

# Applying SPE Techniques for Modeling a Grid-enabled JAVA Platform

Mariela Curiel  
Computer Science  
Department  
Simón Bolívar University  
Caracas, Venezuela  
mcuriel@ldc.usb.ve

M. Angelica Perez  
Information Systems  
Department  
Simón Bolívar University  
Caracas, Venezuela  
mov@usb.ve

Ricardo Gonzalez  
Computer Science  
Department  
Simón Bolívar University  
Caracas, Venezuela  
rgonzalez@ldc.usb.ve

## ABSTRACT

Advances in Internet and the availability of powerful computers and high-speed networks have propitiated the rise of Grids. The scheduling of applications is complex in Grids due to factors like the heterogeneity of resources and changes in their availability. Performance models provide a way of performing repeatable and controllable experiments for evaluating scheduling algorithms under different scenarios. This article describes the development of a performance model for a JAVA based distributed platform using a SPE methodology. Our case study is SUMA, a distributed execution platform implemented on top of Grid services. It was necessary the use of Software Performance Engineering techniques for understanding and modeling the system. We applied a software performance methodology where Layered Queuing Network (LQN) models are derived from Use Case Maps (UCM). At the end we obtained a performance model of SUMA and a better-documented system.

## Categories and Subject Descriptors

C.4 [Computer Systems Organization]: Performance of Systems—*modeling techniques*

## General Terms

Performance, Design

## Keywords

Computational Grids, Software Performance Engineering, Performance Evaluation, Layered Queuing Network Models, Use Case Maps

## 1. INTRODUCTION

Advances in Internet and the availability of powerful computers and high-speed networks are changing the way that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WOSP '05 Palma de Mallorca, Balears Spain  
Copyright 2004 ACM .

large-scale parallel and distributed computing is being done, leading to what is popularly called the Grid ([6], [10]). In Grids the resources are usually geographically distributed in multiple administrative domains, managed and owned by different organizations and interconnected by wide-area networks and Internet. Globus [10] is the most important Grid model defined so far.

SUMA (Scientific Ubiquitous Metacomputing Architecture) [7] (<http://suma.ldc.usb.ve>) is a distributed platform implemented on top of Grid services that transparently executes JAVA bytecode on remote machines. The goal was to extend the JAVA Virtual Machine model to provide seamless access to distributed high performance resources. SUMA middleware is built on top of commodity software and communication technologies, including CORBA. We have a prototype of SUMA running in our Campus. Currently, this prototype is being exhaustively tested before putting it at other institutions disposal. The aim of our research is to develop a performance model of SUMA. The following reasons justify the development:

- *To evaluating scheduling algorithms:* The scheduling in Grids is complex and critical to achieve high performance [21]. Some of the factors that contribute to the complexity are the heterogeneity and changes in resources availability. In order to prove the effectiveness of scheduling algorithms, their performance should be evaluated under different scenarios such as varying number of resources, users requirements, load characteristics, etc. In Grids, it is impossible to get repeatable results from experiments due to the constantly changing nature of compute and network resources. Performance models get around this problem by providing a way of performing repeatable and controllable experiments. Through modeling we can develop and compare algorithms and techniques for the efficient use of the Grid. Some packages have been developed for modeling Grid environments and evaluating scheduling algorithms (Bricks[1], SimGrid[8] and GridSim[6]). These frameworks are based in the simulation technique.

Despite the existence of these tools we made a bet on our own model. In this way we can study the possibility of analytical solutions, which can provide predictions quickly, thereby enabling a large number of experiments to be conducted on the models. On the

other hand, tools like *Bricks* or *SimGrid* do not allow us to model the software architecture as it can be done with the Layered Queuing Network approach. The layering makes the model appropriate for describing complex distributed systems.

- *To validate the performance aspects of the design and identify problems such as bottlenecks.* SUMA is still in testing phase, so we have got time to correct design problems that could affect the performance.
- *To design distributed applications that will use SUMA as execution platform.*

Due to the size of SUMA and the lack of documentation it was necessary to adopt a Software Performance Engineering methodology. SUMA is a complex software that contains thousands of lines of source code, developed by many designers and programmers. Until now the developers have been focused in the functionality neglecting the up-to-date documentation and the design models. In this sense the application of Software Performance Engineering techniques allowed us a better understanding of the system, which is necessary to develop the performance model. As a result, we also obtained a better-documented system.

In this article we describe the criteria for selecting a Performance Engineering methodology suitable for SUMA. A Queuing Network based methodology was chosen. It derives Layered Queuing Network (LQN) performance models from system scenarios described by means of Use Case Maps (UCM). The methodology was applied and we discuss the results obtained in each step. We achieved a performance model of SUMA, which was then parameterized. The model represents the execution of sequential applications in the Grid. Sequential applications can be multithreaded. We use a JAVA benchmark for obtaining model parameters.

