NTNU
Innovation and Creativity

SAM Engine

Model-based Framework for Scalability Assessment

Anders Johan Holmefjord

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Peter Hughes, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

A method for assessing scalability of distributed systems (SAM) has been explored by several recent master projects at IDI. It is now proposed to develop a prototype supporting framework for SAM.
Desirable capabilities include:
- scaling paths with strict, non-uniform or differential scaling
- alternative definitions of operating point
- customisable user interface for simple "what-if" questions
- static and dynamic models which can reflect successive stages of different transactions
- assessing the impact of alternative frameworks for measurement experiments, with respect to the emulation of a large number of different users, and the corresponding synthetic data-base model.
- interface for updating and tracking model parameters derived from measurement.

In the autumn project, a preliminary study was conducted using the simulation language DESMO/J and measurement data from the KBM/KPM system (by courtesy of EDB Business Partner). It is planned to use this or similar data as a test case for the framework.The method of investigation will be to construct a prototype SAM engine which supports some of the above features. The first task will be to prioritise these features.

Assignment given: 20. January 2006
Supervisor: Peter Hughes, IDI

Abstract

Today's way of life includes increasing amounts of information, and therefore handling and processing of information. Almost everything you do involves some sort of a computer somewhere, and many businesses have implemented comprehensive computer systems into their corporative structure, to serve both employees and customers.

But if a new service is introduced to the users, or a new group of users are introduced to an existing service, how do you know if the performance will be satisfying?

To deal with such questions, a method called The Scalability Assessment Method (SAM) has been developed. The Scalability Assessment Method is a general procedure for evaluating the scalability of a system architecture. Other projects have applied SAM to real reference systems, and their results have shown that SAM is a method that can be trusted to give credible predictions.

Until recently, dedicated software tools that support the SAM method have been absent, and the researchers have been using i.a. spreadsheets in an ad hoc approach to the problems. Therefore, a SAM software package is in development.

The SAM Engine (SAMe) is a Java program developed in this project, with an intuitive user interface that is enabling a non-expert user to apply the method on a desired architecture. This report documents the development of the prototype SAM Engine (SAMe), and how the program supports the SAM method.

Keywords: Performance evaluation, scalability, simulation, Structure and Performance, SAM.

Preface

This Master's Thesis has been produced in the Performance Group at the Department of Computer and Information Science (IDI) at NTNU, spring 2006.

I would like to thank my supervisor, professor Peter Henry Hughes for invaluable help and interesting discussions surrounding the academic aspects of this project, and professional comments on my work. I would also like to thank my assistant supervisor, doctoral candidate Jacob Sverre Løvstad for all the hours of help with the practical and technical aspects of design and programming. Also Geir Bostad in the Performance Group deserve thanks for social and professional discussions.

I would like to thank Erik Rød and Erlend Mongstad for providing me with essential information through their Master Thesis [RM05].

Arild Røsdal of EDB Business Partner has been very cooperative with this project, and allowed me to build on Rød and Mongstad's results. Thank you.

Trondheim, June 16, 2006.

Anders Johan Holmefjord

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
APM	Action Port Model
CBMG	Customer Behaviour Management Graph
g	gain
GUI	Graphical User Interface
JFC	Java Foundation Classes
JNI	Java Native Interface
MVA	Mean Value Analysis
MVC	Model View Controller
OP	Operating Point
r	requirement
SAM	Scalability Assessment Method
SAMe	SAM Engine
SD	Service Demand
SIMSAM	SIMulation framework for the SAM method
SP	Structure and Performance
UML	Unified Modeling Language
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Title

SAM Engine: Model-based Framework for Scalability Assessment

1.2 Problem definition

A method for assessing scalability of distributed systems (SAM) has been explored by several recent master projects at IDI([FL03], [RM05]). It is now proposed to develop a prototype supporting framework for SAM. Desirable capabilities include:

- scaling paths with strict, non-uniform or differential scaling
- alternative definitions of operating point
- customisable user interface for simple "what-if" questions
- static and dynamic models which can reflect successive stages of different transactions
- assessing the impact of alternative frameworks for measurement experiments, with respect to the emulation of a large number of different users, and the corresponding synthetic database model
- interface for updating and tracking model parameters derived from measurement

In the autumn project[Hol05], a preliminary study was conducted using the simulation language DESMO/J[Des] and measurement data from the KBM/KPM system (by courtesy of EDB Business Partner). It is planned to use this or similar data as a test case for the framework. The method of investigation will be to construct a prototype SAM engine which supports some of the above features. The first task will be to prioritise these features.

1.3 Motivation

Today's way of life includes more and more information, and therefore handling and processing of information. This handling and processing is increasingly often handed over to computers. Almost everything you do involves some sort of a computer somewhere, and many businesses have implemented comprehensive computer systems into their corporative structure, to serve both employees and customers.

A trend these days is to make services available through internet, to increase availability and popularity. This would mean that a varying number of users will produce load on the computers at any time of day, and still expect constant, good quality of service, no matter how many other users there are in the system.

But if a new service is introduced to the users, or a new group of users are introduced to an existing service, how do you know if the performance will be satisfying? The need for capacity planning is obvious [MAD04]. To set up a user test for such a service would be infeasible, because then you could need a considerable amount of people sitting around and testing at the same time. A better solution is to use some kind of load generator and measure the performance of the system. However, both methods have a great disadvantage if you discover that the performance is not good enough. Because then you have the question: What kind of a system do I need to provide the users with the quality of service they demand? Of course, you can just buy a huge amount of computers that is guaranteed to meet the needs, but that is usually not an option because of budgets.

To deal with such questions, a method called The Scalability Assessment Method has been developed [Hug99, Hug05]. From now on called SAM, the method establishes a reference system, which is the system as it currently is configured. Then it explores different ways to increase the system capacity, and what impact the changes has on performance. In the end, you will be given a scaled-up system that will handle a specified number of users.

Until recently, dedicated software tools that support the SAM method have been absent, and the researchers have been using i.a. spreadsheets in an ad hoc approach to the problems. Therefore, a SAM software package is in development. The package will include several software modules, which each will support a part of the SAM method. My task is to develop a module that ties the other modules together, and guides the researcher towards a solution. A certain degree of automation is also essential, as the search for the answer could involve a lot of iterations.

In addition, it is too easy for software developers to make mistakes when thinking about performance issues. Even the simplest formulas can cause serious errors if not every detail is correct. And there is surely enough detail to get caught up in. Our experience is that many developers disregard the performance issues, because of the complexity. One of the goals for the SAM software package, is to abstract away the more complex issues and make it easy for a non-expert user to evaluate scenarios for simple what if-scenarios.

Companies that deliver performance-critical services will probably be the group that takes the most interest in this kind of software, but hopefully others will find it useful too.

Chapter 2

Project Plan

2.1 Original project plan

The original plan was made in the period from project start, to February 3rd. The blocks of time scheduled per task was estimated on the basis of experience from previous system engineering projects. The Gantt-diagram is shown in Figure B.1 in Appendix B.

Planning and preparing This phase consists of making a project plan, estimating time usage, creating a report template, and settle on a problem definition.

Requirements and design This phase is where requirements are worked out and agreed on. A detailed design is also made, according to the requirements.

Implementation This phase is where the program is actually made, with coding and continuous unit testing.

Testing The testing in this phase is on a high level, and is concluded with a user test.

Delivery Two preliminary drafts for a report are made to get feedback and read correction, before the final Master's Thesis is submitted June 16th.

2.2 Revised project plan April 26th

However, the phase "requirements and design" required a lot more time than estimated. This may be due to the fact that this is a research project, not just a plain development project like the projects mentioned in Section 2.1. As the requirements specification and the design document evolved, new requirements and design issues which had be taken into account appeared, and the phase needed to be extended. Because this project should develop a prototype, the design needed to be correct, so others could extend the prototype to a complete version. The Gantt-diagram is shown in Figure B.2 in Appendix B.

2.3 Revised project plan May 23rd

The testing brought out some new design issues, and some extra days was needed to implement the requested changes and test the changes again. That led to the delay of making a first draft, and subsequent tasks. The final Gantt-diagram is shown in Figure B.3 in Appendix B.

Chapter 3

Theory

This chapter will give a brief insight to the theory behind this project.

3.1 Scalability

The scalability of a system architecture is a property that can be thought of in several ways. This report will only focus on the scalability in terms of physical resource usage.

> *A system architecture is considered to be scalable over a particular set of requirements if the physical resource usage per unit of capacity remains roughly constant.* [BH04]

In this context, a dimension of scalability can be seen as a certain hardware capacity. According to [BH04, RM05], the three dimensions are:

Processing capacity:
Processing capacity is the rate of which a certain task is performed, defined for the whole system or subsystems or both. This can be both speed of a CPU, or speed of a disk, or other.

Storage capacity:
Storage capacity is the amount of storage at some level, e.g. the actual size in GB of a disk, or size of a cache at a CPU.

Connectivity:
Connectivity is the number of physical or logical connection points between components.

One might want to know how a system will perform if one add an extra node, add extra memory or insert a better CPU. [BH04] describes two different kinds of goals:

Speed-up
The objective with speed-up is to achieve a shorter execution time, when the amount of work remains the same. The size of the system increases, e.g. you double the CPU speed, but not the work. The boost is dependent on which part of the software that benefits most from the speed-up.

Scale-up
The objective with scale-up is to make your system cope with increased load, without increasing response times. The size of the system increases as well as the load, e.g. the number of users are doubled and you add an extra node.

There are two aspects of scaling in a scale-up process. The scaling within one dimension, and the scaling along all dimensions.([BH04])

Strict and differential scaling:
When you change the system by the same factor in all dimensions, the scaling is considered to be *strict*. If you scale by a different factor in different dimension, the scaling is considered to be *differential*.

Uniform and non-uniform scaling:
The scaling is *uniform* if all subsystems are changed by the same factor within one dimension. If they are changed by different factors, it is called *non-uniform* scaling.

In relation with simulation, [Hug05] describes some more concepts that may need an explanation:

Work is the actual operations, the services requested by the user.

Load is the frequency, the quantity, of invocations of the work.

Workload is then the product of work and load.

Operating Point is an appropriate workload for a node configuration regarding response time, utilisation and throughput

3.2 The Scalability Assessment Method

This description of the Scalability Assessment Method is based on [Hug99]. The Scalability Assessment Method (SAM) is a generic approach for analyzing the processing load scalability of a particular system. A general procedure description is summarized below.

1. **Define baseline for scalability analysis**
   First you need to determine the reference workload, a baseline node configuration and the operating point. Then define the work as a normalised, or possibly a worst-case if desirable, mix of the services available in the system.

2. **Model the scale baseline**
   Define a dynamic model relating processing capacity to properties of the baseline configuration, e.g. hardware specifications. Using a combination of standard performance evaluation techniques and measurement, you can parameterize the model. Different techniques are appropriate, depending on design stage and development method. In general, both a static and an dynamic model will be required. Perform controlled measurements where possible, both to determine capacity at the operating point, and also to get resource usage data per unit of throughput.

3. **Explore scalability**
   In this step we analyze how the system capacity is affected by increasing the relative system size under a chosen scale-invariant. This analysis will discover if the scale-invariant is maintained, and if so, the processing function at the operating point will scale identically. For this analysis, a static model is required. Decompose the service demand parameters, and map them onto the properties of the software and hardware components. A modification analysis is then applied to the baseline model parameters, to make a projection on capacity of the scaled-up system under the scale-invariants. This analysis has to capture any non-linear effect, not due to congestion at the hardware devices.

4. **Explore robustness**
   Use the model to carry out further modification analysis with varying workload. The robustness of the architecture is determined by the degree of variance allowed to the workload without losing scalability.

5. **Explore effects of technology change**
   Factor in the effect of expected technology change over the timescale. This is done by yet another modification analysis of the model parameters. Financial considerations may be taken, by considering only technology changes within the same cost range.

3.3 Scaling options

The SAM method has a hierarchical approach to analysing and scaling [Hug06a]. There are many paths for scaling the system, and the choices can be described as a scaling tree as showed in Figure 3.1. According to [BH04] there are usually two ways of increasing capacity at a certain level:

Replication is adding a number of copies of an existing component at some level in the system to the architecture, so that the workload is distributed evenly on the components giving the same services. E.g. buying an extra Web Server, or an extra disk.

Upgrade is achieved by a change of internal capacity at a component at some level, so that the component can give an improved service. E.g. increasing the clock speed of a CPU.

Figure 3.1 shows the four levels of the scaling tree, and the possible scaling paths. It is based on [BH04], but has been extended after some discussions within the Performance Group at IDI.

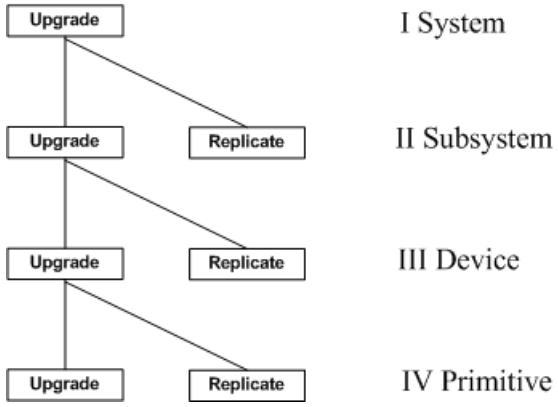

Figure 3.1: The upgrade tree of scaling options.

- Level I System: The system at the highest level cannot be replicated itself, the intension is always to upgrade the system.
- Level II Subsystem: A subsystem can be replicated by adding an identical subsystem at the same level, or upgraded by an inferior change, which is determined at the next level. Replication might not always be feasible either because of constraints at a higher level.

- Level III Devices: A device can be replicated by adding an identical device at the same level, or upgraded by an inferior change, which is determined at the next level. Replication might not always be feasible either because of constraints at a higher level. Replicating a CPU at this level is rather unusual, as that would typically be a cluster.
- Level IV Primitives: This is the bottom level, one cannot go any deeper than this. Upgrade will not always be feasible, depending on the modelling detail, because the component at this level could be a logic gate, which might not be upgradable. Replication might not always be feasible either because of constraints at a higher level, but is also an option here.

Chapter 4

Requirements

4.1 Use cases

This section will through textual use cases describe the most important interactions the user will be having with the application. The use cases will go through the most important scenarios for the user, which are the steps in the SAM procedure as described in Section 4.2. The procedure of assessing scalability consists of two phases, establishing the reference scale point, and exploring the scale path. These main functions of the application can be summarized in Figure 4.1. Each of these use cases are described in detail on the following pages.

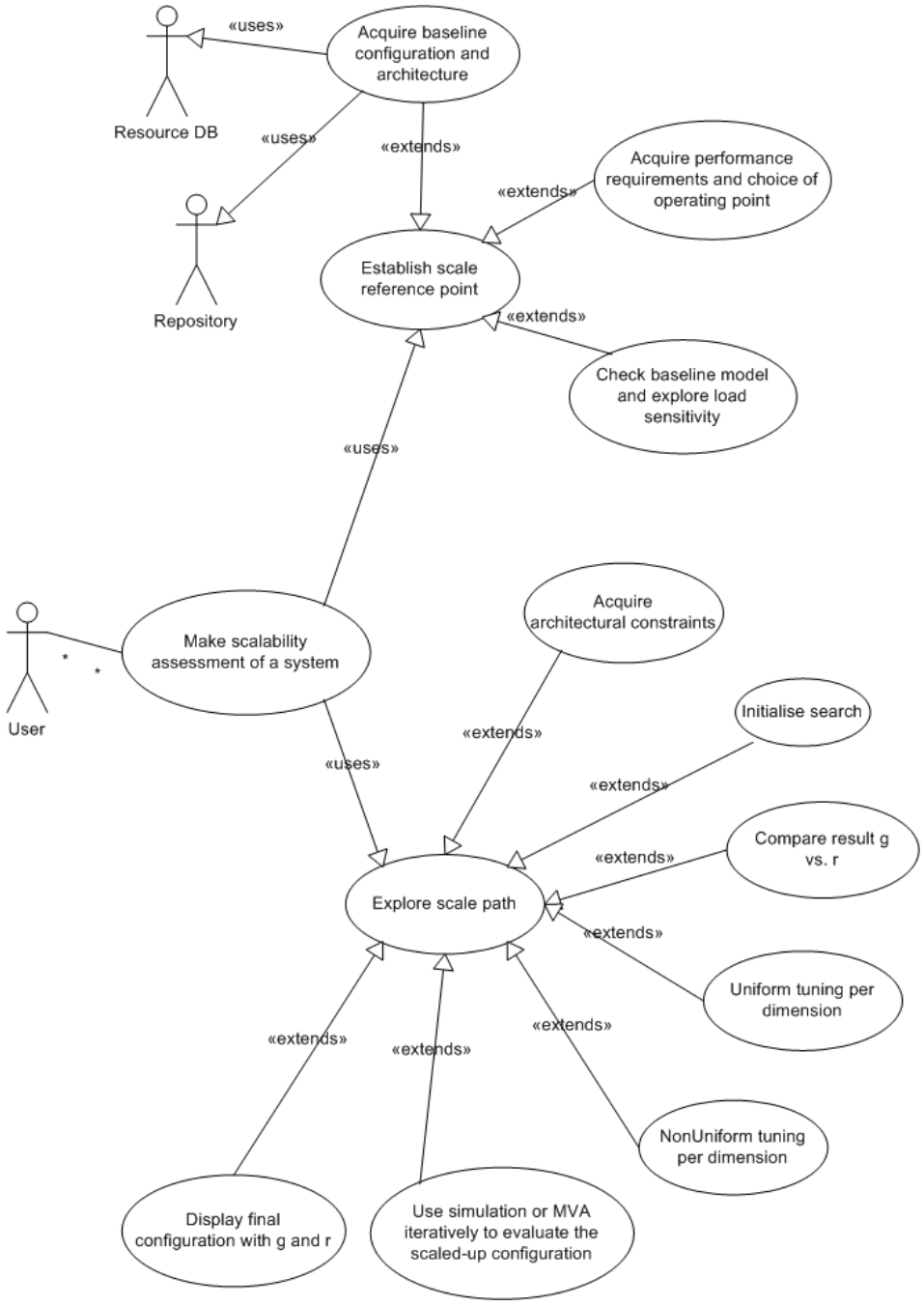

Figure 4.1: Use cases showing overall system usage

4.1.1 Acquire baseline.

This is the first step in the SAM procedure. The hierarchic structure of the system to be assessed is entered into the program, with subsystems and their hardware devices. The components and their relevant specifications are fetched from a database of real devices. As shown in Figure 4.1 there is only one role, so that anyone who wants can use the program. This use case is shown in Table 4.1.

