 5.0 \\
r_write1 &= 3.0 \\
r_read2 &= 10.0 \\
r_write2 &= 6.0 \\
r_lgc &= 3.0 \\
r_lgd &= 3.5 \\
r_think &= 0.08 \\
p &= 0.95 \\
n_s &= 1.0 \\
n_p &= 1.0 \\
n_d &= 1.0 \\
n_l &= 1.0 \\
n_ps &= 1.0 \\
n_pd &= 1.0
\end{aligned}
$$

$$
\begin{aligned}
[2.0ex]StdThink &\stackrel{def}{=} (think, r_think).StdBrowse\\
StdBrowse &\stackrel{def}{=} (request_student_browse, nu).(reply_student_browse, nu).StdSelect\\
StdSelect &\stackrel{def}{=} (request_student_select, nu).(reply_student_select, nu).StdConfirm\\
StdConfirm &\stackrel{def}{=} (request_student_confirm, nu).(reply_student_confirm, nu).StdRegister\\
StdRegister &\stackrel{def}{=} (request_student_register, nu).(reply_student_register, nu).StdThink\\
StdThinkTag &\stackrel{def}{=} (think, r_think).StdBrowseTag\\
StdBrowseTag &\stackrel{def}{=} (request_student_browse, nu).(reply_student_browse, nu).StdSelectTag\\
StdSelectTag &\stackrel{def}{=} (request_student_select, nu).(reply_student_select, nu).StdConfirmTag\\
StdConfirmTag &\stackrel{def}{=} (request_student_confirm, nu).(reply_student_confirm, nu).StdRegisterTag\\
StdRegisterTag &\stackrel{def}{=} (request_student_register, nu).(reply_student_register, nu).StdThinkTag\\
Portal &\stackrel{def}{=} (request_student_browse, nu).Browse+\\
&\quad (request_student_select, nu).Select + (request_student_confirm, nu).Confirm+\\
&\quad (request_student_register, nu).Register\\
Browse &\stackrel{def}{=} (acquire_ps, nu).Cache\\
Cache &\stackrel{def}{=} (cache, p * r_cache).Internal + (cache, 1.0 - p * r_cache).External\\
Internal &\stackrel{def}{=} (acquire_ps, nu).(internal, r_int).BrowseRep\\
External &\stackrel{def}{=} (request_external_read, nu).\\
&\quad (reply_external_read, nu).(acquire_ps, nu).\\
&\quad (external, r_ext).BrowseRep\\
BrowseRep &\stackrel{def}{=} (reply_student_browse, nu).Portal\\
Select &\stackrel{def}{=} (acquire_ps, nu).(prepare, r_prep).ForkPrepare\\
ForkPrepare &\stackrel{def}{=} (fork, nu).JoinPrepare\\
JoinPrepare &\stackrel{def}{=} (join, nu).Display\\
Display &\stackrel{def}{=} (acquire_ps, nu).(display, r_disp).SelectRep\\
SelectRep &\stackrel{def}{=} (reply_student_select, nu).Portal\\
ValUni &\stackrel{def}{=} (fork, nu).(acquire_ps, nu).(validate_uni, r_uni).(join, nu).ValUni\\
ValCur &\stackrel{def}{=} (fork, nu).(acquire_ps, nu).(validate_cur, r_cur).(join, nu).ValCur\\
Confirm &\stackrel{def}{=} (acquire_ps, nu).(confirm, r_con).LogStudent\\
LogStudent &\stackrel{def}{=} (request_confirm_log, nu).(reply_confirm_log, nu).(reply_student_confirm, nu).Portal\\
Register &\stackrel{def}{=} (acquire_ps, nu).(register, r_reg).Store\\
Store &\stackrel{def}{=} (request_register_write, nu).(reply_register_write, nu).(reply_student_register, nu).Portal\\
Database &\stackrel{def}{=} (request_external_read, nu).Read + (request_register_write, nu).Write
\end{aligned}
$$