This paper is organized as follows. In Section 2 we present the criteria to choose the Software Performance Engineering methodology. The case study is described in Section 3. Sections 4 to 10 discuss the results obtained in each step of the methodology. The final Section summarizes the paper along with suggestions for future work.

## 2. METHODOLOGY FOR SOFTWARE PERFORMANCE ENGINEERING

Nowadays, many software products fail to meet their performance objectives when they are initially constructed. If the performance analysis is postponed until the end of the development, it may not be possible to fix performance problems without redesign and re-implementation, specially in complex systems. In the last decade, several research efforts have been directed to prevent these problems by integrating performance analysis in the process of software development. Several approaches have been successfully applied, but there is still a gap to be filled in order to see performance analysis integrated in ordinary software development [3].

Balsamo et al [3] review the recent developments in the field. The report presents different methodologies for software performance and groups them by the underlying performance model: queuing networks, petri nets, process algebras and simulation. In order to analyze the performance of the computational Grid SUMA and construct a performance model, we review the methodologies presented in the article.

We choose the approach of Woodside et al [16] that derives Layered Queuing Network Models (LQNM) from systems scenarios described by means of Use Case Maps (UCM). The following reasons support the choice:

- LQNM have been proposed for modeling complex distributed systems that run on networks and multiprocessors [22]. Much of this software have separate tasks which communicate by a request-wait-reply pattern. This is the pattern of the classic RPC (Remote Procedure Call), which is the basis of a wide variety of distributed system technology (DCE, CORBA, ANSA). The communication of SUMA components follows this pattern and it is CORBA-based. The LQNM has been used for modeling CORBA [20], CORBA-based applications [18] and other kind of complex distributed systems [15].
- We look for a methodology with automated support, better if the tools are free for researching purposes. The Woodside methodology is characterized by the availability of tools that automate the process. The UCM Navigator [14] was developed as an editor of UCM and repository manager. The UCM2LQN converter was added to UCMNav and generates a file with the LQN Model. There are two tools developed in Carleton University that can be used to solve LQN models and get performance metrics: The Layered Queuing Network Solver (LQNS) solves the model analytically [11], whereas the ParaSol Stochastic Rendez-Vous Network Simulator (ParaSRVN) [12] uses simulation techniques.
- It should be possible to apply the methodology in a system whose development is in advanced state, i.e., we are going to apply the methodology for doing reverse-engineering. In reverse-engineering most tools are currently limited to extracting static structural models. The recovery of dynamic behavior is still scarce. Authors of [2] suggest several approaches to the extraction of behavior aspects, in the form of UCM.

The approach presented in [16] focuses in deriving scenarios from UCM but do not addresses other important steps in the development of responsive software such as the identification of “critical” use cases or “critical” scenarios. In this sense, the software performance engineering (SPE) methodology proposed by Smith [19] is more complete. Finally we built a methodology adapted to our goals which combines the Woodside approach with elements of Smith methodology [19] and some steps proposed by Menasce et al [13] for constructing performance models. In the next paragraphs we explain each activity and justify its usefulness in our study case.

1. **Identify critical use cases:** The critical use cases are those that are important to responsiveness as seen by users, or those for which there is a performance risk. We use UML use case diagrams to represent the critical use cases of SUMA. We decide to include this step because use case diagrams generate important documentation of the system, which did not exist at the beginning of our research.
2. **Select key performance scenarios:** The key performance scenarios are those that are frequently ex-

cuted, or those that are critical to the performance of the system. The scenarios are represented using Use Case Maps (UCM). Despite workload parameters can be inserted in the UCM, we decide to add them later in the performance model. Due to the complexity and the lack of documentation we prefer to focus in understanding and modeling the system correctly with UCM, deferring the parameterization phase until the culmination of the model. During the UCMs construction we only introduce fictitious parameters.

3. **Establish performance objectives:** Performance objectives should be identified for each scenario selected. Performance objectives specify quantitative criteria for evaluating the performance characteristics of the system under study. Objectives can be expressed in several different ways, including response time, throughput, or constraints on resource usage. Although we do not apply this step in the work described in the article, it was included in the methodology because of its usefulness in future work. The establishment of performance objectives will be necessary, for example, in the evaluation of scheduling algorithms. However it is important to remark that it could be senseless to fix response times in Grids, due to their heterogeneity and dynamic nature. It is more meaningful to establish goals related to resource usage because users that put resources at Grid disposal can impose restrictions on their use.
4. **Construct performance models:** In this step the LQN models of the systems are built. The derivation of LQN models from UCMs is quite direct, due to the close correspondence between UCM and LQN basic elements.
5. **Determine model parameters:** The input parameters of a QNM fall into one of three categories: workload, basic software parameters and hardware parameters. The workload parameters describe the load imposed on the computer system by the entities submitted to it (processes, transactions, message, etc). The basic software parameters describe features of the basic software -such as the operating system and the middleware- that affect performance. The hardware parameters characterize the computers and the networks used to interconnect them. The representativeness of a model depends directly on the quality of its input parameters. The main source of information for determining input parameters is the set of performance measurements collected from the observation of the real system under study. In our case it is possible to obtain parameters by means of measurements because we have a functional prototype of SUMA. Also we have included additional instructions in SUMA to get performance measurements [9]. These parameters are architecture dependent. In this article we do not present a workload characterization process to parameterize the model. We only take measurements from a JAVA benchmark aimed at clearly understanding the mapping between performance data provided by SUMA and data requested by the LQN model.
6. **Validate the Model:** One purpose of building a performance model is to be able to carry out an analysis of