Use case name	**Acquire baseline configuration and architecture.**
ID:	1
Basic course of events:	1. The subsystems are entered into a table. 2. The linking of the subsystems is determined by choosing children. 3. A list of the system parts is created and exported. 4. A list of real devices are fetched from the database. 5. The hardware devices of the subsystems are chosen from the list. 6. The linking of the hardware devices is determined by choosing children. 7. The relevant specifications for the hardware devices are registered. 8. The user presses ”OK”.
Alternative paths:	
Exception paths:	

Table 4.1: Acquire baseline configuration and architecture.

4.1.2 Acquire requirements.

This is the second step in the SAM procedure. Important performance and scale parameters are entered into the program. This use case is shown in Table 4.2.

Use case name	**Acquire performance requirements and choice of operating point.**
ID:	2
Basic course of events:	1. The vector components of the requirement r is entered into the program. 2. The type of operating point is chosen. 3. The numerical value of the operating point(OP) is entered into the program. 4. The user presses ”OK”.
Alternative paths:	
Exception paths:	

Table 4.2: Acquire performance requirements and choice of operating point.

4.1.3 Check baseline.

This is the third step in the SAM procedure. Important performance and scale parameters are entered into the program. This use case is shown in Table 4.3.

Use case name	**Check baseline model, and explore load sensitivity.**
ID:	3
Basic course of events:	1. Based on the information entered in use case 1, an SP model is made. 2. The SP tool calculates devolved work and saves them in a file. 3. SAMe reads the file. 4. Storage mappings are completed, and service demands are computed. 5. The user chooses to use simulation for calculating the load at the operating point (OP). 6. The system runs iterative calculations until the OP is reached. 7. The load and information about the load sensitivity is displayed to the user. 8. The user presses ”OK”.
Alternative paths:	5.a The user chooses to use an MVA algorithm module for calculating the load at the operating point (OP). 8.a The user changes the calculating method and runs new calculations. 8.b The user changes the OP and runs new calculations.
Exception paths:	

Table 4.3: Check baseline model, and explore load sensitivity.

4.1.4 Acquire constraints.

This is the fourth step in the SAM procedure. Architectural constraints, e.g. maximum number of disks at a server, are entered here. This use case is shown in Table 4.4.

Use case name	**Acquire architectural constraints.**
ID:	4
Basic course of events:	1. The list of subsystems is presented to the user. 2. The user chooses a subsystem. 3. The user enters an architectural constraint for the chosen subsystem. 4. Repeat 2 and 3 if desired. 5. The user presses ”OK”.
Alternative paths:	
Exception paths:	

Table 4.4: Acquire architectural constraints.

4.1.5 Initialise search.

This is the fifth step in the SAM procedure. The first search on the scaling path is initialised here. This use case is shown in Table 4.5.

Use case name	**Initialise search.**
ID:	5
Basic course of events:	1. The relative system size increase k is set to be equivalent to the required relative system capacity increase r. 2. New service demands are calculated based on the factor k. 3. The user chooses to use simulation for calculating the load at the operating point (OP). 4. The system runs iterative calculations until the OP is reached. 5. The gain g is computed by dividing the new load by the baseline load. 6. The gain is displayed in a plain way. 7. The user presses ”OK”.
Alternative paths:	2.a Load dependent effects exist, and the SP model is reevaluated. 3.a The user chooses to use an MVA algorithm module for calculating the load at the operating point (OP). 7.a The user changes the calculating method and runs new calculations.
Exception paths:	

Table 4.5: Initialise search.

4.1.6 Compare result g vs. r.

This is the sixth step in the SAM procedure. The gain g in performance resulted by the increase k in system size is compared to the required increase in performance r. This use case is shown in Table 4.6.

Use case name	**Compare result g vs. r.**
ID:	6
Basic course of events:	1. The previous and the new load, the gain g, and the requirement r is displayed to the user. 2. A visual representation of the difference between g and r is displayed. 3. The user chooses to adjust the scale factors by pressing ”Tune uniformly”.
Alternative paths:	3.a The user chooses to adjust the scale factors by pressing ”Tune non-uniformly”. 3.b The scale up is complete and the user presses ”Done”.
Exception paths:	

Table 4.6: Compare result g vs. r.

4.1.7 Uniform tuning.

This is the seventh step in the SAM procedure. The scale factors at the components are adjusted uniformly to better fulfill the requirement. This use case is shown in Table 4.7.

Use case name	**Uniform tuning per dimension.**
ID:	7
Basic course of events:	1. The program presents the system components with their present scale factors in the desired dimension. 2. The program displays the gain with the present scale factors in the desired dimension. 3. The user adjusts the scale factor in this dimension. 4. Repeat from 1 for each dimension. 5. The user presses ”Compute”. 6. Step 9 is executed.
Alternative paths:	
Exception paths:	

Table 4.7: Uniform tuning per dimension.

4.1.8 Non-uniform tuning.

This is the eight step in the SAM procedure. The scale factors at the components are adjusted non-uniformly to better fulfill the requirement. This use case is shown in Table 4.8.

Use case name	**Non-uniform tuning per dimension.**
ID:	8
Basic course of events:	1. The program presents the system components with their present scale factors in the desired dimension. 2. The program displays the gain with the present scale factors in the desired dimension. 3. The user chooses a component to scale. 4. The user adjusts the scale factor for the component in this dimension. 5. Repeat from 3 for all components that shall be scaled. 6. Repeat from 1 for each dimension. 7. The user presses ”Compute”. 8. Step 9 is executed.
Alternative paths:	
Exception paths:	

Table 4.8: Non-uniform tuning per dimension.

4.1.9 Use simulation or MVA.

This is the ninth step in the SAM procedure. A simulation model or an MVA algorithm is used to calculate the load at the operating point(OP). The calculations will be iterative searches with different load. An important issue is to minimize the number of iterations. This use case is shown in Table 4.9.

Use case name	**Use simulation or MVA algorithm iteratively to evaluate the new configuration.**
ID:	9
Basic course of events:	1. The user chooses to use simulation for calculating the load at the operating point (OP). 2. The system runs iterative calculations until the OP is reached. 3. The gain g is computed by dividing the new load by the baseline load. 4. The gain is displayed in a plain way. 5. The user presses ”OK”.
Alternative paths:	1.a Load dependent effects exist, and the SP model is reevaluated. 1.b The user chooses to use an MVA algorithm module for calculating the load at the operating point (OP). 5.a The user changes the calculating method and runs new calculations.
Exception paths:	

Table 4.9: Use simulation or MVA algorithm iteratively to evaluate the new configuration.

4.1.10 Display final configuration.

This is the tenth step in the SAM procedure. The final results of the scale-up is displayed. This use case is shown in Table 4.10.

Use case name	**Display final configuration with g and r.**
ID:	10
Basic course of events:	1. The final system configuration is displayed. 2. The final gain g is displayed. 3. The requirement r is displayed. 4. Any differences between g and r are displayed graphically.
Alternative paths:	
Exception paths:	

Table 4.10: Display final configuration with g and r.

4.2 The SAMe procedure

The SAM procedure below is based on [Hug99] and [Hug06b], but has been developed to more detail after thorough research and discussion in the Performance Group at NTNU.

1. Acquire baseline configuration and architecture.
2. Acquire performance requirements and choice of operating point.
3. Check baseline model, and explore load sensitivity.
4. Acquire architectural constraints.
5. Initialise search.
6. Compare result g vs. r
7. Uniform tuning per dimension.
8. Non-uniform tuning per dimension.
9. Use simulation or MVA algorithm iteratively to evaluate the new configuration.
10. Display final configuration with g and r.

This is a general design requirement, and serves as a basis for the rest of the requirements. The steps are described in more detail below.

1. Acquire baseline configuration and architecture.
   (a) The baseline model and configuration is entered into the GUI of SAMe. First enter the configuration at level I, then at level II, and then at level III. The model could be read from a file created by SP Light.
   (b) Real components are chosen from the resource database, and their relevant specifications are registered by SAMe.
2. Acquire performance requirements and choice of operating point.
   (a) The requirement r (a vector) is entered into the GUI of SAMe.
   (b) The desired operating point is entered into the GUI of SAMe. This could be a level of utilisation at a bottleneck device.
3. Check baseline model, and explore load sensitivity.
   (a) Service demands are calculated.
   (b) SAMe runs simulations until the operating point is reached. Or the load could be calculated using an MVA algorithm ([MAD94]).
   (c) The baseline load is registered by SAMe.
   (d) SAMe displays the results to the user.
4. Acquire architectural constraints.
   (a) Register possible architectural constraints in the system. These constraints will be checked every time an upgrade or replication is made.
5. Initialise search.
   (a) k is set to r

   (b) If load dependent effects are present, the SP model is reevaluated, and new service demands are calculated.
   (c) SIMSAM simulations are executed with the new configuration until the operating point is reached. Or the load could be calculated using an MVA algorithm.
   (d) The new load is registered by SAMe.
   (e) The gain g is computed by dividing the new load by the baseline load.

6. Compare result g vs. r
   (a) The gain g is compared to the requirement r. The difference is displayed clearly for the user.
   (b) If the gain is satisfactory, go to step 10, else proceed to step 7.

7. Uniform tuning per dimension.
   (a) The k is adjusted uniformly in each of the three dimensions, processing, storage and connectivity, so that the gain g will more likely fulfill the requirement r.
   (b) The adjustment could be upgrade or replication of components.

8. Non-uniform tuning per dimension.
   (a) The k is adjusted non-uniformly in each of the three dimensions, processing, storage and connectivity, so that the gain g will more likely not exceed the requirement r more than desired for each subsystem.
   (b) The adjustment could be upgrading or downsizing, or replicating or removing of components.
   (c) This step involves that real components are chosen from the resource database, and their relevant specifications are registered by SAMe.

9. Use simulation or an MVA algorithm iteratively to evaluate the new configuration.
   (a) If load dependent effects exists or the workmix has changed, the SP model is reevaluated, and new service demands are calculated. Else skip this step.
   (b) Decide whether to use simulation or the MVA algorithm. If MVA, go to step 9d.
   (c) SIMSAM reads the file with the resources and their service demands, and simulations are executed until the operating point for the current configuration is reached. Go to step 9e.
   (d) The MVA algorithm is used to calculate the load at the operating point for the current configuration.
   (e) The new load is registered by SAMe.
   (f) The gain g is computed by dividing the new load by the previous load.
   (g) Return to step 6.

10. Display final configuration with g and r.
   (a) The final configuration of the system is displayed.
   (b) The gain g and the requirement r are displayed so that the differences are clear to the user.

4.3 Functional requirements

The functional requirements will define what functionality is demanded of the SAM engine (SAMe) program, and are based on [Hug06b]. The sections are divided in logically different components of the application:

- Requirements for SAMe
- Requirements for the GUI
- Requirements for subsystems
- Requirements for primitives
- Requirements for computations

4.3.1 Requirements for SAMe

This section defines the requirements for the program as a whole, and not a specific part of the software. They describe what top level functionality that must be supported by SAMe.

F1 The SAMe program must support the SAM procedure as described in Section 4.2.

F2 SAMe must have a connection to a database.

F3 SAMe must have a connection to an SP tool.

F4 The user must be able to save a scalability assessment study.

F5 The user must be able to load a scalability assessment study.

F6 The user must be able to reset a scalability assessment study back to baseline.

F7 SAMe must have an interface to the subsystem repository.

F8 SAMe must be able to import subsystems from the repository.

F9 The system modeled in SAMe must have a tree structure.

F10 SAMe must contain a list of all the subsystems in the system.

F11 SAMe must contain a list of all the primitives in the system.

F12 SAMe must be able to export a list of all the components in the system.

4.3.2 Requirements for the GUI

This section defines the requirements for the graphical user interface. They describe how the GUI shall show information, and support user interaction.

F13 The user must be able to enter the three vector components of the requirement r.

F14 The user must be able to enter the numerical value of the operating point(OP).

F15 The user must be able to choose the type of the OP.

F16 The user must be able to choose the method for calculating the load at the OP.

F17 The user must be able to change the type and value of the OP.

F18 The user must be able to choose whether to upgrade or replicate a system component.

F19 SAMe must display a graph describing the load sensitivity in a given interval around the OP.

F20 SAMe must display a graph comparing the relative system capacity increase g, and the requirement.

F21 After comparing g and r, the user must be able to choose to continue the scalability assessment, or to end it.

F22 When the scalability assessment is finished, SAMe must display the final configuration of the system, and a graphical comparison between the final gain g and the requirement r.

4.3.3 Requirements for subsystems

This section defines the requirements for the part of the solution called a subsystem. A subsystem is a node in the system tree, that is allowed to have children. It could model e.g. a server.

F23 The subsystems must have a identification number(ID).

F24 The subsystems must have a name.

F25 The subsystems must have a list of child subsystems.

F26 The subsystems must have a list of primitives.

F27 The subsystems must be able to have architectural constraints.

F28 The user must be able to choose a subsystem and enter any architectural constraints.

F29 It must be possible to replicate subsystems.

F30 It must be possible to change the specifications of the subsystems after they have been created in the system.

4.3.4 Requirements for primitives

This section defines the requirements for the part of the solution called a primitive. A primitive is a leaf node in the system tree, that is not allowed to have children. It could model e.g. a cpu.

F31 The primitives must have a identification number(ID).

F32 The primitives must have a name.

F33 The primitives must have a description.

F34 The primitives must have devolved work.

F35 The primitives must have a speed.

F36 The primitives must have a storage size.

F37 The primitives must have a connectivity.

F38 The primitives must get their data from a database.

F39 It must be possible to replicate primitives.

F40 Primitives must have a replication factor property.

F41 It must be possible to upgrade primitives.

F42 The primitives must have a scale factor property for each dimension.

F43 It must be possible to change the specifications of the primitives after they have been created in the system.

F44 SAMe must read devolved work from the SP tool, and calculate service demands for the primitives.

4.3.5 Requirements for computations

This section defines the requirements for the computations made by SAMe. The computations will be supported by a simulation framework, or other solutions.

F45 SAMe must be able to use a simulation framework to calculate the gain.

F46 The user must be able to uniformly adjust the scale factors for the subsystems in the three dimensions: processing, storage and connectivity.

F47 The user must be able to non-uniformly adjust the scale factors for the primitives in the three dimensions: processing, storage and connectivity.

F48 While in the process of adjusting the scale factors for the system, the user must at any time be able to compute the gain g and compare it with the requirement.

F49 SAMe must have the possibility to use a heuristic that automatically adjusts the scale factors to meet the requirement in an optimal way.

F50 SAMe must write to a log everything an action is performed.

F51 A log record must consist of which action was performed, and which parameters were used in the action.

F52 SAMe must save every scaling action that is made, so that it will be possible to go back to an earlier choice and try a different scaling path.

F53 SAMe must be able to compute the gain, which is the relative system capacity increase, in each of the dimensions.

F54 The operating point (OP) must be either utilisation, response time or Kleinrock saturation point.

Chapter 5

Design

A correct and thorough design is imperative for a project in system engineering. Together with a complementary requirements specification (Chapter 4), a thought-through design builds a necessary foundation for the implementation of the program. The design in this project is especially important, because the purpose is that others will continue the development of the SAM engine (SAMe) after this project is finished.

5.1 Technical choices

Here the technical choices of the project are explained and justified. In any project, time is a critical factor and as far as possible known technical solutions should be used, instead of using critical time learning a new programming tool.

5.1.1 Java

In the beginning of the project, it was decided to create a new software module from scratch, and the chosen programming language to accomplish this was Java 5.0. It is important that users notice this, because a lot of features in version 5.0 is not compatible with earlier versions of Java. However, the improvements in 5.0 are good enough to justify the choice.

This section tries to justify the choice of programming language. There exist several reasons why Java was selected before C++, MS.NET or any other language, but the three most important are as follows:

- Existing knowledge and experience with Java
- Java has classes for designing graphical user interfaces (GUI)
- Platform independence

Existing knowledge of the programming language

The experience level with Java as an implementation tool was clearly higher than other languages. This fact had a very high weight when programming language was selected. Implementing in another language would not have been an impossible task, but it would cost a lot of time to learn it, and would really be a waste of time in this project. There was no demands concerning the selection of programming tools as long as it did not concern functionality, and there is nothing within this project that can not be done with Java.