$$
\begin{aligned}
Read &\stackrel{def}{=} (acquire_pd, nu).(read, r_read).ReadReply \\
ReadReply &\stackrel{def}{=} (reply_external_read, nu).Database \\
Write &\stackrel{def}{=} (acquire_pd, nu).(write, r_write).LogWrite \\
LogWrite &\stackrel{def}{=} (request_database_log, nu).(reply_database_log, nu).WriteReply \\
WriteReply &\stackrel{def}{=} (reply_register_write, nu).Database \\
Logger &\stackrel{def}{=} (request_confirm_log, nu).LogConfirm + (request_database_log, nu).LogDatabase \\
LogConfirm &\stackrel{def}{=} (acquire_pd, nu).(log_conf, r_lgc).ReplyConfirm \\
ReplyConfirm &\stackrel{def}{=} (reply_confirm_log, nu).Logger \\
LogDatabase &\stackrel{def}{=} (acquire_pd, nu).(log_db, r_lgd).ReplyDatabase \\
ReplyDatabase &\stackrel{def}{=} (reply_database_log, nu).Logger \\
PD_1 &\stackrel{def}{=} (acquire_pd, nu).PD_2 \\
PD_2 &\stackrel{def}{=} (read, r_read1).PD_1 + (write, r_write1).PD_1 + (log_conf, r_lgc).PD_1 + (log_db, r_lgd \\
PD2_1 &\stackrel{def}{=} (acquire_pd, nu).PD2_2 \\
PD2_2 &\stackrel{def}{=} (read, r_read2). \\
&\quad PD2_1 + (write, r_write2).PD2_1 + \\
&\quad (log_conf, r_lgc).PD2_1 + (log_db, r_lgd).PD2_1 \\
PS_1 &\stackrel{def}{=} (acquire_ps, nu).PS_2 \\
PS_2 &\stackrel{def}{=} (cache, r_cache).PS_1 + \\
&\quad (internal, r_int).PS_1 + (external, r_ext). \\
&\quad PS_1 + (prepare, r_prep).PS_1 + (display, r_disp). \\
&\quad PS_1 + (validate_uni, r_uni).PS_1 + \\
&\quad (validate_cur, r_cur).PS_1 + (confirm, r_con). \\
&\quad PS_1 + (register, r_reg).PS_1
\end{aligned}
$$

$$
StdThinkTag[600.0] \underset{*}{\bowtie} Portal[80.0] \underset{S_5}{\bowtie} ValUni[80.0] \underset{S_6}{\bowtie} ValCur[80.0]
$$

$$
\underset{S_7}{\bowtie} Database[80.0] \underset{S_8}{\bowtie} Logger[80.0] \underset{*}{\bowtie} PS_1[40.0] \parallel PD_1[40.0] \parallel PD2_1[40.0]
$$

$$
\begin{aligned}
[2.0ex]S_1 &= \{fork, join\} \\
S_2 &= \{fork, join\} \\
S_3 &= \{request_external_read, reply_external_read, request_register_write, reply_register_write\} \\
S_4 &= \{request_confirm_log, reply_confirm_log, request_database_log, reply_database_log\} \\
S_5 &= \{fork, join\} \\
S_6 &= \{fork, join\} \\
S_7 &= \{request_external_read, reply_external_read, request_register_write, reply_register_write\} \\
S_8 &= \{request_confirm_log, reply_confirm_log, request_database_log, reply_database_log\}
\end{aligned}
$$

A.5 Example System model

$$
\begin{aligned}
payRate &= 0.8 \\
receiveRate &= 5.0 \\
confirmRate &= 3.0 \\
authoriseRate &= 5.0 \\
readRate &= 10.0 \\
updateRate &= 1.0 \\
writeRate &= 5.0
\end{aligned}
$$

$$
\begin{aligned}
User_0 &\stackrel{def}{=} (pay, payRate).User_1 \\
User_1 &\stackrel{def}{=} (receive, receiveRate).User_0 \\
WebServer_0 &\stackrel{def}{=} (pay, payRate).WebServer_1 \\
WebServer_1 &\stackrel{def}{=} (confirm, confirmRate).WebServer_2 \\
WebServer_2 &\stackrel{def}{=} (authorise, authoriseRate).WebServer_3 \\
WebServer_3 &\stackrel{def}{=} (receive, receiveRate).WebServer_0 \\
TTApplicationServer_0 &\stackrel{def}{=} (confirm, confirmRate).ApplicationServer_1 \\
ApplicationServer_1 &\stackrel{def}{=} (read, readRate).ApplicationServer_2 \\
ApplicationServer_2 &\stackrel{def}{=} (update, updateRate).ApplicationServer_3 \\
ApplicationServer_3 &\stackrel{def}{=} (authorise, authoriseRate).ApplicationServer_4 \\
ApplicationServer_4 &\stackrel{def}{=} (write, writeRate).TTApplicationServer_0 \\
DatabaseServer_0 &\stackrel{def}{=} (read, readRate).DatabaseServer_0 + (write, writeRate).DatabaseServer_0
\end{aligned}
$$

$$
User_0[500.0] \underset{S_4}{\bowtie} WebServer_0[100.0] \underset{S_5}{\bowtie} TTApplicationServer_0[60.0] \underset{S_6}{\bowtie} DatabaseServer_0[8.0]
$$

$$
\begin{aligned}
S_1 &= \{pay, receive\} \\
S_2 &= \{confirm, authorise\} \\
S_3 &= \{read, write\} \\
S_4 &= \{pay, receive\} \\
S_5 &= \{confirm, authorise\} \\
S_6 &= \{read, write\}
\end{aligned}
$$