the effect on the performance measures of the variation in the parameters values. This process is called modification analysis by Menasce et al [13]. But before the modification analysis can be executed, it is necessary to build a performance model that reflects the action of the current workload on the actual system, this model is so called the baseline model. The baseline model must be validated and calibrated to guarantee that it reflects the system being modeled with reasonable accuracy. The validation and calibration activities are done by comparing model predictions with their actual values obtained through measurements done in the actual systems.

## 3. THE SYSTEM UNDER STUDY: SUMA

### 3.1 SUMA

SUMA (Scientific Ubiquitous Metacomputing Architecture) [7] is a distributed platform that transparently executes JAVA bytecode on remote machines, with additional support for scientific computing development. SUMA middleware is object oriented and built on top of commodity technologies like JAVA and CORBA; JAVA provides a portable, secure and clean object oriented environment for application development. SUMA executes three kinds of code: sequential JAVA bytecode, parallel JAVA bytecode and SUMA native code.

#### 3.1.1 Execution Model

The basics of executing JAVA programs in SUMA are simple. A user invokes execution of a program through the services *suma Execute* (on-line execution mode) or *suma Submit* (off-line execution). Once SUMA receives the request from the client machine, it transparently finds a platform for execution and sends a request message to that platform. An *Execution Agent* at the designated platform starts execution of the program. In case of *suma Execute* service the user only has to specify the program main class. The rest of the classes and data files are read from the Client at run-time, on demand from the executing class; the output is sent back. For *suma Submit* service the client machine packs classes and input files and delivers these to SUMA; the output is kept in SUMA until the user requests it. A number of execution attributes can be optionally introduced along with the program with both commands, such as scheduling constraints and platform specifications. Figure 1 depicts the sequence of method invocations inside SUMA for *execute* command. Boxes in Figure represent the main modules of SUMA Core and arrows represent invocations to methods.

When a Client wants to send an execution request to SUMA, it firstly must find a Proxy, by invoking the *find-Proxy* method in Scheduler. The Scheduler finds an appropriate Proxy and returns a CORBA reference to the Client (step 1 in Figure 1). Then the Client invokes the *execute* method in its Proxy (step 2), passing the name of the main class as a parameter. This Proxy invokes user authentication methods in User Control (step 3) and asks for a suitable Execution Agent in Scheduler (step 4), getting a CORBA reference for the Execution Agent. Then the Proxy invokes method *execute* in the selected Execution Agent (step 5), passing all necessary information for the Execution Agent to start loading classes and files from the Client Stub. This is done by invoking appropriate methods directly in the Client

Stub (step 6). The Execution Agent sends output files to the Client Stub.

Before any *execute* or *submit* method is invoked, essential SUMA objects must register themselves in the CORBA name server. Additionally, Execution Agents and Proxies must register themselves in the Scheduler. The invocations of registration methods are not shown in Figure 1.

### 3.2 SUMA Architecture

The main components of SUMA Core were shown in Figure 1. The Core runs in each administrative domain. A description of SUMA Core modules follows.

- **The Client Stub (CS)** is a library for SUMA Client's implementation, which provides services for on-line and off-line execution. The Client Stub creates the application object, retrieves results and performance data and serves Execution Agents callbacks to load classes and data dynamically.
- **A Proxy (P)** has basically two functions. First, it connects a Client with an Execution Agent in such a way that these components communicate directly with each other. The second function of a Proxy is to implement the *submit* service. If a program terminates abnormally, the Proxy requests a new node and restarts execution.
- **The Scheduler (SCH)** implements methods for Proxy and Execution Agent selection. The Scheduler keeps information about SUMA resources, i.e., execution platforms description in terms of their type (clusters, network of workstations, workstations, PCs, etc.), relative power, memory size, available libraries, and average load. The Scheduler is static because it bases its decisions on information stored in local data structures about the characteristics of execution platforms and application requirements. Other kind of scheduling strategies base its decisions on the load of each resource at the time the application will execute. This information is obtained by means of predictive models. In this case, the scheduling algorithm involves several communications among the Core and the Execution Agents. These are called dynamic schedulers. Currently, we are including dynamic strategies in SUMA using the Network Weather Service (WS) [21], but we have not modeled this architecture yet.
- **The User Control (UC)** is in charge of user registration and authentication. It handles capabilities, which control access to resources.
- **The Execution Agent (EA)** receives an order from a Proxy to execute an application. During the execution classes and files are dynamically loaded from the Client. For parallel platforms, the Execution Agent plays the role of the front end. The resources where the Executions Agents run are called nodes or execution platforms.