GUI

A considerable amount of the planned application will be GUI and Java has a extensive API for designing what is needed of graphical user interfaces, through Java Foundation Classes (JFC). Also, building GUIs with Java enables a lot of reuse which is time-saving and an important part of object-oriented design.

Platform independence

Java programs can be run on any platform, so there will be no special platform demand for the user. This is important, because there is no limited or known group of users for SAMe, and therefore no assumptions can be made about what platform SAMe will be run on in the future. The best solution is to prepare for everything.

5.1.2 Design Pattern - MVC paradigm

It is highly desirable to increase the ease of development as well as ensuring a high degree of extensibility when developing the software of this project. Patterns are a solution to well-known programming problems. The Model-View-Controller[MVC] paradigm separates the components of the program into three different categories: Model, view and controller, as seen Figure 5.1. By doing this it increases the clarity of design and also makes it easier to change modules without having to change other modules.

For this reason the Model-View-Controller paradigm was selected to build upon when constructing the application.

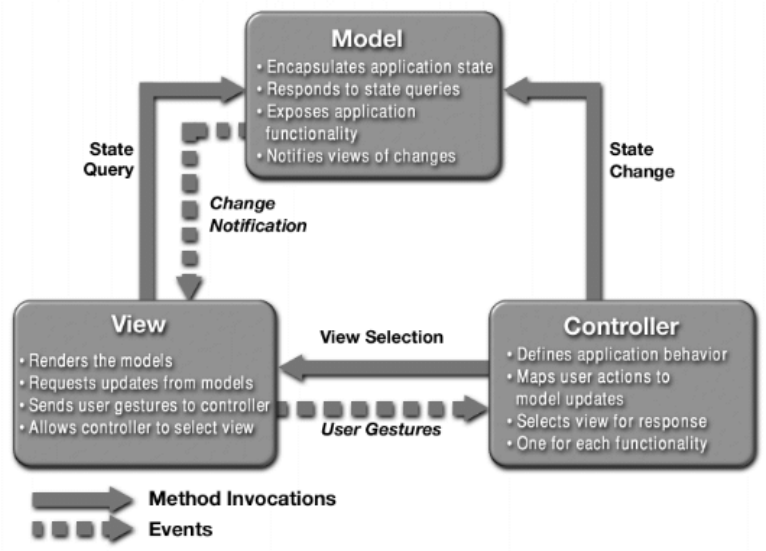

Figure 5.1: Basic Model-View-Controller relationship, from [MVC]

MVC components overview The MVC architecture consists of the model, the view and the controller.

Model The model is the core of the application that maintains the state and data that the application represents. When significant changes occur in the model it notifies observers (Typically a view) that updates all the changes.

If one needs to model two groups of unrelated data and functionality, two separate models are created.

View The view is the user interface which displays information about the model to the user. Any object that needs information about the model needs to be a registered view with the model.

A viewport typically has a one to one correspondence with a display surface and knows how to render to it. A viewport attaches to a model and renders its contents to the display surface. In addition when the model changes the view port automatically redraws the affected part of the image to reflect those changes.

Controller The controller is the user interface presented to the user to manipulate the applications data. A controller accepts input from the user and instructs the model and view port to perform actions based on that input. In effect, the controller is responsible for mapping end-user action to application response. For example, if the user clicks the mouse button or chooses a menu item, the controller is responsible for determining how the application should respond.

Summary of MVC Figure 5.1 shows the basic lines of communication among the model, viewport and controller. In this figure, the model points to the viewport, which allows it to send the viewport notifications of change. Of course, the model's viewport pointer is only a base class pointer; the model should know nothing about the kind of viewport that observe it. By contrast, the viewport knows exactly what kind of model it observes. The viewport also has a strongly-typed pointer to the model, allowing it to call any of the model's functions. In addition, the viewport also has a pointer to the controller, but it should not call functions in the controller aside from those defined in the base class. The reason is you may want to swap out one controller for another, so you'll need to keep the dependencies minimal. The controller has pointers to both the model and the viewport and knows the type of both. Since the controller defines the behavior of the modules, it must know the type of both the model and the view port in order to translate user input into application response.

Advantages of the MVC paradigm Here some of the advantages of the MVC architecture are explained.

- Clarity of design

The public methods in the model stand as an API for all the commands available to manipulate its data and state. By looking at the model's public method list, it is easy to understand how to control the model's behavior. When designing the application, this trait makes the entire program easier to implement and maintain.

- Multiple views

The application can display the state of the model in a variety of ways, and create/design them in a scalable, modular way. As an example one could think of a html page presenting a model as one view, and a Java SWING application based on the same data is another view.

- Efficient modularity

The components can be swapped in and out as the user or programmer desires, also the model. Changes to one aspect of the program aren't coupled to other aspects, eliminating many nasty debugging situations. Also, development of the various components can progress in parallel once the interface between the components is clearly defined.

Java and MVC Java GUI components implements some sort of MVC. For example a JList component graphically represents the model (ListModel) to the user as data in the JList. The Controller takes care of mouse and keyboard events generated when the user selects an element in the list.

The Java programming language provides support for the Model-View-Controller architecture with two classes.

- Observer

Any object that wishes to be notified when the state of another object changes.

- Observable

Any object whose state may be of interest, and in whom another object may register an interest.

These two classes can be used to implement much more than just the Model-View-Controller architecture. They are suitable for any system wherein objects need to be automatically notified of changes that occur in other objects. The model often is a subtype of Observable and the view implements Observer. These two classes handle the automatic notification function of the Model-View-Controller architecture. They provide the mechanism by which the views can be automatically notified of changes in the model. Object references to the model in both the controller and the view allow access to data in the model.

5.1.3 Development environment

The choice of development environment fell on Eclipse ([Ecl]). Eclipse is an IDE and framework for Java, and a number of other structured languages. It is also an open source project. Eclipse has effective features like code completion and displaying of errors as you type, as well as a wide range of plug-ins.

The diagrams in this report will be drawn in Microsoft Visio 10.0, which has support for i.a. UML diagrams.

5.2 Flow diagrams

This section describes the flow in the program, and shows how the user is guided through the different steps in the SAM procedure (Section 4.2).

First, a Customer Behaviour Management Graph (CBMG) is shown in Figure 5.2. This graph describes the possible paths the user can take when using the program. It is useful for exploring how a user would navigate in the system. See [MA01] for more information about CBMG.

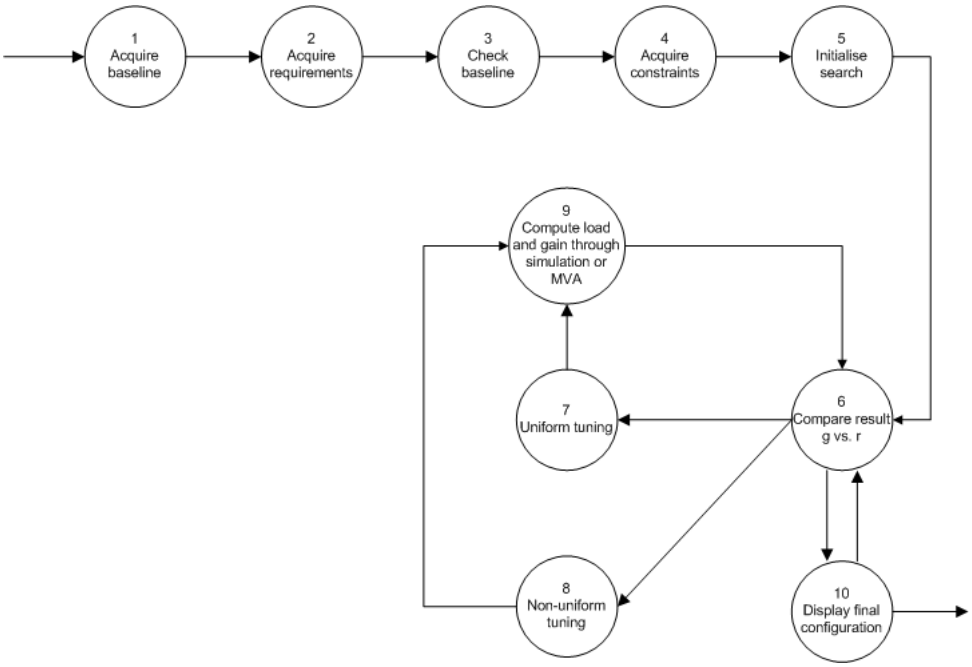

Figure 5.2: Customer Behaviour Management Graph for the SAMe program.

The diagram seen in Figure 5.3 is an Action Port Model (APM) for SAMe. APM is a conceptual workflow modelling language. It is used to make a process diagram, that describes some interesting information for each task: What is done, which roles are involved and both software and physical resources used. See [Car98] for more information about APM.

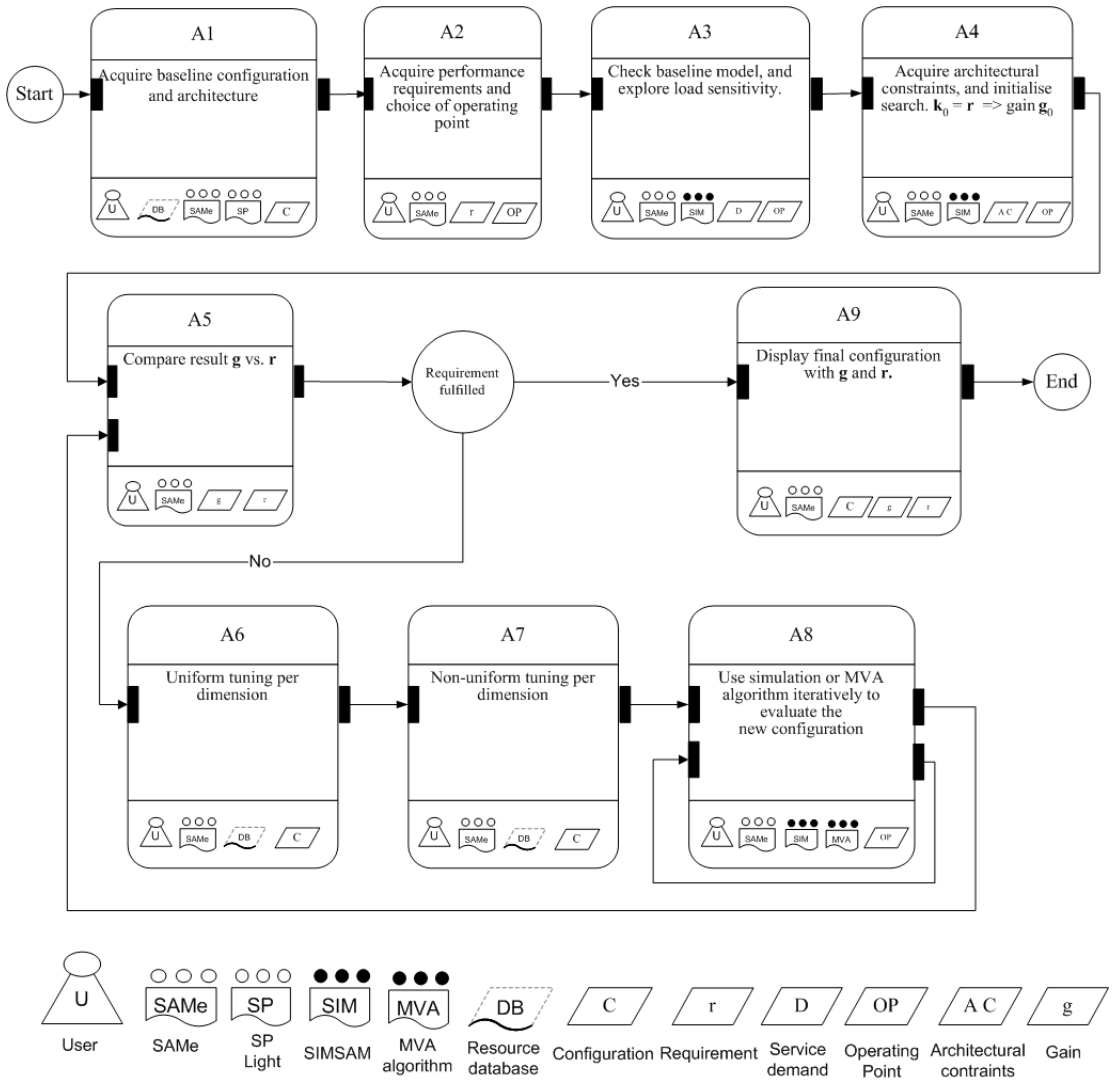

Figure 5.3: APM diagram for the SAMe program.

5.3 Static structure diagrams

The static structure diagrams are part of the Unified Modelling Language (UML), and is describing connections and associations between software units [FS00]. See [UML] for more information about UML.

Figure 5.4 is a package diagram for SAMe. It shows the packages and who they communicate with.

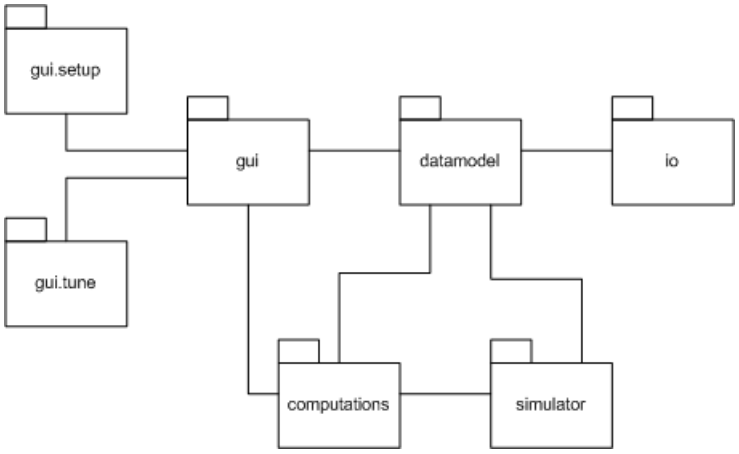

Figure 5.4: Package diagram for the SAMe program.

Figure 5.5 is a class diagram for SAMe. It shows the essential classes, and how the structure of the software is, regarding associations and quantity. The classes for the graphical user interfaces (GUI) is summarised in the class SAMeGUI, because including all the GUI classes would just decrease readability of the diagram, and the information value of the class diagram is the same without them. Class diagrams are very useful for planning and describing which classes are needed, and important attributes and methods.

5.3.1 Class descriptions

Here follows a description of each class in the class diagram (Figure 5.5) and their most important attributes and methods.

Start

This class contains the *main* method, and starts the program in a new thread when the system is ready for it.

SAMeGUI

This is really an abstraction for all the GUI classes in SAMe. They show information, and handle all interaction with the user. There are many GUI classes, and including them in the diagram would decrease readability.

System

This class is a very important class, that contains important information. The requirements are stored here along with the operating point. The system also has a pointer to the root subsystem.

It's central methods are *getAllPrimitives* and *recoverSystemState*. The first creates a list of all the primitives in the tree below a given subsystem. The latter sets the root to be a different subsystem, after the user has chosen to return to a previously saved point in the scalability study.

Tuner

This class takes care of the tuning. It's methods *tuneUniform* and *tuneNonUniform* tries to upgrade or replicate the given component(s) with the given parameters. If not all components could be upgraded or replicated, a list of those who could not is returned.

Every time a tuning is performed, a log record is made by the method *writeToLog* that says what scaling operation was executed.

Log

This class contains the *rootLogPost*, and the point where a new post is inserted. The method *addLogPost* adds a new post to the *previousLogPost*.

LogPost

This class contains the data for a log record. *parent* and *children* are the tree structure pointers, *action* is a String saying what was done in this tuning execution, and *systemState* is the root subsystem as it was before the tuning.

Component

This is an interface that defines the two methods *clone* and *makeTree*. These methods are necessary for the two classes *SubSystem* and *Primitive*. They implement the *Component* interface because the parameters in the methods in this interface can be instances of both classes.

SubSystem

A SubSystem is a component in the system architecture that contains other components. This class implements the interface *Component* described above, and extends *DefaultMutableTreenode* which is a Java class. This is because the system hierarchy is defined as a tree, and with that inheritance SubSystems can be organised in trees as well.

The SubSystems has an architectural constraint *maxReplication* which limits the number of copies that can be made of this SubSystem. The method *clone* creates a new identical copy of the given SubSystem, and is used in the method *replicate* which creates a given number of replications if allowed.

Primitive

A Primitive is a component in the system architecture that cannot be decomposed into other components. This class implements the interface *Component* described above, and extends *DefaultMutableTreenode* which is a Java class. This is because the system hierarchy is defined as a tree, and with that inheritance Primitives can be organised in trees as well.

The Primitives has an attribute *replication* which says the number of copies that are made of this Primitive. The method *clone* creates a new identical copy of the given Primitive, while the method *replicate* increases the *replication* attribute if allowed. The Primitives also has a number of attributes describing the capacity

of the primitive, and *devolvedWork* which describes the amount of work performed at the Primitive.

SAMeFileReader

This class provides the method *readSPModel*, that reads a file with a specified format (described in Appendix A). The *filename* is an attribute to make testing easy, but could be a file chosen by the user in the future.