A.6 Example System ab model

$$
\begin{aligned}
payRate &= 0.8 \\
receiveRate &= 5.0 \\
confirmRate &= 3.0 \\
authoriseRate &= 5.0 \\
readRate &= 10.0 \\
updateRate1 &= 5.0 \\
updateRate2 &= 10.0 \\
writeRate &= 5.0
\end{aligned}
$$

$$
\begin{aligned}
User_0 &\stackrel{def}{=} (pay, payRate).User_1 \\
User_1 &\stackrel{def}{=} (receive, receiveRate).User_0 \\
WebServer_0 &\stackrel{def}{=} (pay, payRate).WebServer_1 \\
WebServer_1 &\stackrel{def}{=} (confirm, confirmRate).WebServer_2 \\
WebServer_2 &\stackrel{def}{=} (authorise, authoriseRate).WebServer_3 \\
WebServer_3 &\stackrel{def}{=} (receive, receiveRate).WebServer_0 \\
ApplicationServer_0 &\stackrel{def}{=} (confirm, confirmRate).ApplicationServer_1 \\
ApplicationServer_1 &\stackrel{def}{=} (read, readRate).ApplicationServer_2 \\
ApplicationServer_2 &\stackrel{def}{=} (update1, updateRate1).ApplicationServer_3 \\
ApplicationServer_3 &\stackrel{def}{=} (authorise, authoriseRate).ApplicationServer_4 \\
ApplicationServer_4 &\stackrel{def}{=} (write, writeRate).ApplicationServer_0 \\
ApplicationServer2_0 &\stackrel{def}{=} (confirm, confirmRate).ApplicationServer2_1 \\
ApplicationServer2_1 &\stackrel{def}{=} (read, readRate).ApplicationServer2_2 \\
ApplicationServer2_2 &\stackrel{def}{=} (update2, updateRate2).ApplicationServer2_3 \\
ApplicationServer2_3 &\stackrel{def}{=} (authorise, authoriseRate).ApplicationServer2_4 \\
ApplicationServer2_4 &\stackrel{def}{=} (write, writeRate).ApplicationServer2_0 \\
DatabaseServer_0 &\stackrel{def}{=} (read, readRate).DatabaseServer_0 + (write, writeRate).DatabaseServer_0
\end{aligned}
$$

$$
User_0[500.0] \underset{S_4}{\bowtie} WebServer_0[1.0] \underset{S_5}{\bowtie} ApplicationServer_0[1.0]
$$

$$
\| \; ApplicationServer2_0[1.0] \underset{S_6}{\bowtie} DatabaseServer_0[1.0]
$$

$$
\begin{aligned}
S_1 &= \{pay, receive\} \\
S_2 &= \{confirm, authorise\} \\
S_3 &= \{read, write\} \\
S_4 &= \{pay, receive\} \\
S_5 &= \{confirm, authorise\} \\
S_6 &= \{read, write\}
\end{aligned}
$$

A.7 Large-t model

$$
\begin{aligned}
[2.0ex]P1 &\stackrel{def}{=} (a,1.0).P2 + (b,1.0).P2 + (c,1.0).P2 + \\
&\quad (d,1.0).P2 + (e,1.0).P2 + (f,1.0).P2 + (g,1.0).P2 + \\
&\quad (h,1.0).P2 + (i,1.0).P2 + (j,1.0).P2 + (k,1.0).P2 + (l,1.0).P2 \\
P2 &\stackrel{def}{=} (a,1.0).P3 + (b,1.0).P3 + (c,1.0).P3 + \\
&\quad (d,1.0).P3 + (e,1.0).P3 + (f,1.0).P3 + (g,1.0).P3 + \\
&\quad (h,1.0).P3 + (i,1.0).P3 + (j,1.0).P3 + (k,1.0).P3 + (l,1.0).P3 \\
P3 &\stackrel{def}{=} (a,1.0).P4 + (b,1.0).P4 + (c,1.0).P4 + \\
&\quad (d,1.0).P4 + (e,1.0).P4 + (f,1.0).P4 + (g,1.0).P4 + \\
&\quad (h,1.0).P4 + (i,1.0).P4 + (j,1.0).P4 + (k,1.0).P4 + (l,1.0).P4 \\
P4 &\stackrel{def}{=} (a,1.0).P1 + (b,1.0).P1 + (c,1.0).P1 + \\
&\quad (d,1.0).P1 + (e,1.0).P1 + (f,1.0).P1 + (g,1.0).P1 + \\
&\quad (h,1.0).P1 + (i,1.0).P1 + (j,1.0).P1 + (k,1.0).P1 + (l,1.0).P1 \\
Q1 &\stackrel{def}{=} (a,1.0).Q2 + (b,1.0).Q2 + (c,1.0).Q2 + (d,1.0).Q2 + (e,1.0).Q2 + (f,1.0).Q2 \\
Q2 &\stackrel{def}{=} (a,1.0).Q3 + (b,1.0).Q3 + (c,1.0).Q3 + (d,1.0).Q3 + (e,1.0).Q3 + (f,1.0).Q3 \\
Q3 &\stackrel{def}{=} (a,1.0).Q4 + (b,1.0).Q4 + (c,1.0).Q4 + (d,1.0).Q4 + (e,1.0).Q4 + (f,1.0).Q4 \\
Q4 &\stackrel{def}{=} (a,1.0).Q1 + (b,1.0).Q1 + (c,1.0).Q1 + (d,1.0).Q1 + (e,1.0).Q1 + (f,1.0).Q1
\end{aligned}
$$