The architecture that enables to cross domain boundaries contains additional components: the **DRBrokers** (Domains Resource Brokers). When SUMA cannot find an adequate execution platform in the local domain it calls to DrBrokers for requesting a remote Scheduler. The DrBroker selects one Scheduler in an administrative domain where an

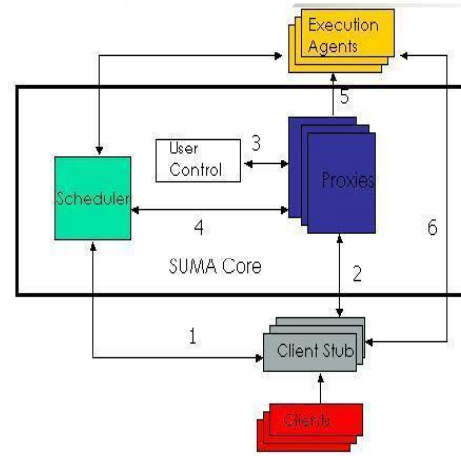


Figure 1: SUMA Core Architecture

adequate execution platforms exist. After that, local and remote Scheduler communicate each other to connect the Client with the remote Execution Agent that will execute the application. The complete architecture was modelled with UCMs.

DrBrokers also use static information for choosing probable execution platforms.

The next sections describe the results obtained after applying the main steps of the methodology.

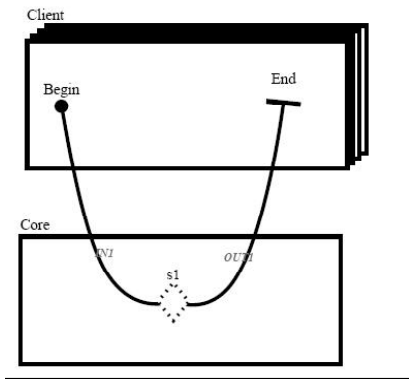
## 4. IDENTIFY CRITICAL USE CASES

Most use cases were identified and represented using UML diagrams. A few of them had already been identified as part of requirements analysis. The critical use case was undoubtedly the *execute* command. It corresponds to the on-line execution, whose performance is directly perceived by users. Users that use *submit* command are not interested in performance. The other set of commands are the administrative ones, which are less frequent and are not seen by end-users.

## 5. THE KEY PERFORMANCE SCENARIO

Each use case consists of a set of scenarios that describe the sequence of actions required to execute the use case. Only the scenarios associated to the normal execution of the critical use case (the *execute* service) are presented in this article. The methodology indicates the representation of scenarios by means of UCM. The UCM Navigator (UCMNav) [14], version 2.2.1 for Microsoft Windows was used.

A Use Case Map is a collection of elements that describe one or more scenarios unfolding throughout a system [5][4]. The main elements of the notation are shown in Figures 2, 3 y 4. Figure 2 shows the top level UCM of SUMA, with the Client sending a request to the SUMA Core. The basic building block of the UCM notation is the path, which is the visual representation of a scenario. A path is a line with a start point (a filled circle) and an end point (a bar). UCM paths can be overlaid on components representing functional or logical entities that are encountered during the execution of a scenario. They can represent both hardware and software resources in a system. In the UCM notation, teams



**Figure 2: The top level Use Case Map showing the service request**

(rectangles) are allowed to contain components of any type. In Figure 2 we observe two components: one representing the SUMA Core and other representing multiple Clients (Stack) that submit requests. The term Stack is used for multiple instances. The path crosses over the SUMA Core by the stub **S1**. Stubs can be used to hierarchically refine a path. They represent separately specified maps called plug-ins. Static stubs (plain diamonds) contain only one plug-in. Dynamic stubs (dashed plug-ins) may contain several plug-ins, whose selection can be determined at run-time.

Figure 3 shows the **execute** plug-in map for the **S1** stub. It introduces responsibilities, processes, objects and slots. Responsibilities, denoted with a X-shaped mark on the map, represent functions that need to be accomplished at given points in the execution of the scenario. Parallelograms are active components (processes) whereas rounded rectangles are passive ones (objects). Dashed components are called slots and may be populated with different instances at different times. Slots are containers for dynamic components in execution. In the Execution Agent, one or several Slaves (virtual machines) are dynamically created for executing the application. Dynamic components are created by means of dynamic responsibilities; arrows represent the dynamic creation and destruction of Slaves. The meaning of stubs and responsibilities in the map can be observed in Table 1. Responsibilities **r5**, **r10**, **r17** and **r18** represent the load introduced in the network layer.