DatabaseHandler

This class communicates with the database containing all the primitive components. It creates a *newConnection* that requires a jdbc driver and a database path.

getNextUpgradeFactor returns the closest upgrade factor to the given desired upgrade factor in the given dimension for a primitive component, by searching the database for a suitable component in the same category.

getMaxUpgradeFactor returns the greatest upgrade factor in the given dimension for a primitive component, by searching the database for the component in the same category with the greatest capacity.

getDeviceData returns all the attributes contained in the database for a primitive component.

GainCalculator

This class calculates the gain in each of the three dimensions, by executing the methods *calculateGainProcessing*, *calculateGainStorage* and *calculateGainConnectivity* after a tuning operation has been performed. The two last methods is a simple division of new capacity by baseline capacity. But the first involves a number of simulations or MVA iterations to *calculateLoadAtOP* before dividing it by the baseline load.

SimulatorModel

This class is the main simulator class, which contains all the resources in the queue network model, and their distributions of service demands. The method *init* initialises the resources, and *doInitialSchedules* starts the simulation.

SAMResource

This class extends the Desmo-J ([Des]) class *Res*, which has the methods *provide* and *takeBack*. At the time, nothing new is added in this class, but it is extended to ease future development.

Source

This class creates a *User* and schedules it to start when ready through the method *eventRoutine*.

User

This class is a process in the queue network, that executes the method *lifeCycle* and then stops. Inside this method, the process uses resources in a given order and for a given time. If a resource is not available, the process waits in a queue until the resource is ready.

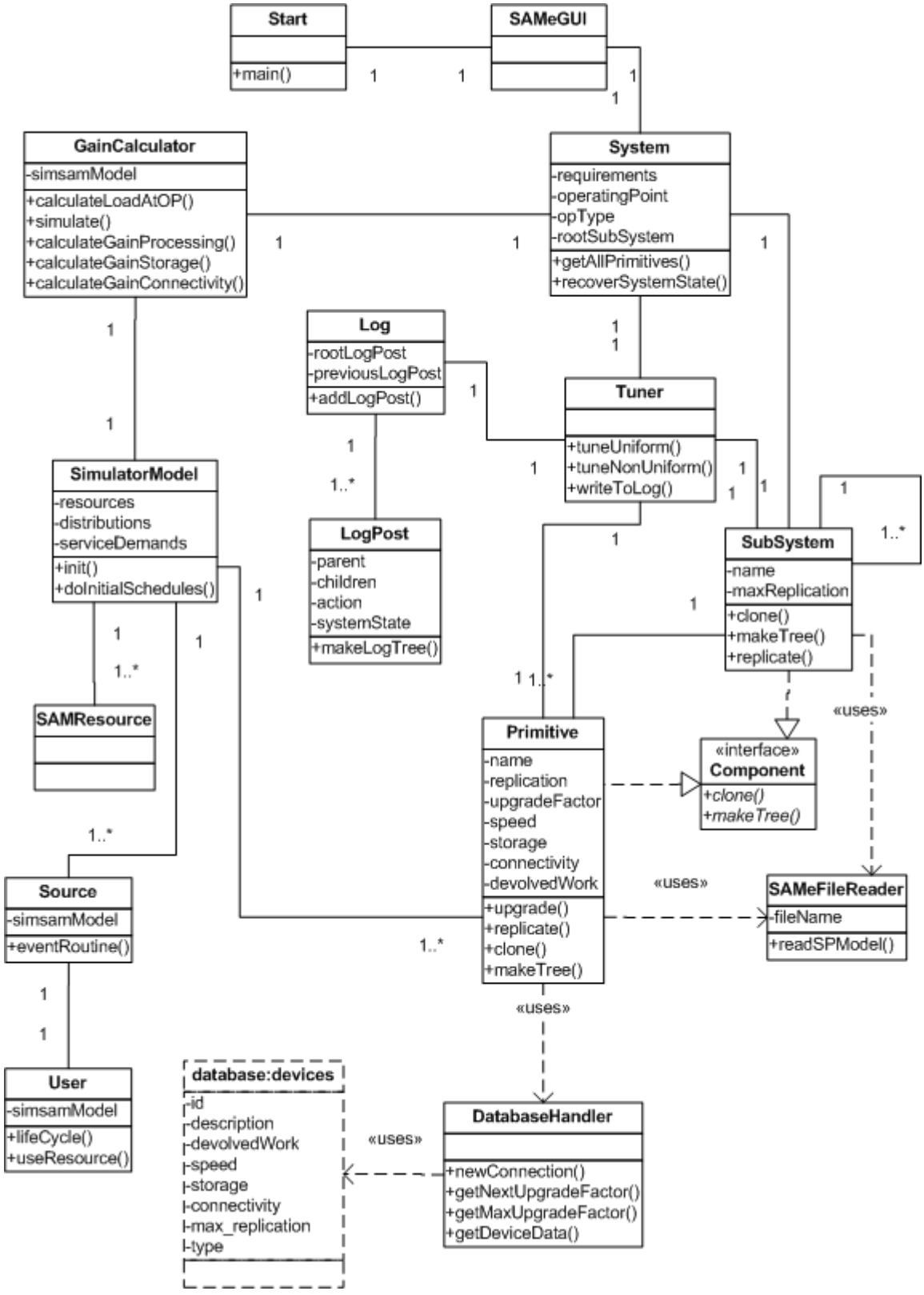

Figure 5.5: Class diagram for the SAMe program.

5.4 Sequence diagrams

The sequence diagrams are also part of the Unified Modelling Language (UML), and are describing communication between objects. They are very useful for detailed planning and describing of procedure calls and interaction between objects.

Figure 5.6 shows a sequence diagram for how SAMe calculates the gain in the processing dimension. There are a number of simulation iterations until the operating point is reached, but the simulation is described in a separate diagram (Figure 5.7). After a simulation iteration is finished, the bottleneck utilisation is compared to the operating point (OP). If *isOPReached* returns false, *adjustLoad* gives a linear estimation of a load that will make it return true, and a new simulation is started.

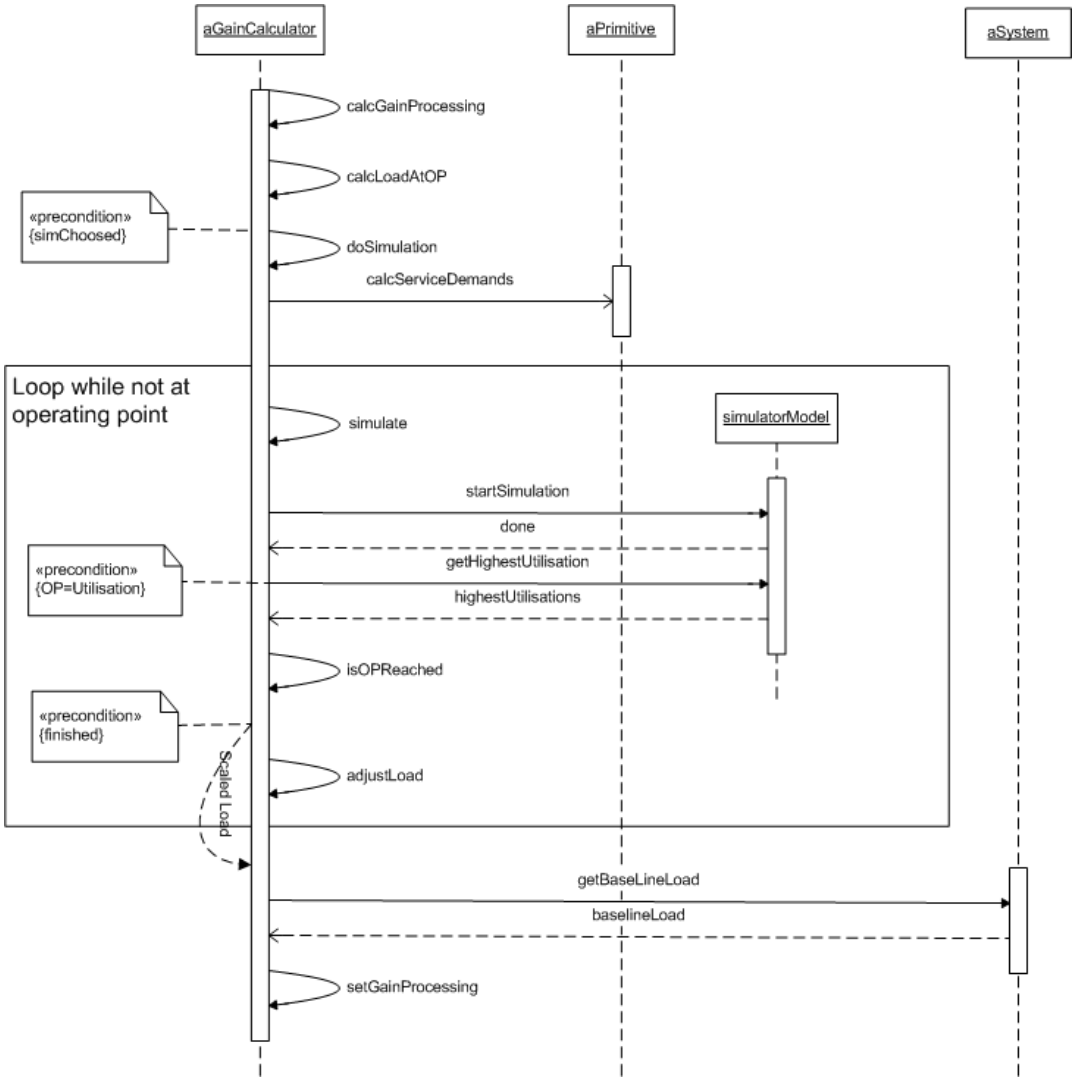

Figure 5.6: Sequence diagram for calculating the gain.

The simulation is an important part of SAMe. A simulation model creates a number of sources, who creates a user each. The users remains inside the *lifeCycle* until the simulation is finished. There they compete for resources with other users. A user issues a *provide* message to a resource, and waits until the resource is available. After a predetermined (exponentially distributed) time, the user sends *takeBack* to the resource, and releases it. Figure 5.7 shows the sequence of the interactions in the simulation.

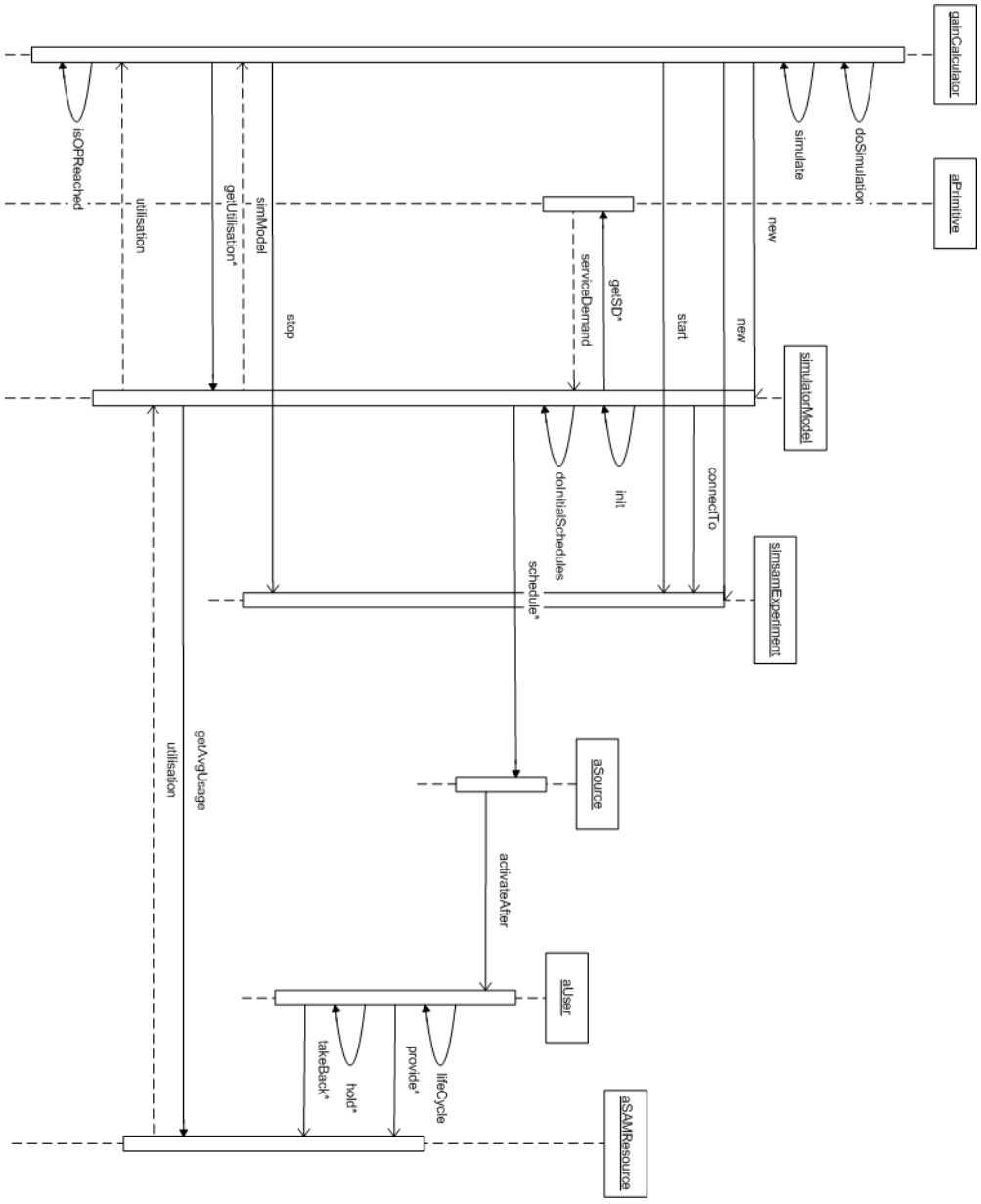

Figure 5.7: Sequence diagram for the simulation.

Upgrade of a primitive is a procedure that retrieves the closest upgrade factor from the database, and sets an attribute in the primitive object. If there is no component that can satisfy the request, an *UpgradeFactorException* is returned. Figure 5.8 shows the sequence of the communication.

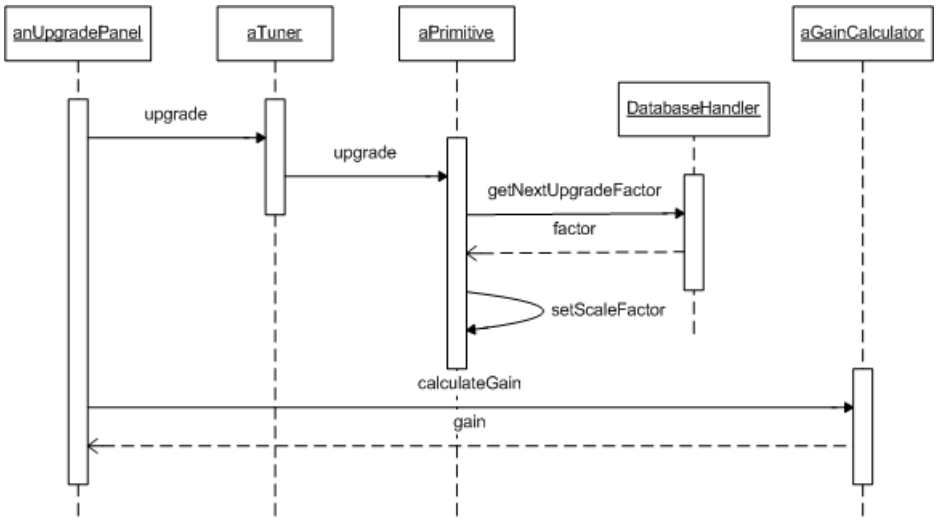

Figure 5.8: Sequence diagram for upgrading a primitive component.

Figure 5.9 shows the sequence of procedures that is involved when a primitive is replicated. Every primitive has an attribute limiting the number of allowed replications, that is checked. If the limit is reached, a *ReplicationFactorException* is returned.

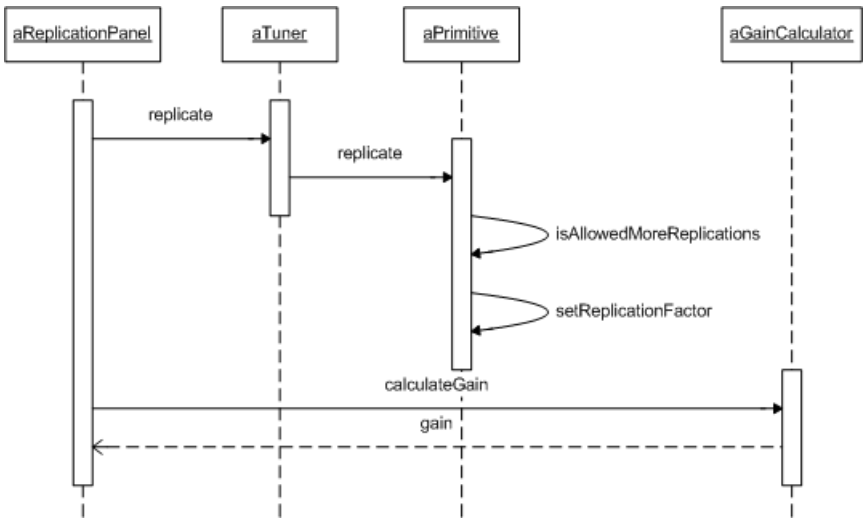

Figure 5.9: Sequence diagram for replicating a primitive component.