$$P1[4.0] \parallel Q1[4.0]$$

A.8 Simple model

$$
\begin{aligned}
a1 &= 1.0 \\
a2 &= 10.0 \\
b1 &= 1.0 \\
b2 &= 0.1 \\
c1 &= 1.0 \\
c2 &= 1.0
\end{aligned}
$$

$$
\begin{aligned}
TTA &\stackrel{def}{=} (a,a1).A1 \\
A1 &\stackrel{def}{=} (x,a2).TTA \\
TTB &\stackrel{def}{=} (a,b1).B1 \\
B1 &\stackrel{def}{=} (x,b2).TTB \\
C &\stackrel{def}{=} (a,c1).C1 \\
C1 &\stackrel{def}{=} (y,c2).C
\end{aligned}
$$

$$C[1.0] \underset{S_2}{\bowtie} TTA[1.0]$$

$$
\begin{aligned}
S_1 &= \{a\} \\
S_2 &= \{a\}
\end{aligned}
$$

A.9 Simple ab model

$$
\begin{aligned}
a1 &= 1.0 \\
a2 &= 10.0 \\
b1 &= 1.0 \\
b2 &= 0.1 \\
c1 &= 1.0 \\
c2 &= 1.0
\end{aligned}
$$

$$
\begin{aligned}
TTA &\stackrel{def}{=} (a, a1).A1 \\
A1 &\stackrel{def}{=} (x, a2).TTA \\
TTB &\stackrel{def}{=} (a, b1).B1 \\
B1 &\stackrel{def}{=} (x, b2).TTB \\
C &\stackrel{def}{=} (a, c1).C1 \\
C1 &\stackrel{def}{=} (y, c2).C
\end{aligned}
$$

$$C[1.0] \underset{S_2}{\bowtie} TTA[1.0] \parallel TTB[1.0]$$

$$
\begin{aligned}
S_1 &= \{a\} \\
S_2 &= \{a\}
\end{aligned}
$$

A.10 Traffic model

$$
\begin{aligned}
tn &= 1.0 \\
ts &= 1.0 \\
te1 &= 1.0 \\
tw &= 1.0 \\
rw &= 1.0 \\
rn &= 5.0 \\
rs &= 5.0 \\
re &= 1.0 \\
pg &= 10.0 \\
pr &= 10.0
\end{aligned}
$$

$$
\begin{aligned}
N0 &\stackrel{def}{=} (think, 1.0/tn).N1 \\
N1 &\stackrel{def}{=} (greenn, rn).N0 \\
S0 &\stackrel{def}{=} (think, 1.0/ts).S1 \\
S1 &\stackrel{def}{=} (greens, rs).S0 \\
E0 &\stackrel{def}{=} (think, 1.0/te1).E1 \\
E1 &\stackrel{def}{=} (rede, re).E0 \\
W0 &\stackrel{def}{=} (think, 1.0/tw).W1 \\
W1 &\stackrel{def}{=} (redw, rw).W0 \\
G &\stackrel{def}{=} (greenn, rn).G + (greens, rs).G + (change, 1.0/pg).R \\
R &\stackrel{def}{=} (rede, re).R + (redw, rw).R + (change, 1.0/pr).G
\end{aligned}
$$

$$N0[100.0] \parallel S0[1.0] \parallel E0[1.0] \parallel W0[1.0] \underset{S_2}{\bowtie} G$$

$$
\begin{aligned}
S_1 &= \{greenn, greens, rede, redw\} \\
S_2 &= \{greenn, greens, rede, redw\}
\end{aligned}
$$