All the plug-in maps have been developed. As it is impossible to show all of them in this article, we choose the most representative: Executing the Application (stub **s14**). Figure 4 shows the main activities related to the execution of an application. The map presents two new elements: the OR-fork and the OR-join. An OR-fork splits the path into two (or more) alternatives while an OR-join merges two (or more) overlapping paths. We observe a loop where the application executes instructions (responsibility **r14.1**) and afterward, either loads classes from Client (responsibilities **LC.1**, **LC.2** and **LC.3**) or makes I/O operations (responsibilities **I/O.1**, **I/O.2** and **I/O.3**). After concluding the network operation the application either continues the execution or it finishes. The alternatives and shared segments on routes are represented as overlapping paths.

For the purposes of performance model construction we decide to start with a simplified version of the Suma Archi-

itecture. Accordingly, we made some changes in the map of Figure 3 and obtained the map of Figure 5. On the basis of this second map the LQN model is built. The simplifications are:

- We will only model the local architecture, i.e., the architecture of Figure 1. Consequently the **Remote Execution** stub was removed and the **s8** stub was replaced by a responsibility that contains the time of looking for an execution platform in the same administrative domain.
- All the SUMA Core components run in the same computer. This allows us to substitute communications across the network layer in stub **s7** for a single responsibility that contains the communication time between the Proxy and the User Control.
- Stubs **s11**, **s15** hide just one network operation between the Execution Agent and the Client (**s11**) or the Execution Agent and the SUMA Core (**s15**). Because it is about just two network operations we decide to put their associated overhead into two responsibilities.

## 6. THE PERFORMANCE MODEL

### 6.1 LQN Approach

Queueing Network Models (QNM) are an abstraction for computer systems that have been used since the early 1970's. These models are appropriate for describing contention to the physical resources (CPU, Memory, Disk, Network bandwidth), but not capture the performance impact of access to logical resources (thread, pools, etc).

LQN defines a system in terms of its objects (software and hardware). LQMs are QNM extended to reflect interactions between client and server processes. In ordinary queuing networks there is one layer of servers; in LQN, servers may submit requests to other servers, with any number of layers. This layered models can be solved either by simulation, or by analytic approximations. The Method of Layers (MOL) [17] and Stochastic Rendezvous Network (SRVN) [15] techniques have been proposed as performance evaluation techniques that estimate the performance behavior of LQNs.

### 6.2 LQN Notation

An LQN can be represented by a graph with nodes for tasks and devices, and arrows for service requests. A task is a software object that has its own thread of execution. Tasks are divided into three categories: client tasks (only sends requests), active server tasks (can receive and send requests) and pure server tasks (only receive requests). The hardware resources are called devices (CPUs, disks, etc). There are three types of arrows that represent asynchronous messages, with no reply, synchronous interactions which block the sender until there is a reply, and forwarding messages that combine synchronous and asynchronous behavior. In a forward, the sending client task makes a synchronous call and blocks until it receives a reply. The receiving task partially processes the call and then forwards it to another server which becomes responsible for sending a reply to the blocked client task. Figure 6 shows the visual notation for the main elements.

Tasks receive either kind of request message in points called entries. A task has a different entry for every kind of

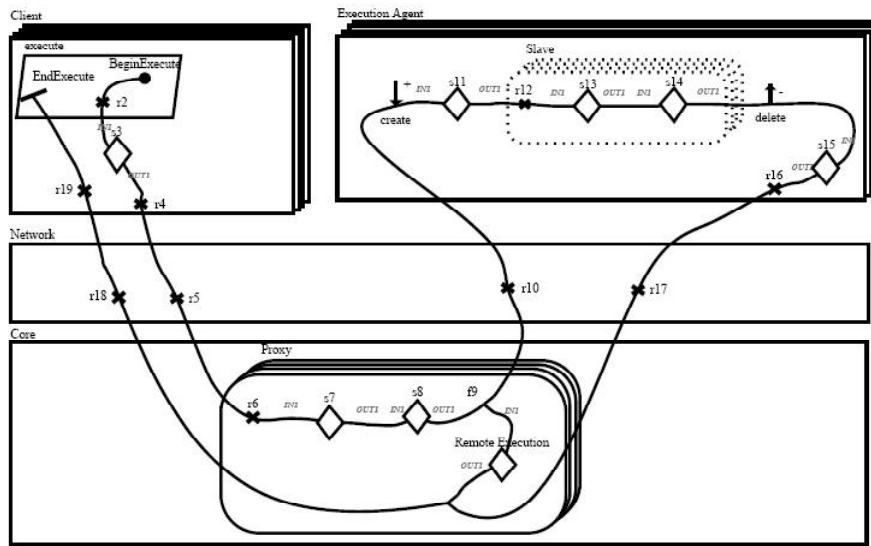


Figure 3: The Use Case Map for the *execute* service

Table 1: Stubs and Responsibilities in the *execute* plug-in

r2	Getting user parameters
s3	It hides the steps for requesting a Proxy to the Scheduler ( <i>findProxy</i> method)
r4	Requesting the execution of an application to the Proxy
r5	Communication Client-SUMA Core
r6	Generating an application ID
s7	It hides the communication with the User Control to verify the User ID
s8	It conceals the steps for requesting an execution platform
f9	Requesting the execution to the selected node or platform (local or remote)
Remote execution	It hides the execution in a remote administrative domain
r10	Communication SUMA Core-Execution Agent
s11	It hides preliminary connections with the Client
r12	Initiating the execution environment
s13	It hides the process of loading the main class
s14	It hides the execution of the application
s15	It conceals the end of the execution
r16, r17, r18 and r19	Processing and returning results