The sequence of interactions that are executed when a uniform upgrade of a subsystem with primitive children is performed, is shown in Figure 5.10. The calculation of the gain is described in a separate diagram.

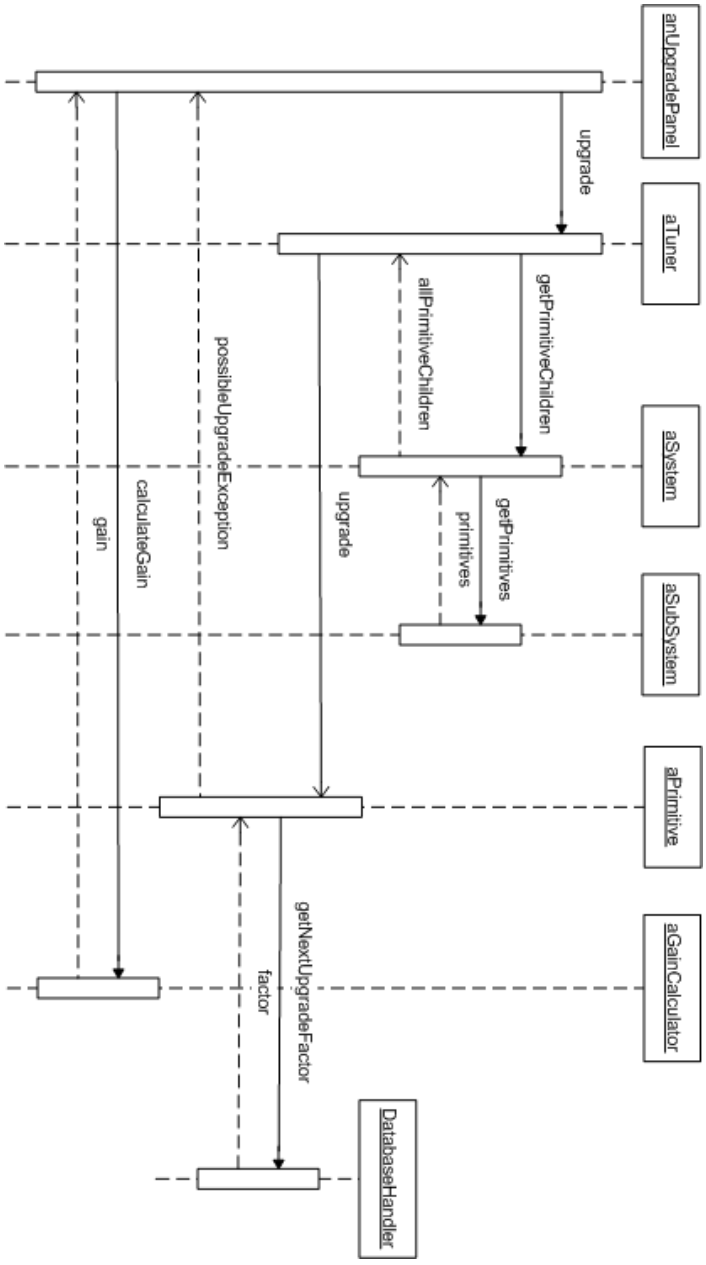

Figure 5.10: Sequence diagram for upgrading a subsystem with primitive children.

Replication of a subsystem with primitive children is a recursive procedure, because the structure is a tree. The *clone* method is called on every child of the replicated subsystem, as shown in Figure 5.11. The calculation of the gain is described in a separate diagram.

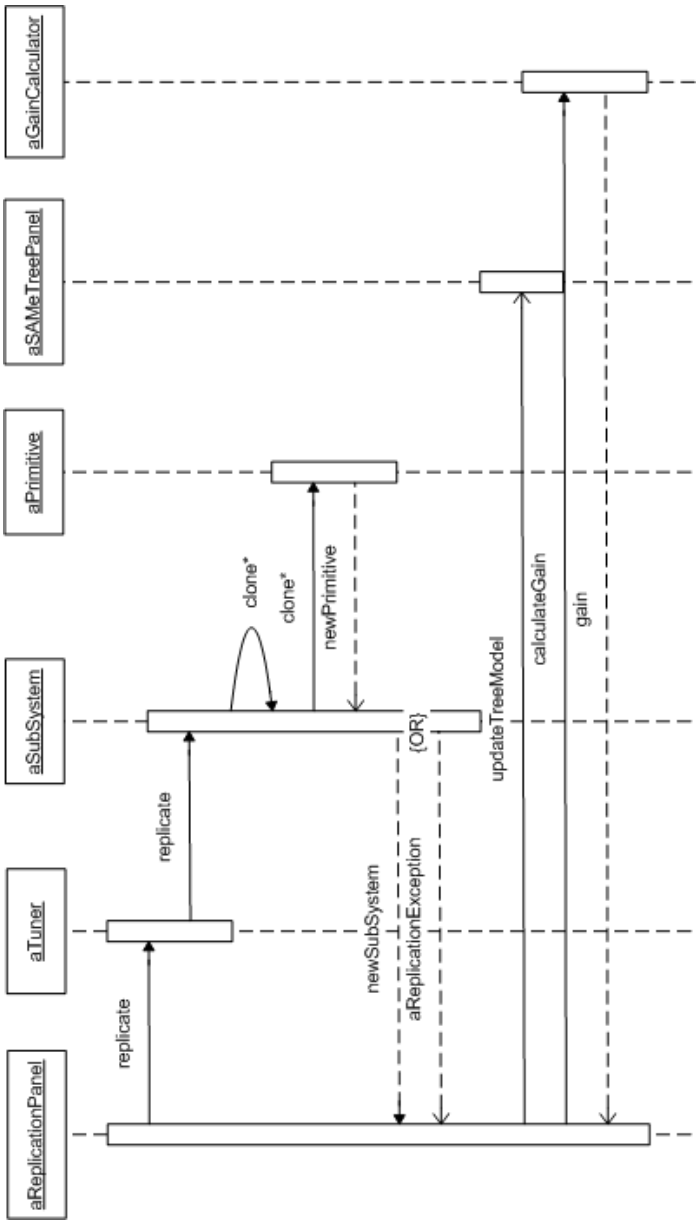

Figure 5.11: Sequence diagram for replicating a subsystem with primitive children.

5.5 User interfaces

The SAMe prototype is supposed to have a graphical user interface, so that it will be easy to use. A number of sketches is drawn up front to create a discussion about appearance and human-computer interaction. This section presents the proposed user interfaces for SAMe, and relates them to the procedure in Section 4.2.

First are the windows where the system architecture is acquired, as described in step 1 of the SAM procedure. They are shown in Figure 5.12.

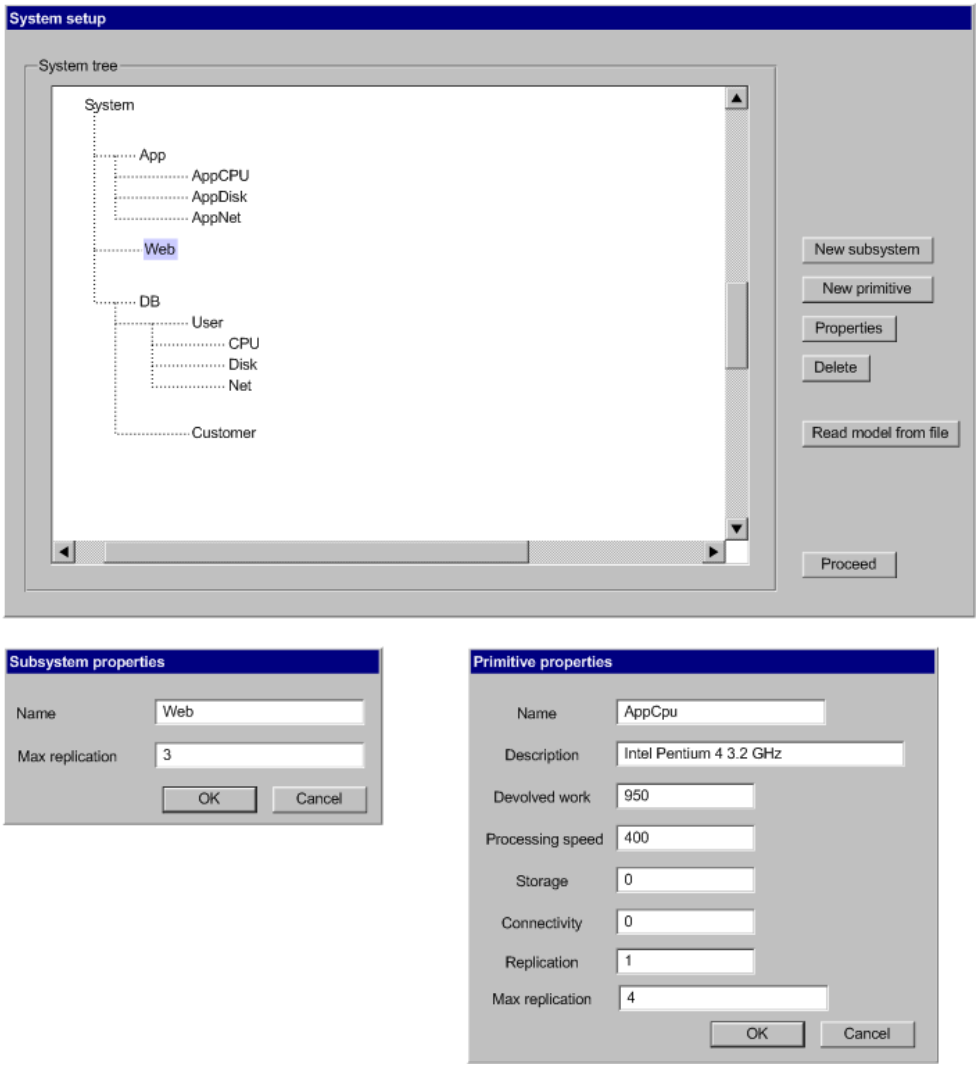

Figure 5.12: Design for the system setup user interface.

Figure 5.13 shows the window where the requirements are acquired, along with the operating point (OP), as described in step 2 of the SAM procedure.

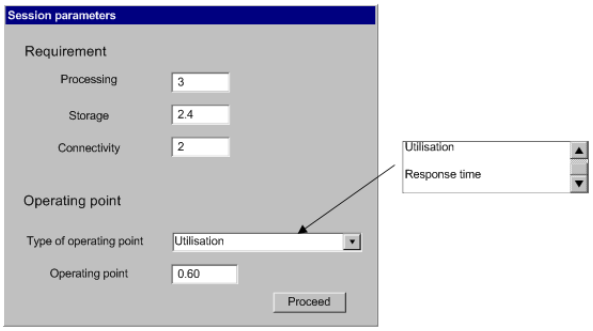

Figure 5.13: Design for the requirement parameters user interface.

Figure 5.14 shows the window where the baseline load is calculated, as described in step 3 of the SAM procedure. After the baseline load is calculated, the search can be initialised by pressing the ”Initialise search”-button, as described in step 5 of the SAM procedure.

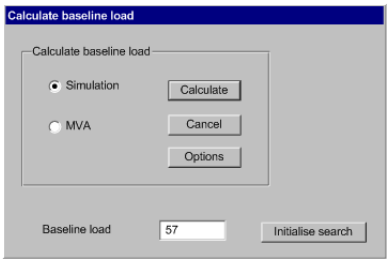

Figure 5.14: Design for the check baseline user interface.

Figure 5.15 shows the window where the calculation options are presented. It should be possible to decrease or increase the calculation accuracy, depending on the purpose of the scalability study. The operating point can also be changed here.

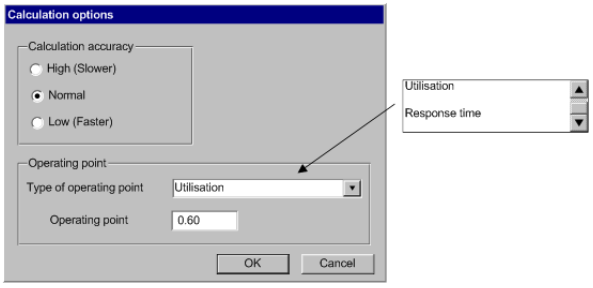

Figure 5.15: Design for the calculation options user interface.

The tuning interface is shown in Figure 5.16, where the App node is selected in the system tree. The system can have at most 8 App nodes, and with the one node already existing, the number of replications left is 7. The ”max uniform upgrade” field displays the lowest ”max upgrade factor” for the primitives below in the subtree, for this dimension. A non-uniform upgrade at this level will not produce any result, since the chosen treenode is not a primitive.

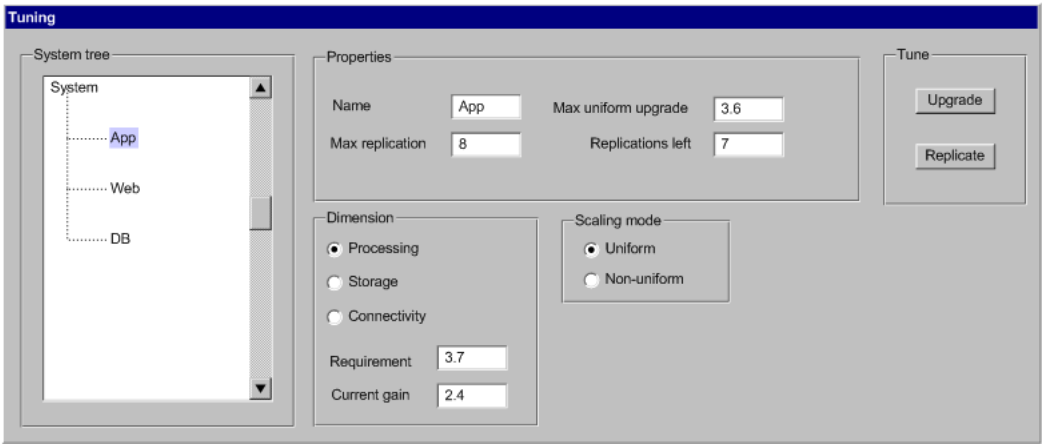

Figure 5.16: Design for the tuning user interface.

In Figure 5.17, the AppCPU node is selected in the system tree. The AppCPU can at most be replicated 6 times, and with the factor two replication already performed, the number of replications left is 4. The ”max upgrade” field displays the highest upgrade factor retrieved from the database for the current primitive category and dimension. A uniform and non-uniform upgrade at this level will produce the same result, since the chosen treenode is a primitive and does not have children.

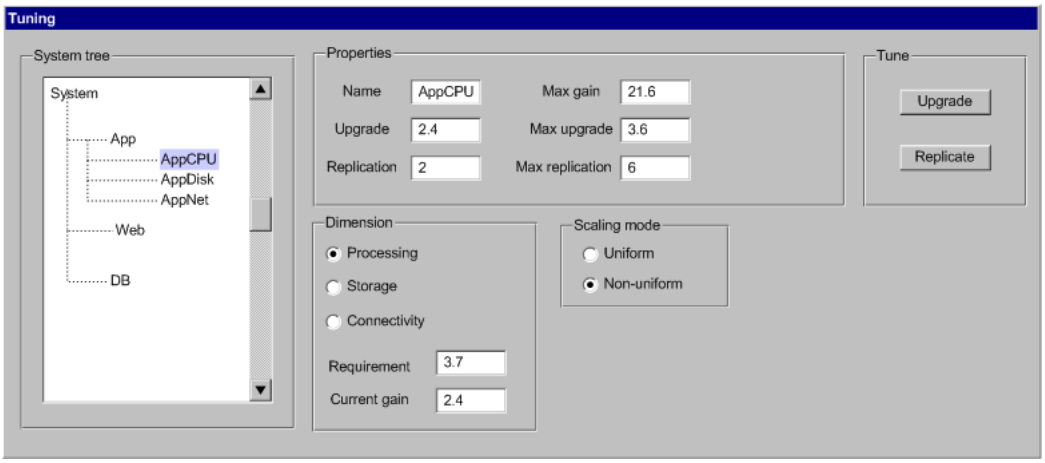

Figure 5.17: Design for the tuning user interface.

The upgrade dialog window is shown in Figure 5.18. The user enters the desired upgrade factor, presses ”calculate”, and so starts the calculation of the gain. Which component that is upgraded is determined by the selected node in the system tree.

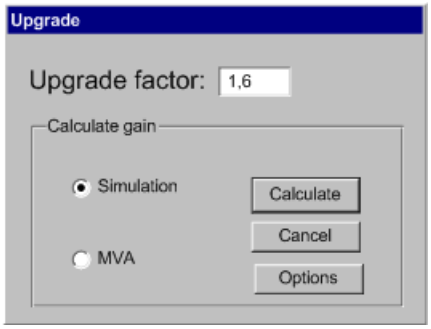

Figure 5.18: Design for the upgrade user interface.

The replication dialog window is shown in Figure 5.19. The user enters the desired replication factor, presses ”calculate”, and so starts the calculation of the gain. Which component that is replicated is determined by the selected node in the system tree.

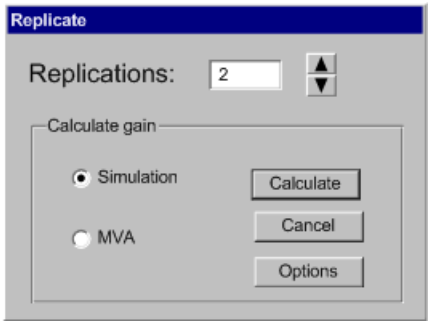

Figure 5.19: Design for the replication user interface.

Figure 5.20 shows the user interface where a graphical comparison between the gain an requirement is displayed. The final system configuration is also displayed. The user can choose to exit, save, tune more, or view the log.

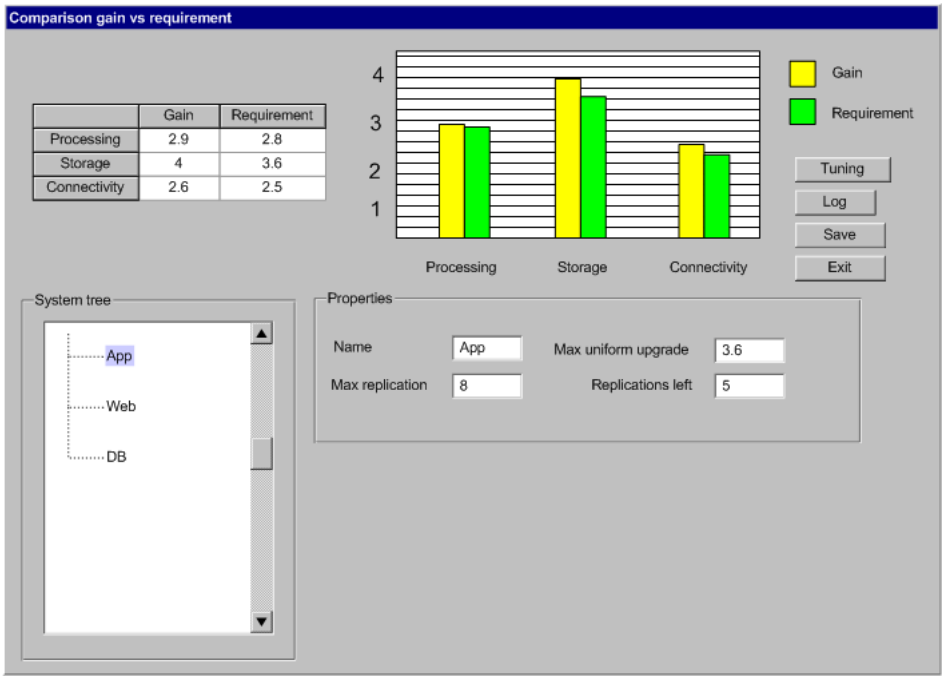

Figure 5.20: Design for the comparing user interface.

Figure 5.21 shows the user interface where the log is displayed. The user can select a node in the tree that symbolises a tune action, and restore the system to the state it was in before that tuning was performed.

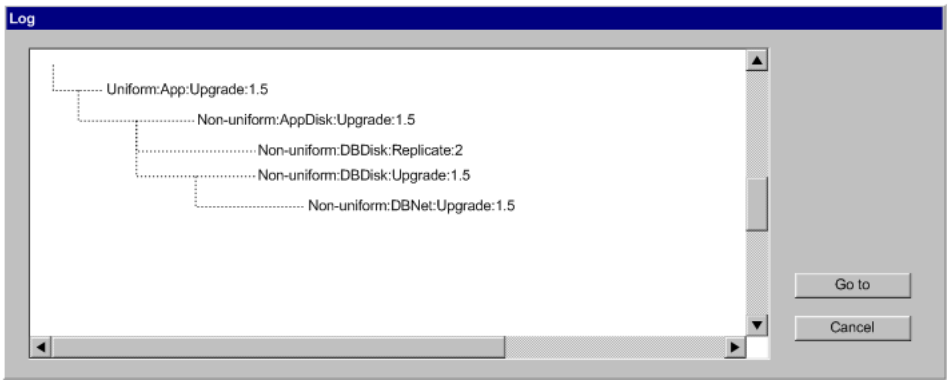

Figure 5.21: Design for the log user interface.

Chapter 6

Implementation

This chapter describes details about the implementation of the SAM Engine prototype (SAMe). Important parts of the program, like the connection to other modules in the SAM software package are explained below. In the end, some features that are not complete are looked at.

6.1 The database

SAMe creates a connection to a database, to retrieve the specifications for primitive components. This could of course be any kind of database, but the testing has been using a MySQL database created solely for this purpose.

The primitives in the database have an integer field ”type”. This field indicates what category the component can be upgraded within, i.e. a primitive cannot be upgraded to a component with a different value of ”type” in the database. The decision of what category a component belongs to is left to the database manager, and is not editable in SAMe. Components with different instruction architectures should typically be in different categories, so the database manager ought to have a detailed knowledge of primitives and their compatibility. Also, there can be an unlimited number of categories, and an unlimited number of components within a category.

A tool for putting components into the database should be very simple to make. To help the user putting the components in the right category, a separate table ”categories” that consists of the category integer and category name could be made. Then a simple index comparison would create a named category list to choose from, and abstract away the category number.

Technical details

- The database is accessible through a PHP-interface at `https://www.itea.ntnu.no/mysql/`
- The username is: ”anderh_master”
- The database is: ”anderh_master”
- The password is: ”1234”
- The jdbc driver is: ”org.gjt.mm.mysql.Driver” which is provided with the prototype.

- The database path is: ”jdbc:mysql://mysql.stud.ntnu.no/anderh_master” but it is uncertain how long this database will exist after the user ”anderh” has left NTNU.

6.2 The connection to SP Light

SP Light is a program developed in a student project, see [Løv04] and [spl] for more information. It is an important part of the SAM software package, and SAMe is meant to communicate with SP Light in some way, because the calculations in SAMe needs some other calculations done in SP Light.

After some thought and discussion, it was decided that the currency of Structure and Performance (SP) is *devolved work*. Initially it was supposed to be service demand, but that would mean duplicate work being done.

In the end, a final list of data that should be transferred between SP Light and SAMe was settled on. The model made in SP Light should be the only one made, and SAMe should read the model with its components and their data.

Because there was no defined output from SP Light at the time, the interchange format was determined in this project and passed on the maintainers of SP Light. They will make SP Light output the necessary data in the summer of 2006.

The possibilities for communication between SAMe and SP Light are many, but the solution had to be simple, because of time constraints. Therefore, writing and reading a formatted file became the solution as shown in Figure 6.1. The format is shown in Appendix A, and is a prefix traversal of the system tree. XML was of course a serious option, but that would require more time than was available, and could be a future extension if desired. Though XML was an inspiration for the format, it is much more simple and depends on the file being correctly formatted.

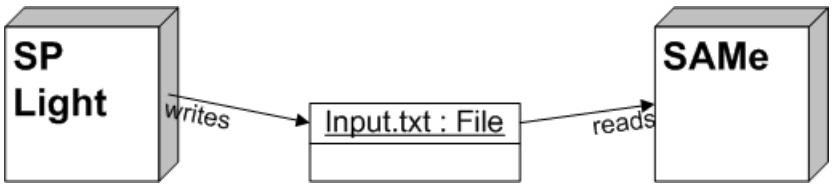

Figure 6.1: Communication between SP Light and SAMe.

6.3 The tuning

This section describes how the tuning is implemented in SAMe, at an appropriate level of detail. The tuning is either upgrade or replicate, as explained in Section 3.3.

6.3.1 Upgrading

Subsystems

Whether the upgrading of a subsystem is uniform or non-uniform depends on the chosen level in the system hierarchy, but in practice in SAMe, the scaling mode will be implicit.

If a subsystem is chosen for upgrade, the scaling is uniform. All the primitive components in the subtree rooted by the chosen subsystem is attempted up-

graded by the given factor in the given dimension. If any primitives could not be upgraded, the user will be notified.

Primitives

Upgrading a primitive is always non-uniform, because a primitive cannot be decomposed further.

If a primitive is chosen for upgrade, its upgrade factor in the chosen dimension will be adjusted to the upgrade factor returned by *getNextUpgradeFactor* in *DatabaseHandler*. The upgrade factors in the other two dimensions are not changed. If the primitive could not be upgraded, the user will be notified.

6.3.2 Replicating

Subsystems

Replicating a subsystem is a process that creates a given number of completely similar subsystem copies with all the belonging children. The copies are inserted as children of the parent of the replicated node, as seen in Figure 6.2.

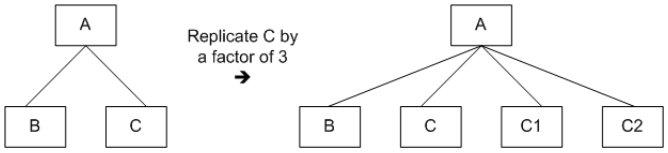

Figure 6.2: Result of replicating a subsystem.

All subsystems have an attribute *maxReplication* that limits the number of replications allowed of that type of subsystem. This is controlled by an idnumber given to each type of subsystem. The idnumber is copied to all replicated subsystems. When a replication is requested, SAMe counts the number of subsystems with the given idnumber, and if that number, plus the requested number of replications, is less than or equal to *maxReplication*, replicating is executed. If the subsystem could not be replicated, the user will be notified.

Primitives

Replicating a primitive in SAMe is a very simple procedure, that needs some thinking to understand why it is correct. When evaluating an architecture, it is important to look for bottlenecks that are hampering the performance. The devices that are examined are categorised by their duties. If a device is replicated by inserting a similar device, or an internal doubling of load capacity, these two devices will still be performing the same duties, and will be considered one unit when looking for bottlenecks in the architecture.

Thus, the procedure is to increase the *replication* attribute in the primitive with the requested number (if not exceeding the *maxReplication*). If *replication* is X after the procedure, X users will be allowed inside the primitive simultaneously when simulating.

6.3.3 Limitations

There are three main limitations in SAMe relating to the tuning.

1. It is not possible to select more than one component in the system, i.e. only one component can be upgraded non-uniformly or replicated at the time.

2. It is not possible to both upgrade and replicate a component in the the same action. SAMe will calculate the gain after upgrading before one can replicate, and vice versa.

3. There is no interaction between the three dimensions. Even though processing, storage and connectivity might have implications for each other, these effects have not been investigated.

6.4 The simulation

The simulation framework used in SAMe is the SIMSAM simulator framework, which is described in [Hol05]. It is a Java discrete event simulator built with DESMO-J [Des] especially for the SAM method. Simulation is used to calculate the load at the operating point and the gain in the processing dimension, as described in Figure 5.6 and Figure 5.7 in Section 5.4.

Although the SIMSAM simulator in [Hol05] assumed a J2EE architecture (Application server, Web server, DB server as in Figure 6.3), the SIMSAM simulation module in SAMe has been generalised to adapt any architecture. As long as it can be represented as a tree, it can be modeled and simulated.

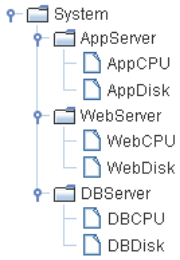

Figure 6.3: The J2EE four-tier architecture.

The Primitives in the system model are used to create SAMResources with names, service demands and replications. The names are used for feedback to the user, the devolved work for the Primitives are converted to service demands, and the replication factors for the Primitives determines the number of slots in the SAMResources.

First an initial run with 10 users is executed, and the bottleneck utilisation is then used to estimate how many users that will be enough to reach the operating point. New iterations with adjusted load are executed until the utilisation is close enough to the operating point.

An objection to simulation is that is takes some time to run. But it is worth mentioning that it will run faster on a faster computer, and it exploits multiple threading, so if the simulations are run on a fast, multi-CPU computer, the runtime would decrease substantially.

In [Hol05] it was shown that simulation in SIMSAM can give realistic results compared to measurement on the reference system. The same test scenario has been carried out in SAMe, and the results are still the same as in [Hol05]. This shows that the integration of SIMSAM into SAMe has been successful.

So the SIMSAM simulator is a trustworthy component of the SAMe prototype, and that is important since no other method for calculating the gain is implemented in SAMe so far.

Chapter 7

The result

7.1 Report

This report is documenting the development of the SAM procedure and its software tool the SAM Engine. The report has been written in Eclipse[Ecl] using LaTeXformatting for good appearance. An effort has been made to give all and the best information in an understandable way, so that every kind of reader may find the contents interesting and useful.

We hope that this report will be of great help in a possible further development of the SAM Engine.

7.2 SAM Engine

A SAM Engine prototype has been implemented in Java2SE5.0[Java] with a connection to a MySQL database[MyS]. As a curiosity it can be mentioned that the development has resulted in a total of 3800 lines of code. The program and all required files are attached separately.

Two screenshots from SAMe are shown and explained in Appendix C. These screenshots shows the user interfaces for the most central user interactions.

A runnable program, and source code are among the attachments to this report, and they are a part of the Master's Thesis. Details about attached files are given in Appendix E.

We hope that the SAM Engine will be of great use to people performing scalability assessment studies.

7.2.1 Requirements fulfilment

Table 7.1 shows to what degree the functional requirements from Section 4.3 are fulfilled. The ranking is on a scale from 1 to 5, where 1 means *not at all fulfilled*, and 5 means *completely fulfilled*.

As the project went along, some requirements had to be given less priority because of time constraints on the development of the SAMe prototype. That is the reason for most of the requirements that have not been completely fulfilled, i.a. three types of operating point, and saving projects. Other requirements, like dealing with load dependent effects and displaying load sensitivity, we have not been sure how to implement.

Looking at the ranking results, it is clear that SAMe has covered a great deal of the requirements. An average fulfilment ranking of 4.1 is very good, and it can be concluded that the SAMe prototype is technically a success, although the most

important requirement F1 needs a further investigation. A validation of requirement F1 is given in Section 8.1.

Number	Requirement	Fulfilment
F1	The SAMe program must support the SAM procedure.	4
F2	SAMe must have a connection to a database.	5
F3	SAMe must have a connection to an SP tool.	4
F4	The user must be able to save a scalability assessment study.	1
F5	The user must be able to load a scalability assessment study.	1
F6	Reset a scalability assessment study back to baseline.	3
F7	SAMe must have an interface to the subsystem repository.	1
F8	SAMe must be able to import subsystems from the repository.	3
F9	The system modeled in SAMe must have a tree structure.	5
F10	SAMe must contain a list of all the subsystems in the system.	5
F11	SAMe must contain a list of all the subsystems in the system.	5
F12	Export a list of all the components in the system.	1
F13	Enter the three vector components of the requirement r.	5
F14	Enter the numerical value of the operating point(OP).	5
F15	The user must be able to choose the type of the OP.	5
F16	Choose the method for calculating the load at the OP.	5
F17	The user must be able to change the type and value of the OP.	3
F18	Choose whether to upgrade or replicate a system component.	5
F19	Graph describing the load sensitivity in a given interval.	1
F20	Display a graph comparing the gain, and the requirement.	1
F21	Choose to continue the scalability assessment, or to end it.	5
F22	Display the final configuration, the gain and the requirement.	4
F23	The subsystems must have a identification number(ID).	5
F24	The subsystems must have a name.	5
F25	The subsystems must have a list of child subsystems.	5
F26	The subsystems must have a list of primitives.	5
F27	The subsystems must be able to have architectural constraints.	4
F28	Choose a subsystem and enter any architectural constraints.	4
F29	It must be possible to replicate subsystems.	5
F30	Change the specifications of the subsystems.	2
F31	The primitives must have a identification number(ID).	5
F32	The primitives must have a name.	5
F33	The primitives must have a description.	5
F34	The primitives must have devolved work.	5
F35	The primitives must have a speed.	5
F36	The primitives must have a storage size.	5
F37	The primitives must have a connectivity.	5
F38	The primitives must get their data from a database.	5
F39	It must be possible to replicate primitives.	5
F40	Primitives must have a replication factor property.	5
F41	It must be possible to upgrade primitives.	5
F42	The primitives must have a scale factor for each dimension.	5
F43	Change the specifications of the primitives.	2
F44	Read devolved work from an SP tool, and calculate SD.	5
F45	Use a simulation framework to calculate the gain.	5
F46	Uniformly adjust the scale factors for the subsystems.	5
F47	Non-uniformly adjust the scale factors for the primitives.	5

Continued on next page.

Table 7.1 – continued from previous page

Number	Requirement	Fulfilment
F48	Compute the gain g and compare it with the requirement r.	4
F49	Use a heuristic that adjusts the scale factors in an optimal way.	1
F50	SAMe must write to a log everything an action is performed.	5
F51	A log record must consist of an action, and it's parameters.	5
F52	Save every scaling action, and try a different scaling path.	4
F53	Compute the gain in each of the dimensions.	5
F54	OP is: utilisation, response time or Kleinrock saturation point.	3

Table 7.1: Degree of fulfilment of requirements

7.2.2 Verification

To verify that the SAM Engine produces correct results, it was decided to use the same test as in [Hol05], which originated from [RM05].

The test involves a reference system that is simulated, scaled up, and simulated again. The architecture consists of three servers, each with a cpu and disk. In addition it was calculated a "User think time", where the user would not apply load on the servers. The architecture is shown in Figure 7.1.

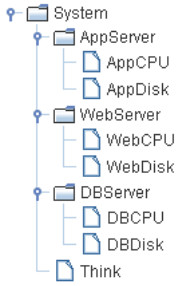

Figure 7.1: The architecture from the test.

It is important to know that SAMe can support any architecture, not just the one in this test. But since the test case from [Hol05] was the most comprehensive, with measurements on the real system, it was decided to be used. However, this test only tests the capabilities in the processing dimension. The simulation currently does not involve storage or connectivity.