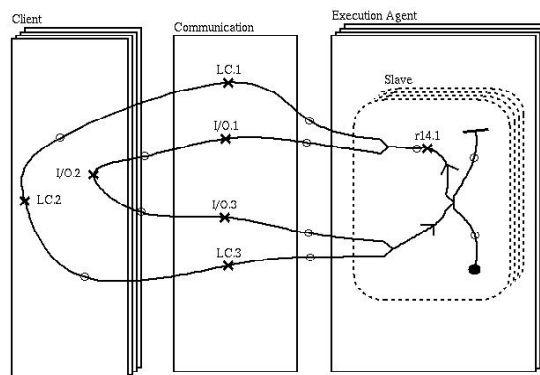


Figure 4: The execution of a sequential application represented in an UCM

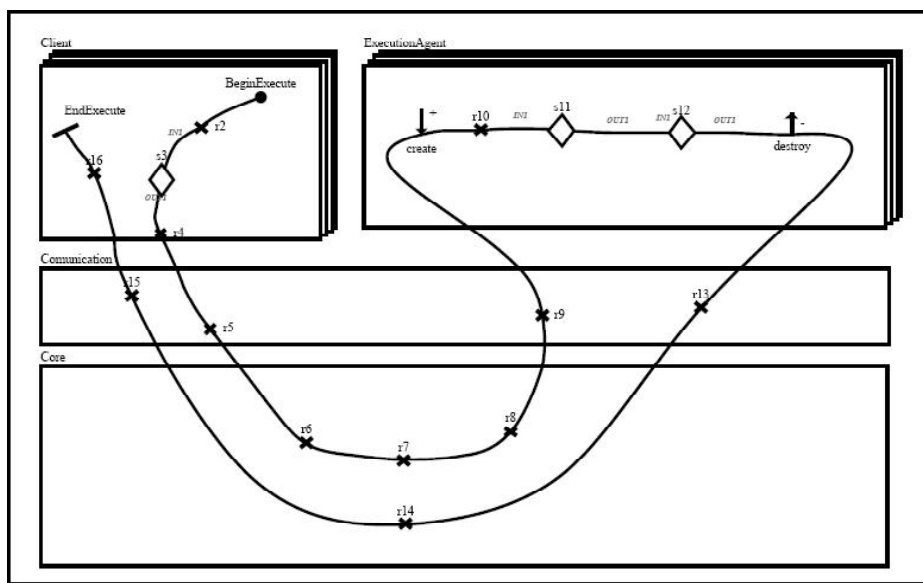


Figure 5: The Use Case Map for the simplified version of the *execute* service

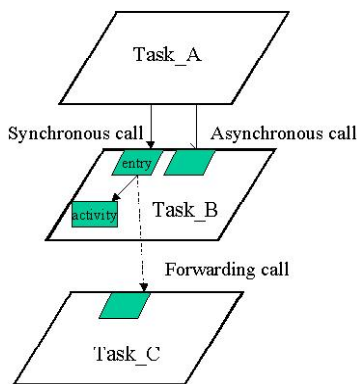


Figure 6: LQN basic elements

service it provides; an entry also represents a class of service. Internally an entry can be composed by sequences of smaller computational blocks called activities, which are related in sequence, loop, parallel (AND fork/joins) and alternative (OR fork/joins) configurations. Activities have processor service demands and generate calls to entries in other tasks.

### 6.3 Converting the UCM to an LQN

We generate a LQN model using the UCMNavigator tool. The resultant lqn file had some “null” words corresponding to the starting activities of some entry points<sup>1</sup>. Since the tool does not provide information about the cause of “null” words, we decide to replace them manually. The work of understanding the model for replacing “null” words was laborious. We could have introduced bugs during the replacement process that hindered the solution of the model. The understanding of the model is difficult because of the complexity of the represented system and the amount of additional activities that are introduced in the generation process.

<sup>1</sup>When an entry point has activities it is necessary to define the initial one.