Each of the devices in the system has a service demand, derived from measurements in [RM05], and the service demands are used as parameters in the simulation. The baseline parameters are shown in Table D.1 in Appendix D.

The test proceeds like this:

- The baseline configuration is acquired and simulated
- The CPUs are upgraded by a factor of 2
- The new configuration is simulated again
- The gain is calculated

The means for calculating the gain is the load at the operating point(OP), which in the SAMe prototype is utilisation at the bottleneck component.
$Gain = \frac{scaled-upLoad}{baselineLoad}$

One thing is different from the test in [Hol05] though, there are currently no load dependent effects in SAMe, as discussed in Section 8.2. So the load will not be the same. But since both the baseline and the scaled-up version are calculated without such effects, the gain should usually not be affected too much. There will of course be some variance if the load dependent effects are non-linear.

The problem with the test case is that the system was heavily affected by non-linear load dependent effects. The source was identified as garbage collection, and it hampered the performance of the system. Measurements in [RM05] give a capacity increase of only 1.4 when using utilisation as OP, while simulations in SAMe give an increase of 1.9 at the same OP. [Hol05] included the non-linear effect ($GC_{CPUUtilization}(n) = 0,09 \times n^{1.810}$) found in [RM05] in the simulation with success. Just to prove that SAMe can produce the same results, it was included in the source code of SAMe as well. But because that would destroy the generality of SAMe, it was removed after the testing was finished.

The test with SAMe including the garbage collection gave a gain of 1.5, which is close enough to the measured gain of 1.4 to assure the capability of SAMe.

This test scenario will be used as an example to evaluate the ten steps of the SAM procedure in Section 8.1.

7.2.3 Possible improvements

These improvements are suggestions for features that could be implemented in the future.

Integrating with an MVA algorithm

A reason for integrating an MVA algorithm([MAD94]) into SAMe is that the simulation is rather slow, and an MVA implementation will solve the queuing network much faster.

One solution that strengthens the speed argument, is to use a faster programming language than Java. There is an implementation of the MVA algorithm i C, that is available under the GPL licence[SF]. It can be found on [Che]. This could be integrated with SAMe by using the Java Native Interface(JNI)[Mic] which is a native programming interface that is part of the Java SDK. JNI lets Java code use code and code libraries written in other languages, such as C and C++, and allows you to call Java code from within native code.

There is also an MVA algorithm already implemented in Java, which could be integrated with SAMe. The source code and demo can be found on [MVA].

Almost completed features

- Navigation in SAMe
  It could be made possible to navigate differently in SAMe than it is now, maybe go back and e.g. change the requirements. That is an easy extension, all the windows are reusable.

- Editing components while tuning
  When in the tuning interface, it could be possible to edit component properties. The component properties windows from step 1 are reusable for this feature.

- Calculation options
  An options window has been made, but not integrated yet. The required work is to reflect the chosen options in the datamodel.

- Logging and going back
  A procedure for logging the scaling actions has been made, but there is currently no interface for viewing the log and going back. The system tree is reusable for this log system.

Other improvements

- Inheriting scale factors in all dimensions
  The tuning is currently adjusting scale factors for exclusively one dimension at the time. This could be changed by returning the scale properties for all three dimensions from the database when tuning.

- Other types of operating point
  Implementing response time and Kleinrock saturation point[RM05] as operating points increases the usefulness of SAMe.

- Saving and loading projects
  A scalability assessment project could be saved and loaded as XML([XML]).

- Saving and importing sub-trees from repository
  A repository of subsystems could be made, either as a database or as XML.

- Multiclass transactions
  If the components received a vector of devolved work, SAMe could be adjusted to calculate multiclass transactions. The SIMSAM simulator would need some small changes, but an implementation of MVA would also support multiclass.

- Load sensitivity
  Displaying the load sensitivity in some way could be done by calculating utilisation/response time for a given interval around the load at the OP.

- Graphical display
  There are several open source Java packages that supports graphs, and some can be found at [Javb].

- Better feedback
  More detailed and visible feedback during simulation could increase the user's understanding of the simulation.

- Icons
  Showing different icons in the tree for subsystems and primitives is simple, just find some suitable images.

Chapter 8

Evaluation

The SAM method is described in Section 3.2 and in more detail in Section 4.2. The SAM engine (SAMe) prototype is described in Chapters 4, 5 and 6. But how well does SAMe follow the SAM method? It is not so important that the prototype works, if it doesn't implement the SAM method. But how can one prove that SAMe actually does that? A step by step validation of the SAM procedure in Section 4.2 will give a clear indication. The test from Section 7.2.2 is used as a walk-through example, to clarify what can and can't be done.

8.1 Validating the ten steps

This section gives an evaluation of how the ten steps are implemented in SAMe. A step is implemented either *completely*, *partially* or *not at all*.

Step 1: Acquire baseline configuration and architecture
The baseline configuration is modeled in SAMe, either manually or by reading a model from file. The model read from file can be altered too. The specifications for the primitive components are chosen from a database.

This step is completely implemented.

Step 2: Acquire performance requirements and choice of operating point

The requirement in each of the three dimensions, and the choice of operating point is entered in SAMe. The requirements can be equal or different for each dimension. The drawback in this step is that only utilisation is implemented as operating point. There was not enough time to implement other types of operating points. The example did not have specific requirements, so they were set to 1. Neither did the example involve the storage or connectivity dimension, but the dimensions does not affect each other in SAMe, so it didn't matter here.

This step is completely implemented.

Step 3: Check baseline model, and explore load sensitivity
The baseline model is simulated, and the load is displayed to the user. The load sensitivity is not explored however, because there was no clear procedure for how it was supposed to be explored and displayed. However, one can interpret the output information from the simulation to be information about the load sensitivity. The output shows the load and bottleneck utilisation for each iteration, and by looking at the increases in utilisation compared to the increases in load, one can get an indication of the load sensitivity.

This step is partially implemented.

Step 4: Acquire architectural constraints
In step 1, when modelling the system, the constraint "max number of replications" is put into the components. But there might be other constraints in the architecture, that are not included in the model. There is no support at the time for registering other constraints, or changing the constraints after step 1. The reason for that is uncertainty of what other constraints might include, and a shortage of time. Since the example did not specify "max number of replications", it was set to 5, so it would not cause any problems.

This step is partially implemented.

Step 5: Initialise search
After the baseline load is calculated, it is possible to initialise the search. The system is attempted upgraded uniformly by the requirement factor in each dimension, and then the gain is calculated. If not all components could be upgraded, the user is notified. The example did not have specific requirements, so they were set to 1. This resulted in the baseline system beeing simulated again, without any changes.

This step is completely implemented.

Step 6: Compare result g vs. r
After the gain has been calculated, the requirement and the gain for the current chosen dimension is shown in separate but adjacent fields. This makes the comparison easy, even though it is not graphical. But it is really not a clear step in the procedure, and the comparison is for only one dimension at the time. There was not enough time to implement this step completely.

This step is partially implemented.

Step 7: Uniform tuning per dimension
It is fully possible to tune a subsystem uniformly (tuning a primitive uniformly would not make sense). If upgrade is chosen, the relative increase in capacity for all the children is attempted set to the given factor in the selected dimension. If replication is selected, the subsystem is attempted replicated by the given factor. The example did not involve uniform tuning.

This step is completely implemented.

Step 8: Non-uniform tuning per dimension
Non-uniform tuning in SAMe includes upgrading and replicating of primitives, as defined in the procedure, but it is not possible to remove components after the model has been created. That could be a simple but useful extension, if the system has grown larger than necessary, but there was not enough time to implement this step completely. The example involved upgrading all the three CPUs in the system by a factor of 2. That was possible, because there was available in the database a component in the same category with double the processing speed. (Put there for the example)

This step is partially implemented.

Step 9: Use simulation or MVA algorithm iteratively
Only simulation is implemented in the SAMe prototype, and not MVA. But the simulation is used iteratively to calculate the load at the OP and thus the gain in the processing dimension. The OP was 0.6 in the example, and that was used to calculate the load and thus gain in the processing dimension. The calulation of the gain in the storage and connectivity dimensions is not

using simulation, SAMe just retrieves the component from the database that best matches the upgrade, and sets the gain to: $Gain = \frac{scaled-upCapacity}{baselineCapacity}$. As mentioned in Section 6.3.3, SAMe does not treat interactions between the dimensions.

The greatest drawback in this step is that there is no integrated handling of load dependent effects. A new SP model has to be made with new devolved work, and then read into SAMe and scaled again. A discussion of this issue is given in Section 8.2.

Both integrating an MVA algorithm and handling load dependent effects are quite complex tasks, and required more time than was available.

This step is partially implemented.

Step 10: Display final configuration with g and r

There is no separate step for displaying the final configuration and gain in the prototype. However, the system and the gain are displayed continuously in the tuning window, and the user can decide at any time when to end the scalability study. But there is unfortunately no functionality for saving and loading studies at the time. There was not enough time to implement this step completely.

This step is partially implemented.

8.2 SAMe and SP boundaries

SAMe and SP Light are both important parts of the SAM software package, but because their use is so interconnected, it is important to separate the tasks assigned to each program.

SP is a way of constructing a static model of a system and distributing load to components in the system, and the currency used to describe the load is *devolved work* [Hug88]. Devolved work is the main information that is transferred from SP Light to SAMe. In SAMe the devolved work is converted to service demand (SD), and used in simulation (and in MVA if implemented). Then there won't be duplicate calculations of service time or service demand.

An important approach to the scalability assessment is the hierarchical approach. It requires that devolved work is stored for intermediate levels in the system. This way, it is up to the user what detail level he/she wants to investigate. If only the server level is investigated, the servers can be modeled with devolved work, and the assessment is performed without further decomposition. This opens for great opportunities in reusability, when previously explored standard subsystems with performance properties can be imported into the model.

A drawback with the current version of the SAMe prototype concerns the handling of load dependent effects. If these effects are linear, they would be relatively easy to deal with, but often the load dependent effects are non-linear, like garbage collection in Java. And then there is no way of representing the load dependent effects in SP Light, because it is a static model which is constructed on the basis of a given load. The way SP Light and SAMe are today, handling of load dependent effects requires that the SP model is reevaluated for every scaling action that increases the load capacity of the system. This is very cumbersome and time-consuming.

What the function for the load dependent effect is, is discovered through empirical studies and measurement, and will usually be different from architecture to architecture and study to study. Therefore, the function can not be determined and cemented in the source code of SAMe. But SAMe should clearly be the one to handle the load dependent effects, as it is SAMe that uses simulation (and MVA)

to calculate the gain for the evolving system. In [Hol05] a non-linear function for garbage collection in Java was implemented into the simulator, and an amount of SD relating to garbage collection was added to the regular SD for a CPU. That function was obviously a good estimate, since the SIMSAM simulation results were better that the MVA results. It shows that a realistic function for load dependent effects can be implemented into SAMe with success. A suggested solution is described in Section 9.2.

8.3 The guidance

Because the users of SAMe most likely will be people with different backgrounds and different knowledge about SAM, it is useful with a workflow in SAMe that guides the user through the processes in the SAM procedure(Section 4.2). The navigation allowed in the SAMe prototype is shown in Figure 5.2 in Section 5.2. It is meant to be guiding, not enforcing.

There is a partially determined path from start to finish, and information is presented in a way that should guide the user to making the most profitable decisions. Especially the first five steps of the procedure are part of a determined path.

In the tuning phases however, the freedom of scaling options is given to the user. But information about the system, the gain and the scaling properties of all components is always visible. This information should be of good assistance while performing a scalability study in SAMe.

8.4 Conclusion

This conclusion will evaluate the project, and whether the main goal for the prototype has been accomplished: *The support for the SAM method*

Section 8.1 looked at the degree of completion for the ten steps. A summary is given in Table 8.1.

Step	**Degree of completion**
Step 1	Completely
Step 2	Completely
Step 3	Partially
Step 4	Partially
Step 5	Completely
Step 6	Partially
Step 7	Completely
Step 8	Partially
Step 9	Partially
Step 10	Partially

Table 8.1: Summary of step completion

It shows that all ten steps are implemented to a certain degree, and most of the absent parts are minor and not critical. Because this version of SAMe is a prototype, the focus was on creating the essential functionality, and therefore a limited amount of time was prioritised on implementation. The most critical drawback is the handling of load dependent effects, as discussed in Section 8.2, but a solution to that is suggested in Section 9.2.

The steps were evaluated by applying the method to an example. The example was not a large and detailed one, regarding modelling detail and scaling operations,

but good enough to test that the SAMe prototype works and supports the main functionality of the SAM method.

A purpose for making a tool like the SAM engine, is to be able to test whether the SAM method is practically feasible for real systems. The first step was to develop the static modeling tool (SP Light), and now a tool for exploring the dynamic model has been made(SAMe). The SAM method is still evolving, new challenges and solutions emerge continuously, and the research is still in progress. Only future work and use will decide if the method is indeed feasible, but the results so far indicate that it will be an important part of scalability and performance evaluation.

Chapter 9

Further work

These sections are suggestions for further work in this direction of scalability assessment.

9.1 Other aspects with scalability

There might be reasons for not building a system the way the SAM method suggests as the best. Obviously, the availability of components will limit the system, but we found that the following two other factors could often play a role.

9.1.1 Economy

Many companies would not afford the optimal solution, because budgets limit their capability of investments in computer hardware. And then one might say that it is not an optimal solution, unless it takes cost into account. The SAM engine prototype does not involve economic issues in the scalability assessment method. But a most likely popular feature from commercial users' viewpoint, would be to have a cost requirement as well as the performance requirement.

This would require that the current prices of computer components are included in the database, and of course regularly updated. A simple algorithm could then tell if it is cheaper to upgrade or replicate. If an overall cost requirement is given, the program could indicate what the user can build with that amount of money.

9.1.2 Power

As computers increase in capacity, their power consumption increases too. Thus, there might be a constraint on power supply, both internally in the system and to the system externally. A lack of power supply to components may decrease performance considerably.

The SAM engine prototype does not deal with power supply, but it could be very interesting to include it as a dimension to the scaling. E.g.: It could be that the electric installation in the rooms containing the computers are old and not scaled for the kind of power consumption a scaled-up system would cause, and therefore the external power supply needs to be scaled up as well.

9.2 Dealing with load dependent effects

As discussed in Section 8.2, the current handling of load dependent effects in SAMe and SP Light is inadequate. Therefore a feature for adding such effects to components in the system could be developed. Here is a suggestion for a solution.

The user could select a component that is affected by load dependent effects, and then would be presented a choice of a set of standard functions like:

- $a \times n^x$
- $a \times x^n$

where n is the load. The user would then enter a and x, and the function will be added to the chosen component.

Successful integration of load dependent effects into the performance calculation of SAMe would substantially increase the usefulness of the whole SAM software package, because of the increased correctness of the calculations.

9.3 Interaction between dimensions

The SAM engine treats each of the three dimension as separate paths, without any influence on each other. We believe that in reality there are effects that originate from interaction between dimensions.

- Processing vs. Connectivity
- Storage vs. Processing
- Connectivity vs. Storage

These effects needs to be investigated and made concrete.

Appendix A

SP model file format

The tree structure is listed in prefix notation, and the columns contains the following *space separated* information:

- The first column determines the type of component to be inserted, either 0 of 1. 0=subsystem and 1=primitive. Or it causes a navigation in the tree, either < or >. < navigates a level down, and > navigates a level up in the tree.
- The second column is the name of the component.
- The third column is not the same for the two types
  - Subsystems: This is the number of maximum replications allowed.
  - Primitives: This is the devolved work.
- The fourth column is just for primitives. It determines the replication factor (integer) for the component, i.e. the number of components.

```
----Begin example:----

0 System 1
<
0 AppServer 4
<
1 AppCPU 43.71 1
1 AppDisk 28.12 2
>
0 WebServer 5
<
1 WebCPU 40.71 1
1 WebDisk 20.12 1
>
1 UserThink 812 99999
```

Appendix B

Gantt-diagrams

Here are the three Gantt-diagrams described in Chapter 2. They are constructed using the tool Microsoft Project 10.0

ID	Task Name	Duration	Start	Finish
1	Planning and preparing	15 days	Mon 16/01/06	Fri 03/02/06
3	Requirements and design	30 days	Mon 06/02/06	Fri 17/03/06
10	Report writing	75 days	Fri 17/02/06	Wed 24/05/06
2	Winter vacation	5 days	Mon 20/02/06	Fri 24/02/06
4	Peter away	14 days	Wed 22/02/06	Mon 13/03/06
8	Implementation	25 days	Mon 20/03/06	Fri 21/04/06
5	Peter away	32 days	Thu 06/04/06	Mon 15/05/06
7	Easter vacation	6 days	Mon 10/04/06	Mon 17/04/06
9	Testing	7 days	Mon 24/04/06	Tue 02/05/06
11	Finish first draft of report	0 days	Wed 10/05/06	Wed 10/05/06
6	Peter away	40 days	Thu 18/05/06	Fri 30/06/06
12	Fisnish second draft of report	0 days	Wed 24/05/06	Wed 24/05/06
13	Delivery of report	0 days	Fri 02/06/06	Fri 02/06/06

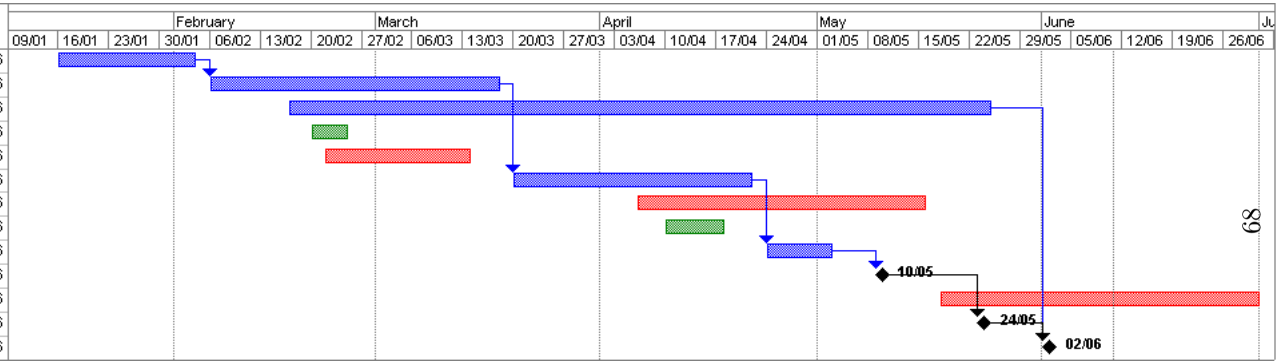

Figure B.1: Gantt diagram as of February 3rd.

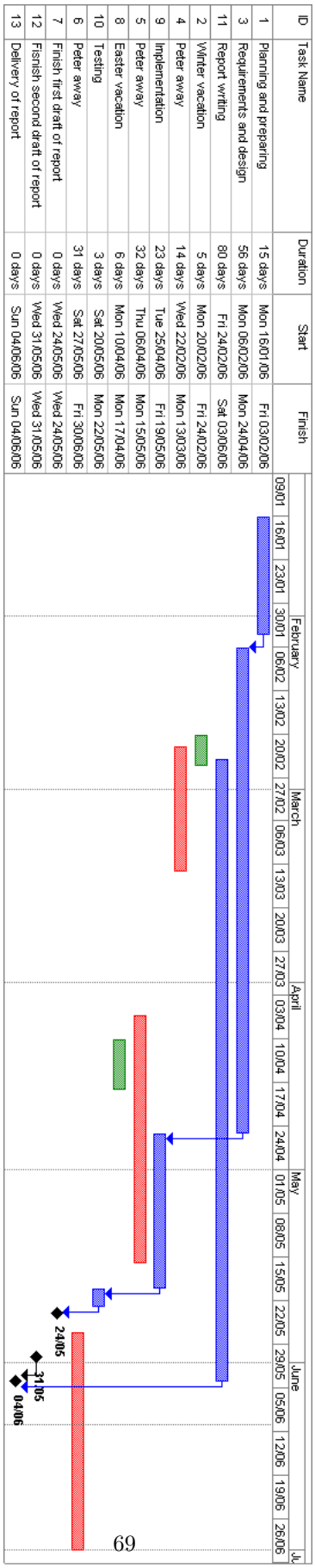

Figure B.2: Gantt diagram as of April 26th.

ID	Task Name	Duration	Start	Finish
1	Planning and preparing	15 days	Mon 16/01/06	Fri 03/02/06
3	Requirements and design	56 days	Mon 06/02/06	Mon 24/04/06
11	Report writing	80 days	Fri 24/02/06	Sat 03/06/06
2	Winter vacation	5 days	Mon 20/02/06	Fri 24/02/06
4	Peter away	14 days	Wed 22/02/06	Mon 13/03/06
9	Implementation	23 days	Tue 25/04/06	Fri 19/05/06
5	Peter away	32 days	Thu 06/04/06	Mon 15/05/06
8	Easter vacation	6 days	Mon 10/04/06	Mon 17/04/06
10	Testing	4 days	Sun 21/05/06	Wed 24/05/06
6	Peter away	31 days	Sat 27/05/06	Fri 30/06/06
7	Finish first draft of report	0 days	Fri 02/06/06	Fri 02/06/06
12	Fisnish second draft of report	0 days	Fri 09/06/06	Fri 09/06/06
13	Delivery of report	0 days	Fri 16/06/06	Fri 16/06/06

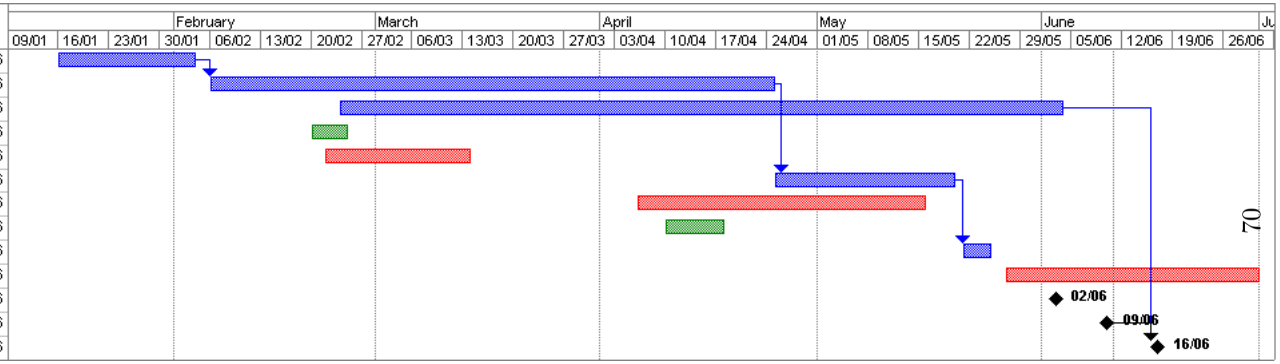

Figure B.3: Gantt diagram as of May 23rd.

Appendix C

Screenshots from SAMe

This section shows and describes some screenshots from the finished prototype, seen in Figures C.1 and C.2. Compared to the designed user interfaces in Section 5.5, they are quite similar.

Figure C.1 shows the window where the baseline configuration is acquired. This can be done manually by inserting one component at the time, or by reading a model from a file (or both). This file will be created by SP Light on the basis of an SP model. The properties of the components are determined by pressing the ”Properties” buttons.

Figure C.2 shows the window where the tuning is performed. The system structure is displayed to the left, and the properties for the selected component are displayed to the right. The dimension can be changed, and the gain is displayed beneath the requirement. The selected component can be upgraded or replicated by pressing the ”Upgrade” or ”Replicate” button.

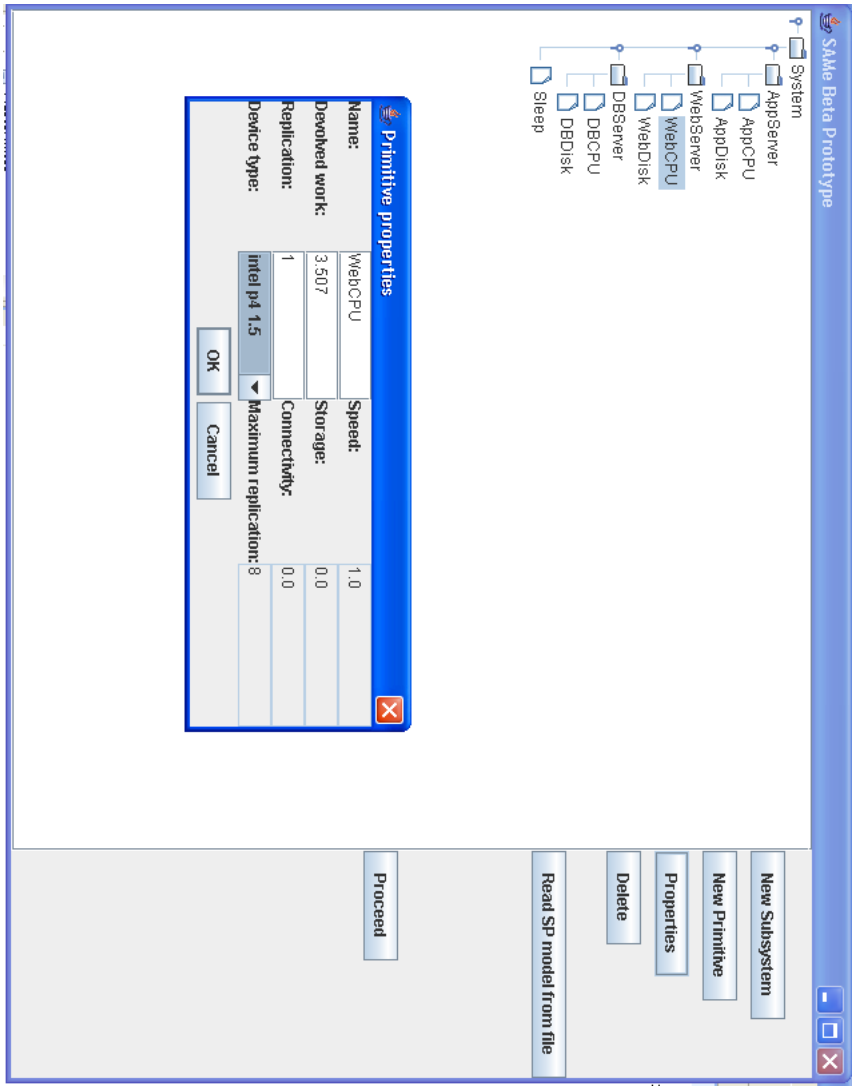

Figure C.1: The user interface for the system setup.

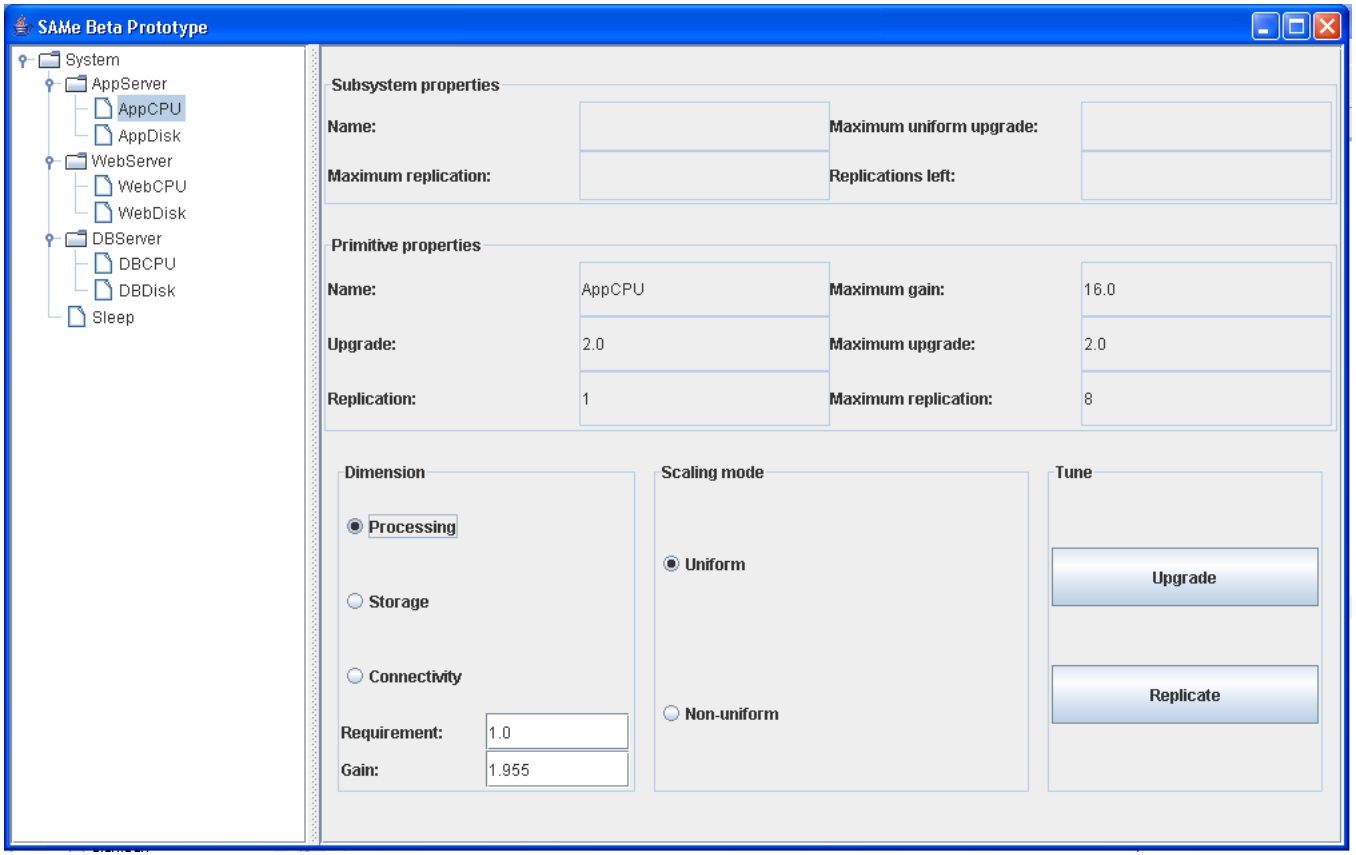

Figure C.2: The user interface for the tuning.

Appendix D

Test data

The data in Table D.1 shows the parameters used in the test of SAMe.

Resource	**Service demand**
App CPU	9,208s
App Disk	0,172s
KBM CPU	3,507s
KBM Disk	1,679s
FD CPU	2,520s
FD Disk	0,031s
Think time	812,00s

Table D.1: Parameters for the baseline system

Appendix E

Attached files

The following files are attached with this report, and are a part of the Master's Thesis:

- Start.bat - The file that starts the SAM Engine program, with a terminal window
- SAMe.jar - Runnable Java program. Requires that JavaSE5.0 are installed.
- DB.sql - The database script
- input.txt - The file containing an SP model
- SAMe.zip - All the source code in a zipped file for convenience

Use Start.bat for starting the program, but make sure the database is available.

Bibliography

[BH04] Gunnar Brataas and Peter H. Hughes. Exploring architectural scalability. In *WOSP*, pages 125–129, 2004.

[Car98] S. Carlsen. Action port model: A mixed paradigm conceptual workflow modeling language. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems, New York City, New York, USA, August 20-22, 1998, Sponsored by IFCIS, The Intn'l Foundation on Cooperative Information Systems*, pages 300–309. IEEE Computer Society, 1998.

[Che] Chandrakanth Chereddi. Implementation of the MVA algorithm in C (Open source). `http://www.crhc.uiuc.edu/~cchered2/mva.html` Last visited 2006-03.

[Des] DESMO-J. `http://asi-www.informatik.uni-hamburg.de/desmoj` Last visited 2005-12.

[Ecl] Eclipse. `http://www.eclipse.org` Last visited 2005-12.

[FL03] John-Arne Fagerli and Odd Christian Landmark. Scalability of a platform for financial services based on the J2EE architecture. Master's thesis, NTNU, 2003.

[FS00] Martin Fowler and Kendall Scott. *UML Distilled*. Addison-Wesley, 2 edition, 2000.

[Hol05] Anders Johan Holmefjord. SIMSAM: A simulation framework for the SAM method. Technical report, NTNU, 2005.

[Hug88] Peter H. Hughes. SP Principles. 059/icl226/0, STC Technology Limited, July 1988.

[Hug99] Peter Hughes. On the scalability and stability of multiple agent systems. Technical report, 18. November 1999. Version 2.3.

[Hug05] Peter Hughes. Lecture Notes in Performance Engineering, 2005.

[Hug06a] Peter Hughes. Towards a theory of scalability for computing systems with evolving requirements. Technical report, 10 February 2006. Version 0.4.

[Hug06b] Peter H. Hughes. SAM Engine - General Requirements for Prototype, 3 2006. Internal working paper.

[Java] Java 2 Standard Edition 5.0 API. `http://java.sun.com/j2se/1.5.0/docs/api/` Last visited 2006-06.

[Javb] Open source charting & reporting tools in java. `http://java-source.net/open-source/charting-and-reporting` Last visited 2006-6.

[Løv04] Jakob Sverre Løvstad. Design requirements for an SP toolset. Master's thesis, NTNU, 2004.

[MA01] Daniel A. Menascé and Virgilio A. F. Almeida. *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall, 2001.

[MAD94] Daniel A. Menasce;, Virgilio A. F. Almeida, and Larry W. Dowdy. *Capacity planning and performance modeling: from mainframes to client-server systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[MAD04] Daniel A. Menasce, Virgilio A. F. Almeida, and Larry W. Dowdy. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, 2004.

[Mic] Sun Microsystems. The Java Native Interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html` Last visited 2006-03.

[MVA] Implementation of the MVA algorithm in Java (Open source). `http://sysmod.icb.uni-due.de/index.php?id=19` Last visited 2006-05.

[MVC] Java blueprints: Model-View-Controller. `http://java.sun.com/blueprints/patterns/MVC-detailed.html` Last visited 2006-06.

[MyS] MySQL AB. `http://www.mysql.com/` Last visited 2006-06.

[RM05] Erik Rød and Erlend Mongstad. Model-driven measurement of a bank system. Master's thesis, NTNU, 2005.

[SF] Free Software Foundation. GNU Public Licence. `http://www.gnu.org/copyleft/gpl.html` Last visited 2006-03.

[spl] SP light. `http://splight.idi.ntnu.no` Last visited 2006-06.

[UML] Unified Modeling Language. `http://www.uml.org` Last visited 2006-05.

[XML] Extensible Markup Language. `http://www.w3.org/XML/` Last visited 2006-06.