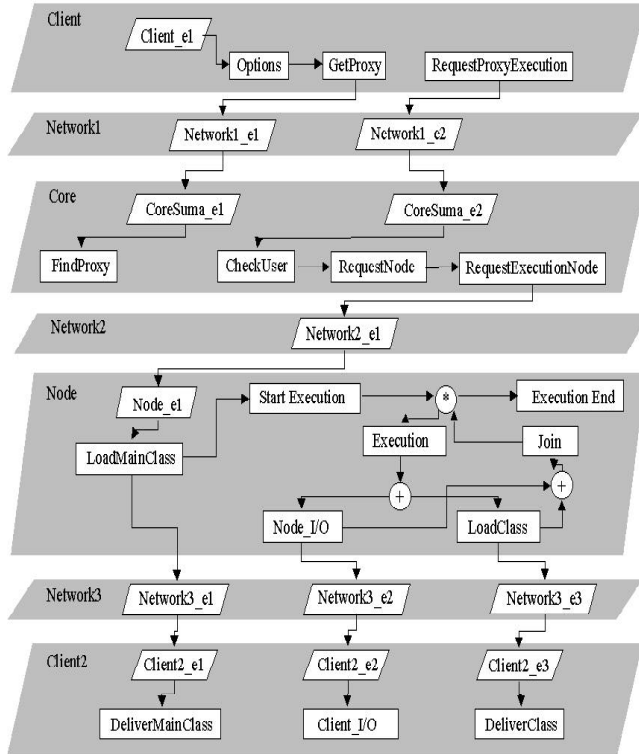
Finally, we opted for developing our own model using the correspondences described in [16]. This model has the main elements of map in Figure 5.

The derivation of LQN models from UCM is quite direct, due to the close correspondence between UCM and LQN basic elements. Table 2 shows the correspondence between UCM and LQN constructs. Figure 7 shows the resultant LQN model. By comparing the map and the LQN model we observe:

- The number of tasks in the LQN is greater than the number of components in the UCM. We replicate the number of clients and the network components. In the real system, the Client invokes a synchronous *execute* call to run an application. Additionally, the Slave in the Execution Agent requests classes and data from the Client while it runs the application. These requests are attended by the virtual machine in the Client computer. One way of modeling this behavior is by duplicating the Client: the original Client requests the execution of the application and its copy serves to the Execution Agent. It makes the model clearer and does not affect significantly the results. Different network components provide us a clearer representation of links among Clients, the SUMA Core and the Execution Agents.
- There is a direct correspondence between the responsibilities in the UCM and activities in the LQNM. Also it is possible to observe the correspondence between other elements like the OR fork/join of Figure 4 and those in the Node task (LQN model).
- Some elements present in Figure 5 were not modeled. For example the creation of a Slave in the execution platform for executing the application. This overhead was included in the execution of the application.

**Table 2: Corresponding UCM and LQN Constructs**

UCM Construct	LQN Construct
start point	reference task
responsibility	activity
AND/OR forks and joins	LQN AND/OR forks and joins
component	task
device	device
service	entry in a task (with a dedicated processor)



**Figure 7: SUMA LQN Model**

## 7. PARAMETERIZATION AND PRELIMINARY VALIDATION

As we have a functional prototype of SUMA it is possible to obtain model parameters by means of measurements. Since SUMA is not totally installed, we do not have a real workload. Thus we do not present a workload characterization process to parameterize the model. On the other hand, this process is senseless if the model is going to be used for evaluating scheduling algorithms.

We only take measurements from one JAVA benchmark and simulate its execution in the SUMA model. This allows us to make a preliminary evaluation of the model, i.e., to test if all the elements are rightly represented. Besides we can evaluate the usefulness of SUMA monitoring tool for providing LQN model parameters. Performance data provided by SUMA could be useful if we are designing applications for the GRID. The SUMA event-based monitor gives the following data per application: total execution time, number of operations to load classes, number of I/O operations and total communication time. Also the tool provides information about the execution time and communication operations of each SUMA module.

We conducted experiments by running SUMA modules in three machines connected by a LAN (10Mbps Ethernet). The processors are pentium III (double processors), 666Mhz with 504Mb of RAM memory. The installed operating system is RedHat 6.2, kernel 2.4.21. We executed the benchmark *GGFLUFact* of JAVAGrande Forum. From the parameterization process we conclude:

- Application data provided by SUMA seems sufficient to parameterize the model.
- As regards data provided per SUMA modules, SUMA breaks it down per module but not for LQN entries or activities. So we distributed data among different activities uniformly.
- The SUMA monitoring tool does not provide information about the resource usage. This information could be useful to validate models.

After parameterizing the model we solve it with the LQN Solver. We observe that real and predicted execution times are quite similar (317 seconds and 314 seconds, respectively). The error is around 1%, which is expected in dedicated platforms. Preliminary results suggest that the model represent the SUMA system rightly and that performance data provided by SUMA is adequate for parameterizing the model. Of course, it is necessary validating the model with a larger number of benchmarks or real applications and different configurations of SUMA into the hardware platforms.

## 8. CONCLUSIONS AND FUTURE WORK

In this paper we have applied a Software Performance Engineering methodology to construct a performance model of the distributed execution platform SUMA. We reviewed recent advances in the field and chose the methodology proposed by Woodside et al, which generates LQN models from UCM. To adequate the methodology to our case study it was then combined with steps proposed by other authors ([19] and [13]).

The main obstacle that we could not overcome was the automated generation of the LQN model. However, it was very

easy to model a simplified version of the Grid by starting from the UCMs and using the method proposed by Petriu in [16]. The following results were obtained after applying the methodology:

- The SUMA system was documented.
- By applying the methodology it was possible to understand and model a complex system like SUMA. Undoubtedly the use of Use Case Maps made easy the construction of the performance model. However, we think that we think that the UCMNav tool should provide more information that help the analyst to rightly generate LQN models. The tool could still have some problems for generating complex systems models.
- We got a performance model that could be useful for evaluating scheduling algorithms, validating performance aspects of SUMA design and designing distributed applications for SUMA. It was possible to solve the model by using analytical techniques.
- We obtained a feedback about the performance data provided by SUMA. Until now, the information seems adequate for parameterizing LQN models but it is necessary to add services that provide data about resource usage; it will ease the validation of the models.

Future work it will be focused on:

- Constructing the LQN model of the complete SUMA architecture, which includes the search of execution platforms in different administrative domains and the execution of parallel applications.
- Validating the performance model with a more complex Grid architecture and a larger number of benchmarks or real applications.

## 9. REFERENCES

- [1] K. Aida, A. Takefusa, H. Nakada, S. Matsouka, and et al. Performance evaluation model for scheduling in global computing systems. In *Proceedings SC'2000*, 2000.
- [2] D. Amyot, N. Mansurov, and G. Mussbacher. Understanding existing software with use case map scenarios. *LNCS*, 2599:124–140, June 2002.
- [3] S. Balsamo, A. D. Marco, P. Inveradi, and M. Simeoni. Software Performance: State of the Art and Perspectives. Technical Report CS-2003-1, Universit Ca' Foscari di Venezia, jan 2003.
- [4] R. Buhr. Use case maps as architectural entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, 1998.
- [5] R. Buhr and R. Casselman. *Use Case Maps for Object-Oriented System*. Prentice Hall, 1996.
- [6] R. Buyya and M. Murshed. Gridsim: A toolkit for the modelling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14:1175–1220, 2002.
- [7] Y. Cardinale, M. Curiel, C. Figueira, P. García, and E. Hernández. Implementation of a corba-based metacomputing system. In *Proceeding of Workshop on Java for High Performance Computing. Lecture Notes in Computer Science*, Jun 2001.
- [8] H. Casanova, A. Legrand, and L. Marshal. Scheduling distributed applications: the simgrid simulation framework. In *Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, 2003.
- [9] M. Curiel, Y. Cardinale, C. Figueira, and E. Hernández. Services for modelling metasystem performance using queuing network models. In *Proceeding of Communication Networks and Distributed System Modelling and Simulation Conference (CNDS'2001)*, 2001.
- [10] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: enabling scalable virtual organizations. *The Int. Journal of Supercomputer Application*, 15(3), 2001.
- [11] G. Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Carleton University, 2000.
- [12] E. Mascarenhas. *A System for Multithreaded Parallel Simulation with Migrant Thread and Objects*. PhD thesis, Purdue University, 1996.
- [13] D. Menascé, V. Almeida, and L. Dowdy. *Capacity Planning and Performance Modelling*. Prentice Hall, New Jersey, 1994.
- [14] A. Miga. Applications of Use Case Maps to System Design with Tool Support . Master's thesis, Carleton University, 1998.
- [15] D. Petriu, C. Shousha, and A. Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *IEEE Transaction on Software Engineering*, 26(11):1049–1065, Nov. 2000.
- [16] D. C. Petriu and C. Woodside. Software performance models from systems scenarios in use case maps. *Proceeding of TOOLS, Springer Verlag LNCS*, 794:159–177, 2002.
- [17] J. Rolia and K. C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):682–688, 1995.
- [18] F. Sheikh, J. Rolia, P. Garg, S. Frolund, and A. Shepherd. Layered performance modeling of a distributed application design. In *Proceedings of the 1st World Congress on Computer Simulation*, September 1997.
- [19] C. Smith and L. Williams. *Performance Solutions. A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, New York, 2002.
- [20] T. Verdickt, B. Dhoedt, F. Gielen, and P. Demeester. Modelling the performance of corba using layered queuing networks. In *Proceedings of 29th Euromicro Conference (EUROMICRO'03)*, September 2003.
- [21] R. Wolski. Dinamically Forecasting Network Performance using The Network Weather Service. Technical Report TR-CS-96494, California University, San Diego, 1998.
- [22] C. Woodside, S. Majumdar, J. Neilson, D. Petriu, J. A. Rolia, A. Hubbard, and R. Franks. *A Guide to Performance Modelling of Distributed Client-Server Software Systems with Layered Queuing Networks*. Carleton University, November 1995.